SCons.Builder module
SCons.CacheDir module
SCons.Conftest module
SCons.Debug module
SCons.Defaults module
SCons.Environment module
SCons.Errors module
SCons.Executor module
SCons.Memoize module
SCons.PathList module
SCons.SConf module
SCons.SConsign module
SCons.Subst module
SCons.Warnings module
SCons.cpp module
SCons.dblite module
SCons.exitfuncs module
SCons.compat package
SCons.Node package
SCons.SConf module
SCons.SConsign module
SCons.Subst module
SCons.Warnings module
SCons.cpp module
SCons.dblite module
SCons.exitfuncs module
SCons.Node package
SCons.SConf module
SCons.SConsign module
SCons.Subst module
SCons.Warnings module
SCons.cpp module
SCons.dblite module
SCons.exitfuncs module
SCons.Node.Alias module
SCons.Node.FS module
SCons.Node.Python module
Module contents
SCons.Script package
SCons.Script.Interactive module
SCons.Script.Main module
SCons.Script.SConscript module
SCons.Script package
SCons.Script.Interactive module
SCons.Script.Main module
SCons.Script.SConscript module
SCons.Script package
SCons.Script.Interactive module
SCons.Script.Main module
SCons.Script.SConscript module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SCons.Taskmaster.Job module
SCons.Taskmaster package
SConPsi...
SCons API Documentation

Attention!

This is the **internal** API Documentation for SCons. The documentation is automatically generated for each release from the source code using the Sphinx documentation generator. Missing information is due to shortcomings in the docstrings in the code, which admittedly could use a lot more work (contributions welcomed!).

The target audience is both developers working on SCons itself, and those writing external Tools, Builders, etc. and other related functionality, who need to reach beyond the Public API. Note that what is Public API is not clearly deliniated in the API Docs. The interfaces available for use in SCons configuration scripts (“SConscript files”), which have a consistency guarantee, are those documented in the **SCons Reference Manual**.

SCons package

Module contents

Subpackages

SCons.Node package

Submodules

SCons.Node.Alias module

Alias nodes.

This creates a hash of global Aliases (dummy targets).

```python
class SCons.Node.Alias.Alias (name)
    Bases: Node
class Attrs
    Bases: object
    shared
BuildInfo
    alias of AliasBuildInfo
Decider (function) → None
GetTag (key)
    Return a user-defined tag.
NodeInfo
    alias of AliasNodeInfo
Tag (key, value) → None
    Add a user-defined tag.
_add_child (collection, set, child) → None
    Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.
_children_get ()
_children_reset () → None
_func_exists
_func_get_contents
_func_is_derived
_func_rexists
_func_target_from_source
```
_get_scanner (env, initial_scanner, root_node_scanner, kw)

_memo

_specific_sources

_tags

add_dependency (depend)

  Adds dependencies.

add_ignore (depend)

  Adds dependencies to ignore.

add_prerequisite (prerequisite) → None

  Adds prerequisites

add_source (source)

  Adds sources.

add_to_implicit (deps) → None

add_to_waiting_parents (node) → int

  Returns the number of nodes added to our waiting parents list: 1 if we add a unique waiting parent, 0 if not. (Note
  that the returned values are intended to be used to increment a reference count, so don’t think you can “clean up”
  this function by using True and False instead…)

add_to_waiting_s_e (node) → None

add_wkid (wkid) → None

  Add a node to the list of kids waiting to be evaluated

all_children (scan: int = 1)

  Return a list of all the node’s direct children.

alter_targets ()

  Return a list of alternate targets for this Node.

always_build

attributes

binfo

build () → None

  A “builder” for aliases.

builder

builder_set (builder) → None

built () → None

  Called just after this node is successfully built.

cached

changed (node=None, allowcache: bool = False)

  Returns if the node is up-to-date with respect to the BuildInfo stored last time it was built. The default behavior is to
  compare it against our own previously stored BuildInfo, but the stored BuildInfo from another Node (typically one in
  a Repository) can be used instead.

  Note that we now always check every dependency. We used to short-circuit the check by returning as soon as we
  detected any difference, but we now rely on checking every dependency to make sure that any necessary Node
  information (for example, the content signature of an #included .h file) is updated.

  The allowcache option was added for supporting the early release of the executor/builder structures, right after a
  File target was built. When set to true, the return value of this changed method gets cached for File nodes. Like
  this, the executor isn’t needed any longer for subsequent calls to changed().

  @see: FS.File.changed(), FS.File.release_target_info()

changed_since_last_build

cHECK_ATTRIBUTES (name)

  Simple API to check if the node.attributes for name has been set
children (scan: int = 1)

  Return a list of the node’s direct children, minus those that are ignored by this node.

children_are_up_to_date () → bool

  Alternate check for whether the Node is current: If all of our children were up-to-date, then this Node was
  up-to-date, too.


clear () → None
Completely clear a Node of all its cached state (so that it can be re-evaluated by interfaces that do continuous integration builds).

```python
clear_memoized_values () → None
convert () → None
del_binfo () → None
    Delete the build info from this node.
depends
depends_set
disambiguate (must_exist=None)
env
env_set (env, safe: bool = False) → None
executor
executor_cleanup () → None
    Let the executor clean up any cached information.
exists () → bool
    Explain whether node exists.
explain ()
for_signature ()
    Return a string representation of the Node that will always be the same for this particular Node, no matter what. This is by contrast to the __str__() method, which might, for instance, return a relative path for a file Node. The purpose of this method is to generate a value to be used in signature calculation for the command line used to build a target, and we use this method instead of str() to avoid unnecessary rebuilds. This method does not need to return something that would actually work in a command line; it can return any kind of nonsense, so long as it does not change.
get_abspath ()
    Return an absolute path to the Node. This will return simply str(Node) by default, but for Node types that have a concept of relative path, this might return something different.
get_binfo ()
    Fetch a node’s build information.
    node - the node whose sources will be collected cache - alternate node to use for the signature cache returns - the build signature
    This no longer handles the recursive descent of the node’s children’s signatures. We expect that they’re already built and updated by someone else, if that’s what’s wanted.
get_build_env ()
get_build_scanner_path (scanner)
    Fetch the appropriate scanner path for this node.
get_builder (default_builder=None)
    Return the set builder, or a specified default value
get_cachedir_csig ()
get_contents ()
    The contents of an alias is the concatenation of the content signatures of all its sources.
get_csig ()
    Generate a node’s content signature, the digested signature of its content.
    node - the node cache - alternate node to use for the signature cache returns - the content signature
get_env ()
get_env_scanner (env, scanner, kw=())[kw=]()
get_executor (create: int = 1) → Executor
    Fetch the action executor for this node. Create one if there isn’t already one, and requested to do so.
get_found_includes (env, scanner, path)
    Return the scanned include lines (implicit dependencies) found in this node.
The default is no implicit dependencies. We expect this method to be overridden by any subclass that can be scanned for implicit dependencies.
get_implicit_deps (env, initial_scanner, path_func, kw=())
    Return a list of implicit dependencies for this node.
This method exists to handle recursive invocation of the scanner on the implicit dependencies returned by the scanner, if the scanner’s recursive flag says that we should.

get_ninfo ()

get_source_scanner (node)
Fetch the source scanner for the specified node

NOTE: “self” is the target being built, “node” is the source file for which we want to fetch the scanner. Implies self.has_builder() is true; again, expect to only be called from locations where this is already verified. This function may be called very often; it attempts to cache the scanner found to improve performance.

get_state ()

get_stored_implicit ()
Fetch the stored implicit dependencies

get_stored_info ()

get_string (for_signature)
This is a convenience function designed primarily to be used in command generators (i.e., CommandGeneratorActions or Environment variables that are callable), which are called with a for_signature argument that is nonzero if the command generator is being called to generate a signature for the command line, which determines if we should rebuild or not.

Such command generators should use this method in preference to str(Node) when converting a Node to a string, passing in the for_signature parameter, such that we will call Node.for_signature() or str(Node) properly, depending on whether we are calculating a signature or actually constructing a command line.

get_subst_proxy ()
This method is expected to return an object that will function exactly like this Node, except that it implements any additional special features that we would like to be in effect for Environment variable substitution. The principle use is that some Nodes would like to implement a __getattr__() method, but putting that in the Node type itself has a tendency to kill performance. We instead put it in a proxy and return it from this method. It is legal for this method to return self if no new functionality is needed for Environment substitution.

get_suffix () → str

get_target_scanner ()

has_builder () → bool
Return whether this Node has a builder or not.

In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly (“if node.builder: …”). When the builder attribute is examined directly, it ends up calling __getattr__ for both the __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.

has_explicit_builder () → bool
Return whether this Node has an explicit builder.

This allows an internal Builder created by SCons to be marked non-explicit, so that it can be overridden by an explicit builder that the user supplies (the canonical example being directories).

ignore
ignore_set
implicit
implicit_set
includes

is_conftest () → bool
Returns true if this node is an conftest node

is_derived () → bool
Returns true if this node is derived (i.e. built).

This should return true only for nodes whose path should be in the variant directory when duplicate=0 and should contribute their build signatures when they are used as source files to other derived files. For example: source with source builders are not derived in this sense, and hence should not return true.

is_explicit

is_literal () → bool
Always pass the string representation of a Node to the command interpreter literally.

is_sconscript () → bool
Returns true if this node is an sconscript

is_under (dir) → bool
is_up_to_date () → bool
Alternate check for whether the Node is current: If all of our children were up-to-date, then this Node was up-to-date, too.

linked
make_ready () → None
Get a Node ready for evaluation.
This is called before the Taskmaster decides if the Node is up-to-date or not. Overriding this method allows for a Node subclass to be disambiguated if necessary, or for an implicit source builder to be attached.
missing () → bool
multiple_side_effect_has_builder () → bool
Return whether this Node has a builder or not.
In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly ("if node.builder: ...”). When the builder attribute is examined directly, it ends up calling __getattr__ for both the __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.

new_binfo ()
new_ninfo ()
ninfo
nocache
noclean
postprocess () → None
Clean up anything we don’t need to hang onto after we’ve been built.
precious
prepare ()
Prepare for this Node to be built.
This is called after the Taskmaster has decided that the Node is out-of-date and must be rebuilt, but before actually calling the method to build the Node.
This default implementation checks that explicit or implicit dependencies either exist or are derived, and initializes the BuildInfo structure that will hold the information about how this node is, uh, built.
(The existence of source files is checked separately by the Executor, which aggregates checks for all of the targets built by a specific action.)
Overriding this method allows for a Node subclass to remove the underlying file from the file system. Note that subclass methods should call this base class method to get the child check and the BuildInfo structure.
prerequisites
pseudo
push_to_cache () → None
Try to push a node into a cache
really_build (**kw)
Actually build the node.
This is called by the Taskmaster after it’s decided that the Node is out-of-date and must be rebuilt, and after the prepare() method has gotten everything, uh, prepared.
This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().
ref_count
release_target_info () → None
Called just after this node has been marked up-to-date or was built completely.
This is where we try to release as many target node infos as possible for clean builds and update runs, in order to minimize the overall memory consumption.
By purging attributes that aren’t needed any longer after a Node (=File) got built, we don’t have to care that much how many KBytes a Node actually requires…as long as we free the memory shortly afterwards.
@see: built() and File.release_target_info()
remove ()
Remove this Node: no-op by default.
render_include_tree ()
Return a text representation, suitable for displaying to the user, of the include tree for the sources of this node.
reset_executor () \rightarrow None
    Remove cached executor; forces recompute when needed.
retrieve_from_cache () \rightarrow bool
    Try to retrieve the node’s content from a cache
    This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe
    stuff in built().
    Returns true if the node was successfully retrieved.
reexists ()
    Does this node exist locally or in a repository?
scan () \rightarrow None
    Scan this node’s dependents for implicit dependencies.
scanner_key ()
    An Alias is not recorded in .sconsign files
select_scanner (scanner)
    Selects a scanner for this Node.
    This is a separate method so it can be overridden by Node subclasses (specifically, Node.FS.Dir) that must use
    their own Scanner and don’t select one the Scanner.Selector that’s configured for the target.
set_always_build (always_build: int = 1) \rightarrow None
    Set the Node’s always_build value.
set_executor (executor: Executor) \rightarrow None
    Set the action executor for this node.
set_explicit (is_explicit) \rightarrow None
set_nocache (nocache: int = 1) \rightarrow None
    Set the Node’s nocache value.
set_noclean (noclean: int = 1) \rightarrow None
    Set the Node’s noclean value.
set_precious (precious: int = 1) \rightarrow None
    Set the Node’s precious value.
set_pseudo (pseudo: bool = True) \rightarrow None
    Set the Node’s pseudo value.
set_specific_source (source) \rightarrow None
set_state (state) \rightarrow None
side_effect
side_effects
sources
sources_set
state
store_info
str_for_display ()
target_peers
visited () \rightarrow None
    Called just after this node has been visited (with or without a build).
waiting_parents
waiting_s_e
wkids

class SCons.Node.Alias.AliasBuildInfo
    Bases: BuildInfoBase
    _getstate__ ()
        Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a
        ‘__dict__’ slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all
        instances of a class.
    _setstate__ (state) \rightarrow None
        Restore the attributes from a pickled state.
bact
bactsig
bdepends
bdepedendsigs
bimplicit
bimlicitsigs
bsources
bsourcesigs
current_version_id = 2
merge (other) → None

Merge the fields of another object into this object. Already existing information is overwritten by the other instance’s data. WARNING: If a ‘__dict__’ slot is added, it should be updated instead of replaced.

class  SCons.Node.Alias.AliasNamespace (dict=None, /, **kwargs)
Bases: UserDict

Alias (name, **kw)
_abc_impl = <_abc._abc_data object>
clear () → None. Remove all items from D.
copy ()
classmethod fromkeys (iterable, value= None)
get (k[, d]) → D[k] if k in D, else d. d defaults to None.
items () → a set-like object providing a view on D’s items
keys () → a set-like object providing a view on D’s keys
lookup (name, **kw)
pop (k[, d]) → v, remove specified key and return the corresponding value.
    If key is not found, d is returned if given, otherwise KeyError is raised.
popitem () → (k, v), remove and return some (key, value) pair
    as a 2-tuple; but raise KeyError if D is empty.
setdefault (k[, d]) → D.get(k,d), also set D[k]=d if k not in D
update ([, E], **F) → None. Update D from mapping/iterable E and F.
    If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
values () → an object providing a view on D’s values

Bases: NodeInfoBase

__getstate__ ()
Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a ‘__dict__’ slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all instances of a class.
__setstate__ (state) → None
Restore the attributes from a pickled state.
convert (node, val) → None
csig
current_version_id = 2
field_list = ['csig']
format (field_list= None, names: int = 0)
merge (other) → None

Merge the fields of another object into this object. Already existing information is overwritten by the other instance’s data. WARNING: If a ‘__dict__’ slot is added, it should be updated instead of replaced.

str_to_node (s)
update (node) → None

SCons.Node.FS module

File system nodes.

These Nodes represent the canonical external objects that people think of when they think of building software: files and directories.

This holds a “default_fs” variable that should be initialized with an FS that can be used by scripts or modules looking for the canonical default.
A generic class for file system entries. This class is for when we don’t know yet whether the entry being looked up is a file or a directory. Instances of this class can morph into either Dir or File objects by a later, more precise lookup. Note: this class does not define \_\_cmp\_\_ and \_\_hash\_\_ for efficiency reasons. SCons does a lot of comparing of Node.FS.{Base,Entry,File,Dir} objects, so those operations must be as fast as possible, which means we want to use Python\’s built-in object identity comparisons.

**Note:**

The Node lookups are relative to this Node (typically a directory), so memoizing result saves cycles from looking up the same path for each target in a given directory.

**Tag (key, value) \rightarrow None**

Add a user-defined tag.

**\_\_Rfindalldirs\_\_key (pathlist)**

Search for a list of directories in the Repository list.

**Rfindalldirs (pathlist)**

Return all of the directories for a given path list, including corresponding “backing” directories in any repositories.

**Tag (key, value) \rightarrow None**

Add a user-defined tag.

**\_\_getattr\_\_ (attr)**

Together with the node\_bwcomp dict defined below, this method provides a simple backward compatibility layer for the Node attributes \_abspath\_, \_labspath\_, \_path\_, \_tpath\_, \_suffix\_ and \_path\_elements\_. These Node attributes used to be directly available in v2.3 and earlier, but have been replaced by getter methods that initialize the single variables lazily when required, in order to save memory. The redirection to the getters lets older Tools and SConstruct continue to work without any additional changes, fully transparent to the user. Note, that \_\_getattr\_\_ is only called as fallback when the requested attribute can’t be found, so there should be no speed performance penalty involved for standard builds.

**\_\_lt\_\_ (other)**

less than operator used by sorting on py3

**\_str\_ () \rightarrow str**

A Node.FS.Base object\’s string representation is its path name.

**\_abspath**

Adds \_child\_ to \_collection\_, first checking \_set\_ to see if it\’s already present.

**\_children\_get ()**

**\_children\_reset () \rightarrow None**

**\_func\_exists**

**\_func\_get\_contents**

**\_func\_is\_derived**

**\_func\_rexists**

**\_func\_sconsign**

**\_func\_target\_from\_source**

**\_get\_scanner (env, initial\_scanner, root\_node\_scanner, kw)**

**\_get\_str ()**

**\_glob1 (pattern, ondisk: bool = True, source: bool = False, strings: bool = False)**

**\_labspath**

**\_local**

**\_memo**

**\_path**
add_dependency (depend)
    Adds dependencies.
add_ignore (depend)
    Adds dependencies to ignore.
add_prerequisite (prerequisite) → None
    Adds prerequisites
add_source (source)
    Adds sources.
add_to_implicit (deps) → None
add_to_waiting_parents (node) → int
    Returns the number of nodes added to our waiting parents list: 1 if we add a unique waiting parent, 0 if not. (Note
    that the returned values are intended to be used to increment a reference count, so don’t think you can “clean up”
    this function by using True and False instead…)
add_to_waiting_s_e (node) → None
add_wkid (wkid) → None
    Add a node to the list of kids waiting to be evaluated
all_children (scan: int = 1)
    Return a list of all the node’s direct children.
alter_targets ()
    Return a list of alternate targets for this Node.
always_build
attributes
binfo
build (**kw)
    Actually build the node.
    This is called by the Taskmaster after it’s decided that the Node is out-of-date and must be rebuilt, and after the
    prepare() method has gotten everything, uh, prepared.
    This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe
    stuff in built().
builder
builder_set (builder) → None
built () → None
    Called just after this node is successfully built.
cached
changed (node=None, allowcache: bool = False)
    Returns if the node is up-to-date with respect to the BuildInfo stored last time it was built. The default behavior is to
    compare it against our own previously stored BuildInfo, but the stored BuildInfo from another Node (typically one in
    a Repository) can be used instead.
    Note that we now always check every dependency. We used to short-circuit the check by returning as soon as we
    detected any difference, but we now rely on checking every dependency to make sure that any necessary Node
    information (for example, the content signature of an #included .h file) is updated.
    The allowcache option was added for supporting the early release of the executor/builder structures, right after a
    File target was built. When set to true, the return value of this changed method gets cached for File nodes. Like
    this, the executor isn’t needed any longer for subsequent calls to changed()
    @see: FS.File.changed(), FS.File.release_target_info()
changed_since_last_build
check_attributes (name)
    Simple API to check if the node.attributes for name has been set
children (scan: int = 1)
    Return a list of the node’s direct children, minus those that are ignored by this node.
children_are_up_to_date () → bool
  Alternate check for whether the Node is current: If all of our children were up-to-date, then this Node was
  up-to-date, too.
clear () → None
  Completely clear a Node of all its cached state (so that it can be re-evaluated by interfaces that do continuous
  integration builds).
clear_memoized_values () → None
cwd
  Delete the build info from this node.
depends
depends_set
dir
disambiguate (must_exist=None)
duplicate
env
eur_set (env, safe: bool = False) → None
executor
executor_cleanup () → None
  Let the executor clean up any cached information.
exists ()
  Reports whether node exists.
explain ()
for_signature ()
  Return a string representation of the Node that will always be the same for this particular Node, no matter what.
  This is by contrast to the __str__() method, which might, for instance, return a relative path for a file Node. The
  purpose of this method is to generate a value to be used in signature calculation for the command line used to
  build a target, and we use this method instead of str() to avoid unnecessary rebuilds. This method does not need to
  return something that would actually work in a command line; it can return any kind of nonsense, so long as it does
  not change.
fs
  Reference to parent Node.FS object
get_abspath ()
  Get the absolute path of the file.
get_binfo ()
  Fetch a node’s build information.
    node - the node whose sources will be collected cache - alternate node to use for the signature cache returns - the
    build signature
    This no longer handles the recursive descent of the node’s children’s signatures. We expect that they’re already
    built and updated by someone else, if that’s what’s wanted.
get_build_env ()
  Fetch the appropriate Environment to build this node.
get_build_scanner_path (scanner)
  Fetch the appropriate scanner path for this node.
get_builder (default_builder=None)
  Return the set builder, or a specified default value
get_cachedir_csig ()
get_contents ()
  Fetch the contents of the entry.
get_csig ()
get_dir ()
get_env ()
get_env_scanner (env, kw={})
get_executor (create: int = 1) → Executor
  Fetch the action executor for this node. Create one if there isn’t already one, and requested to do so.
get_found_includes (env, scanner, path)
Return the scanned include lines (implicit dependencies) found in this node.
The default is no implicit dependencies. We expect this method to be overridden by any subclass that can be
scanned for implicit dependencies.

get_implicit_deps (env, initial_scanner, path_func, kw={})
Return a list of implicit dependencies for this node.
This method exists to handle recursive invocation of the scanner on the implicit dependencies returned by the
scanner, if the scanner’s recursive flag says that we should.

get_internal_path ()
get_labspath ()
Get the absolute path of the file.
get_ninfo ()

get_path (dir=None)
Return path relative to the current working directory of the Node.FS.Base object that owns us.
get_path_elements ()
get_relpath ()
Get the path of the file relative to the root SConstruct file’s directory.

get_source_scanner (node)
Fetch the source scanner for the specified node
NOTE: “self” is the target being built, “node” is the source file for which we want to fetch the scanner.
Implies self.has_builder() is true; again, expect to only be called from locations where this is already verified.
This function may be called very often; it attempts to cache the scanner found to improve performance.

get_state ()
get_stored_implicit ()
Fetch the stored implicit dependencies
get_stored_info ()

get_string (for_signature)
This is a convenience function designed primarily to be used in command generators (i.e.,
CommandGeneratorActions or Environment variables that are callable), which are called with a for_signature
argument that is nonzero if the command generator is being called to generate a signature for the command line,
which determines if we should rebuild or not.
Such command generators should use this method in preference to str(Node) when converting a Node to a string,
passing in the for_signature parameter, such that we will call Node.for_signature() or str(Node) properly,
depending on whether we are calculating a signature or actually constructing a command line.

get_subst_proxy ()
This method is expected to return an object that will function exactly like this Node, except that it implements any
additional special features that we would like to be in effect for Environment variable substitution. The principle use
is that some Nodes would like to implement a __getattr__() method, but putting that in the Node type itself has a
tendency to kill performance. We instead put it in a proxy and return it from this method. It is legal for this method to
return self if no new functionality is needed for Environment substitution.

get_suffix ()
get_target_scanner ()
get_tpath ()
gmtime ()
getsize ()

has_builder () → bool
Return whether this Node has a builder or not.
In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly (“if
node.builder: …”). When the builder attribute is examined directly, it ends up calling __getattr__ for both the
__len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and
slowing things down immensely.

has_explicit_builder () → bool
Return whether this Node has an explicit builder.
This allows an internal Builder created by SCons to be marked non-explicit, so that it can be overridden by an
explicit builder that the user supplies (the canonical example being directories).

ignore
ignore_set
implicit
implicit_set
includes
is_conftest () \rightarrow bool
    Returns true if this node is an conftest node
is_derived () \rightarrow bool
    Returns true if this node is derived (i.e. built).
This should return true only for nodes whose path should be in the variant directory when duplicate=0 and should contribute their build signatures when they are used as source files to other derived files. For example: source with source builders are not derived in this sense, and hence should not return true.
is_explicit
is_literal () \rightarrow bool
    Always pass the string representation of a Node to the command interpreter literally.
is_sconscript () \rightarrow bool
    Returns true if this node is an sconscript
is_under (dir) \rightarrow bool
is_up_to_date () \rightarrow bool
    Default check for whether the Node is current: unknown Node subtypes are always out of date, so they will always get built.
isdir () \rightarrow bool
isfile () \rightarrow bool
islink () \rightarrow bool
linked
lstat ()
make_ready () \rightarrow None
    Get a Node ready for evaluation.
This is called before the Taskmaster decides if the Node is up-to-date or not. Overriding this method allows for a Node subclass to be disambiguated if necessary, or for an implicit source builder to be attached.
missing () \rightarrow bool
multiple_side_effect_has_builder () \rightarrow bool
    Return whether this Node has a builder or not.
In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly ("if node.builder: …"). When the builder attribute is examined directly, it ends up calling __getattr__ for both the __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.
must_be_same (klass)
    This node, which already existed, is being looked up as the specified klass. Raise an exception if it isn’t.
name
new_binfo ()
new_ninfo ()
ninfo
nocache
noclean
postprocess () \rightarrow None
    Clean up anything we don’t need to hang onto after we’ve been built.
precious
prepare ()
    Prepare for this Node to be built.
This is called after the Taskmaster has decided that the Node is out-of-date and must be rebuilt, but before actually calling the method to build the Node.
This default implementation checks that explicit or implicit dependencies either exist or are derived, and initializes the BuildInfo structure that will hold the information about how this node is, uh, built.
(The existence of source files is checked separately by the Executor, which aggregates checks for all of the targets built by a specific action.)
Overriding this method allows for a Node subclass to remove the underlying file from the file system. Note that subclass methods should call this base class method to get the child check and the BuildInfo structure.

prerequisites
pseudo
push_to_cache () → None
Try to push a node into a cache
ref_count
release_target_info () → None
Called just after this node has been marked up-to-date or was built completely.
This is where we try to release as many target node infos as possible for clean builds and update runs, in order to minimize the overall memory consumption.
By purging attributes that aren’t needed any longer after a Node (=File) got built, we don’t have to care that much how many KBytes a Node actually requires...as long as we free the memory shortly afterwards.
@see: built() and File.release_target_info()
remove ()
Remove this Node: no-op by default.
render_include_tree ()
Return a text representation, suitable for displaying to the user, of the include tree for the sources of this node.
retry ()
reset_executor () → None
Remove cached executor; forces recompute when needed.
retrieve_from_cache () → bool
Try to retrieve the node’s content from a cache
This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().
Returns true if the node was successfully retrieved.
rexists ()
Does this node exist locally or in a repository?
rfile ()
rstr () → str
A Node.FS.Base object’s string representation is its path name.
sbuilder
scan () → None
Scan this node’s dependents for implicit dependencies.
scanner_key ()
select_scanner (scanner)
Selects a scanner for this Node.
This is a separate method so it can be overridden by Node subclasses (specifically, Node.FS.Dir) that must use their own Scanner and don’t select one the Scanner.Selector that’s configured for the target.
set_always_build (always_build: int = 1) → None
Set the Node’s always_build value.
set_executor (executor: Executor) → None
Set the action executor for this node.
set_explicit (is_explicit) → None
set_local () → None
set_nocache (nocache: int = 1) → None
Set the Node’s nocache value.
set_noclean (noclean: int = 1) → None
Set the Node’s noclean value.
set_precious (precious: int = 1) → None
Set the Node’s precious value.
set_pseudo (pseudo: bool = True) → None
Set the Node’s pseudo value.
set_specific_source (source) → None
set_src_builder (builder) → None
Set the source code builder for this node.
set_state (state) \rightarrow \text{None}
side_effect
side_effects
sources
sources_set
src_builder ()
    Fetch the source code builder for this node.
    If there isn't one, we cache the source code builder specified for the directory (which in turn will cache the value from its parent directory, and so on up to the file system root).
srcnode ()
    If this node is in a build path, return the node corresponding to its source file. Otherwise, return ourself.
stat ()
state
store_info
str_for_display ()
target_from_source (prefix, suffix, splitext=<function splitext>)
    Generates a target entry that corresponds to this entry (usually a source file) with the specified prefix and suffix.
    Note that this method can be overridden dynamically for generated files that need different behavior. See Tool/swig.py for an example.
target_peers
visited () \rightarrow \text{None}
    Called just after this node has been visited (with or without a build).
waiting_parents
waiting_s_e
wkids
class SCons.Node.FS.Dir (name, directory, fs)
Bases: Base
A class for directories in a file system.
class Attrs
    Bases: object
    shared
BuildInfo
    alias of DirBuildInfo
Decider (function) \rightarrow \text{None}
Dir (name, create: bool = True)
    Looks up or creates a directory node named "name" relative to this directory.
Enter (name)
    Looks up or creates an entry node named "name" relative to this directory.
File (name)
    Looks up or creates a file node named "name" relative to this directory.
GetTag (key)
    Return a user-defined tag.
NodeInfo
    alias of DirNodeInfo
RDirs (pathlist)
    Search for a list of directories in the Repository list.
Rfindalldirs (pathlist)
    Return all of the directories for a given path list, including corresponding “backing” directories in any repositories. The Node lookups are relative to this Node (typically a directory), so memoizing result saves cycles from looking up the same path for each target in a given directory.
Tag (key, value) \rightarrow \text{None}
    Add a user-defined tag.
_Rfindalldirs_key (pathlist)
__clearRepositoryCache (duplicate=None) \rightarrow \text{None}
    Called when we change the repository(ies) for a directory. This clears any cached information that is invalidated by changing the repository.
__getattr__(attr)
Together with the node_bwcomp dict defined below, this method provides a simple backward compatibility layer for the Node attributes 'abspath', 'labspath', 'path', 'tpath', 'suffix' and 'path_elements'. These Node attributes used to be directly available in v2.3 and earlier, but have been replaced by getter methods that initialize the single variables lazily when required, in order to save memory. The redirection to the getters lets older Tools and SConstruct continue to work without any additional changes, fully transparent to the user. Note, that __getattr__ is only called as fallback when the requested attribute can’t be found, so there should be no speed performance penalty involved for standard builds.

__lt__(other)
less than operator used by sorting on py3

__resetDuplicate(node) → None

__str__() → str
A Node.FS.Base object’s string representation is its path name.

_abspath

_add_child(collection, set, child) → None
Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.

_children_get()

_children_reset() → None

_create() → None
Create this directory, silently and without worrying about whether the builder is the default or not.

_func_exists

_func_get_contents

_func_is_derived

_func_rexists

_func_sconsign

_func_target_from_source

_get_scanner(env, initial_scanner, root_node_scanner, kw)

_get_str() → None

_glob1(pattern, ondisk: bool = True, source: bool = False, strings: bool = False)
Globs for and returns a list of entry names matching a single pattern in this directory.
This searches any repositories and source directories for corresponding entries and returns a Node (or string) relative to the current directory if an entry is found anywhere.
TODO: handle pattern with no wildcard. Python’s glob.glob uses a separate _glob0 function to do this.

_labspath

_local

_memo

_morph() → None
Turn a file system Node (either a freshly initialized directory object or a separate Entry object) into a proper directory object.
Set up this directory’s entries and hook it into the file system tree. Specify that directories (this Node) don’t use signatures for calculating whether they’re current.

_path

_path_elements

_proxy

_rel_path_key(other)

_save_str() → None

_sconsign

_specific_sources

_srcdir_find_file_key(filename)

_tags

_tpath

addRepository(dir) → None

add_dependency(depend)
Adds dependencies.

add_ignore(depend)
Adds dependencies to ignore.
add_prerequisite (prerequisite) → None
    Adds prerequisites
add_source (source)
    Adds sources.
add_to_implicit (deps) → None
add_to_waiting_parents (node) → int
    Returns the number of nodes added to our waiting parents list: 1 if we add a unique waiting parent, 0 if not. (Note
    that the returned values are intended to be used to increment a reference count, so don’t think you can “clean up”
    this function by using True and False instead…)
add_to_waiting_s_e (node) → None
add_wkid (wkid) → None
    Add a node to the list of kids waiting to be evaluated
all_children (scan: int = 1)
    Return a list of all the node’s direct children.
alter_targets ()
    Return any corresponding targets in a variant directory.
always_build
attributes
binfo
    A null “builder” for directories.
builder
builder_set (builder) → None
built () → None
    Called just after this node is successfully built.
cached
cachedir_csig
changesig
changed (node=None, allowcache: bool = False)
    Returns if the node is up-to-date with respect to the BuildInfo stored last time it was built. The default behavior is to
    compare it against our own previously stored BuildInfo, but the stored BuildInfo from another Node (typically one in
    a Repository) can be used instead.
    Note that we now always check every dependency. We used to short-circuit the check by returning as soon as we
    detected any difference, but we now rely on checking every dependency to make sure that any necessary Node
    information (for example, the content signature of an #included .h file) is updated.
    The allowcache option was added for supporting the early release of the executor/builder structures, right after a
    File target was built. When set to true, the return value of this changed method gets cached for File nodes. Like
    this, the executor isn’t needed any longer for subsequent calls to changed().
    @see: FS.File.changed(), FS.File.release_target_info()
changed_since_last_build
check_attributes (name)
    Simple API to check if the node.attributes for name has been set
children (scan: int = 1)
    Return a list of the node’s direct children, minus those that are ignored by this node.
children_are_up_to_date () → bool
    Alternate check for whether the Node is current: If all of our children were up-to-date, then this Node was
    up-to-date, too.
clear () → None
    Completely clear a Node of all its cached state (so that it can be re-evaluated by interfaces that do continuous
    integration builds).
clear_memoized_values () → None
contentsig
cwd
del_binfo () → None
    Delete the build info from this node.
depends
depends_set
dir
dir_on_disk(name)
dirname
disambiguate(must_exist=None)
diskcheck_match() → None
do_duplicate(src) → None
duplicate
dirname
entry_abspath(name)
entry_exists_on_disk(name)
    Searches through the file/dir entries of the current directory, and returns True if a physical entry with the given
    name could be found.
    @see reentry_exists_on_disk
entry_labspath(name)
entry_path(name)
entry_tpath(name)
eval
env
env_set(env, safe: bool = False) → None
executor
executor_cleanup() → None
    Let the executor clean up any cached information.
exists()
explain()
file_on_disk(name)
for_signature()
    Return a string representation of the Node that will always be the same for this particular Node, no matter what.
    This is by contrast to the __str__() method, which might, for instance, return a relative path for a file Node. The
    purpose of this method is to generate a value to be used in signature calculation for the command line used to
    build a target, and we use this method instead of str() to avoid unnecessary rebuilds. This method does not need to
    return something that would actually work in a command line; it can return any kind of nonsense, so long as it does
    not change.
fs
    Reference to parent Node.FS object
getRepositories()
    Returns a list of repositories for this directory.
get_abspath() → str
    Get the absolute path of the file.
get_all_rdirs()
get_binfo()
    Fetch a node’s build information.
    node - the node whose sources will be collected cache - alternate node to use for the signature cache returns - the
    build signature
    This no longer handles the recursive descent of the node’s children’s signatures. We expect that they’re already
    built and updated by someone else, if that’s what’s wanted.
get_build_env()
    Fetch the appropriate Environment to build this node.
get_build_scanner_path(scanner)
    Fetch the appropriate scanner path for this node.
get_builder(default_builder=None)
    Return the set builder, or a specified default value
get_cachedir_csig()
get_contents()
    Return content signatures and names of all our children separated by new-lines. Ensure that the nodes are sorted.
get_csig()
Compute the content signature for Directory nodes. In general, this is not needed and the content signature is not
stored in the DirNodeInfo. However, if get_contents on a Dir node is called which has a child directory, the child
directory should return the hash of its contents.
get_dir()
get_env()
get_env_scanner(env, kw={})
get_executor(create: int = 1) → Executor
Fetch the action executor for this node. Create one if there isn’t already one, and requested to do so.
get_found_includes(env, scanner, path)
Return this directory’s implicit dependencies.
We don’t bother caching the results because the scan typically shouldn’t be requested more than once (as
opposed to scanning .h file contents, which can be requested as many times as the files is #included by other
files).
get_implicit_deps(env, initial_scanner, path_func, kw={})
Return a list of implicit dependencies for this node.
This method exists to handle recursive invocation of the scanner on the implicit dependencies returned by the
scanner, if the scanner’s recursive flag says that we should.
get_internal_path()
get_labspath() → str
Get the absolute path of the file.
get_ninfo()
get_path(dir=None)
Get the path relative to the current working directory of the Node.FS.Base object that owns us.
get_path_elements()
get_relpath()
Get the path of the file relative to the root SConstruct file’s directory.
get_source_scanner(node)
Fetch the source scanner for the specified node
NOTE: “self” is the target being built, “node” is the source file for which we want to fetch the scanner.
Implies self.has_builder() is true; again, expect to only be called from locations where this is already verified.
This function may be called very often; it attempts to cache the scanner found to improve performance.
get_state()
get_stored_implicit()
Fetch the stored implicit dependencies
get_stored_info()
get_string(for_signature)
This is a convenience function designed primarily to be used in command generators (i.e.,
CommandGeneratorActions or Environment variables that are callable), which are called with a for_signature
argument that is nonzero if the command generator is being called to generate a signature for the command line,
which determines if we should rebuild or not.
Such command generators should use this method in preference to str(Node) when converting a Node to a string,
passing in the for_signature parameter, such that we will call Node.for_signature() or str(Node) properly,
depending on whether we are calculating a signature or actually constructing a command line.
get_substr_proxy()
This method is expected to return an object that will function exactly like this Node, except that it implements any
additional special features that we would like to be in effect for Environment variable substitution. The principle use
is that some Nodes would like to implement a __getattr__() method, but putting that in the Node type itself has a
tendency to kill performance. We instead put it in a proxy and return it from this method. It is legal for this method to
return self if no new functionality is needed for Environment substitution.
get_suffix()
get_target_scanner()
get_text_contents()
We already emit things in text, so just return the binary version.
get_timestamp() → int
Return the latest timestamp from among our children
get_path ()
getmtime ()
getsize ()
glob (pathname, ondisk: bool = True, source: bool = False, strings: bool = False, exclude=None)→ list

Returns a list of Nodes (or strings) matching a pathname pattern.
Pathname patterns follow POSIX shell syntax:

* matches everything
? matches any single character
[seq] matches any character in seq (ranges allowed)
[!seq] matches any char not in seq

The wildcard characters can be escaped by enclosing in brackets. A leading dot is not matched by a wildcard, and needs to be explicitly included in the pattern to be matched. Matches also do not span directory separators.
The matches take into account Repositories, returning a local Node if a corresponding entry exists in a Repository (either an in-memory Node or something on disk).
The underlying algorithm is adapted from a rather old version of glob.glob() function in the Python standard library (heavily modified), and uses fnmatch.fnmatch() under the covers.
This is the internal implementation of the external Glob API.

Parameters:
- **pattern** – pathname pattern to match.
- **ondisk** – if false, restricts matches to in-memory Nodes. By default, matches entries that exist on-disk in addition to in-memory Nodes.
- **source** – if true, corresponding source Nodes are returned if globbing in a variant directory. The default behavior is to return Nodes local to the variant directory.
- **strings** – if true, returns the matches as strings instead of Nodes. The strings are path names relative to this directory.
- **exclude** – if not None, must be a pattern or a list of patterns following the same POSIX shell semantics. Elements matching at least one pattern from exclude will be excluded from the result.

has_builder () → bool
Return whether this Node has a builder or not.
In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly ("if node.builder: ..."). When the builder attribute is examined directly, it ends up calling __getattr__ for both the __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.

has_explicit_builder () → bool
Return whether this Node has an explicit builder.
This allows an internal Builder created by SCons to be marked non-explicit, so that it can be overridden by an explicit builder that the user supplies (the canonical example being directories).

ignore
ignore_set
implicit
implicit_set
includes
is_conftest () → bool
Returns true if this node is an conftest node

is-derived () → bool
Returns true if this node is derived (i.e. built).
This should return true only for nodes whose path should be in the variant directory when duplicate=0 and should contribute their build signatures when they are used as source files to other derived files. For example: source with source builders are not derived in this sense, and hence should not return true.
is_explicit
is_literal () -> bool
  Always pass the string representation of a Node to the command interpreter literally.
is_sconscript () -> bool
  Returns true if this node is an scons script
is_under (dir) -> bool
is_up_to_date () -> bool
  If any child is not up-to-date, then this directory isn’t, either.
isdir () -> bool
isfile () -> bool
islink () -> bool
  Set this directory as the variant directory for the supplied source directory.
link (srmdir, duplicate) -> None
  make_ready () -> None
  Get a Node ready for evaluation.
  This is called before the Taskmaster decides if the Node is up-to-date or not. Overriding this method allows for a
  Node subclass to be disambiguated if necessary, or for an implicit source builder to be attached.
multiple_side_effect_has_builder ()
  Return whether this Node has a builder or not.
  In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly (“if
  node.builder: …”). When the builder attribute is examined directly, it ends up calling __getattr__ for both the
  __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and
  slowing things down immensely.
must_be_same (klass)
  This node, which already existed, is being looked up as the specified klass. Raise an exception if it isn’t.
nname
new_binfo ()
new_ninfo ()
ninfo
nocache
noclean
  on_disk_entries
postprocess () -> None
  Clean up anything we don’t need to hang onto after we’ve been built.
precious
prepare () -> None
  Prepare for this Node to be built.
  This is called after the Taskmaster has decided that the Node is out-of-date and must be rebuilt, but before actually
  calling the method to build the Node.
  This default implementation checks that explicit or implicit dependencies either exist or are derived, and initializes
  the BuildInfo structure that will hold the information about how this node is, uh, built.
  (The existence of source files is checked separately by the Executor, which aggregates checks for all of the targets
  built by a specific action.)
  Overriding this method allows for a Node subclass to remove the underlying file from the file system. Note that
  subclass methods should call this base class method to get the child check and the BuildInfo structure.
prerequisites
pseudo
push_to_cache () -> None
  Try to push a node into a cache
rdir ()
ref_count
rel_path (other)
  Return a path to “other” relative to this directory.
release_target_info () \rightarrow \text{None}

Called just after this node has been marked up-to-date or was built completely.

This is where we try to release as many target node infos as possible for clean builds and update runs, in order to minimize the overall memory consumption.

By purging attributes that aren’t needed any longer after a Node (=File) got built, we don’t have to care that much how many KBytes a Node actually requires…as long as we free the memory shortly afterwards.

@see: built() and File.release_target_info()

released_target_info

remove ()

Remove this Node: no-op by default.

render_include_tree ()

Return a text representation, suitable for displaying to the user, of the include tree for the sources of this node.

dentry ()

dentry_exists_on_disk (\text{name})

Searches through the file/dir entries of the current and all its remote directories (repos), and returns True if a physical entry with the given name could be found. The local directory (self) gets searched first, so repositories take a lower precedence regarding the searching order.

@see entry_exists_on_disk

drepositories

dreset_executor () \rightarrow \text{None}

Remove cached executor; forces recompute when needed.

dretrieve_from_cache () \rightarrow \text{bool}

Try to retrieve the node’s content from a cache

This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().

Returns true if the node was successfully retrieved.

drexists ()

Does this node exist locally or in a repository?

dfile ()

droot

drstr () \rightarrow \text{str}

A Node.FS.Base object’s string representation is its path name.

dsbuilder
dscan () \rightarrow \text{None}

Scan this node’s dependents for implicit dependencies.
dscanner_key ()

A directory does not get scanned.
dscanner_paths
dscosign ()

Return the .sconsign file info for this directory.
dsearched
dselect_scanner (\text{scanner})

Selects a scanner for this Node.

This is a separate method so it can be overridden by Node subclasses (specifically, Node.FS.Dir) that must use their own Scanner and don’t select one the Scanner.Selector that’s configured for the target.

dset_always_build (always_build: \text{int} = 1) \rightarrow \text{None}

Set the Node’s always_build value.

dset_executor (executor: \text{Executor}) \rightarrow \text{None}

Set the action executor for this node.

dset_explicit (is_explicit) \rightarrow \text{None}

dset_local () \rightarrow \text{None}

dset_nocache (nocache: \text{int} = 1) \rightarrow \text{None}

Set the Node’s nocache value.

dset_noclean (noclean: \text{int} = 1) \rightarrow \text{None}

Set the Node’s noclean value.

dset_precious (precious: \text{int} = 1) \rightarrow \text{None}
Set the Node’s precious value.

```python
set_pseudo(pseudo: bool = True) → None
```

Set the Node’s pseudo value.

```python
set_specific_source(source) → None
```

Set the Node’s specific source.

```python
set_src_builder(builder) → None
```

Set the source code builder for this node.

```python
set_state(state) → None
```

Set the Node’s state.

```python
side_effect
side_effects
sources
sources_set
src_builder() → None
```

Fetch the source code builder for this node.

If there isn’t one, we cache the source code builder specified for the directory (which in turn will cache the value from its parent directory, and so on up to the file system root).

```python
srcdir
srcdir_duplicate(name)
srcdir_find_file(filename)
srcdir_list() → None
srcnode()
```

Dir has a special need for srcnode()...if we have a srcdir attribute set, then that is our srcnode.

```python
stat() → None
state
store_info
str_for_display() → None
target_from_source(prefix, suffix, splitext=<function splitext>)
```

Generates a target entry that corresponds to this entry (usually a source file) with the specified prefix and suffix. Note that this method can be overridden dynamically for generated files that need different behavior. See Tool/swig.py for an example.

```python
target_peers
up() → None
variant_dirs
visited() → None
```

Called just after this node has been visited (with or without a build).

```python
waiting_parents
waiting_s_e
walk(func, arg) → None
```

Walk this directory tree by calling the specified function for each directory in the tree. This behaves like the os.path.walk() function, but for in-memory Node.FS.Dir objects. The function takes the same arguments as the functions passed to os.path.walk():

```python
    func(arg, dirname, fnames)
```

Except that “dirname” will actually be the directory Node, not the string. The ‘.’ and ‘..’ entries are excluded from fnames. The fnames list may be modified in-place to filter the subdirectories visited or otherwise impose a specific order. The “arg” argument is always passed to func() and may be used in any way (or ignored, passing None is common).

```python
wkids
class SCons.Node.FS.DirBuildInfo
    Bases: BuildInfoBase
    __getstate__() → None
```

Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a ‘__dict__’ slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all instances of a class.

```python
__setstate__(state) → None
```

Restore the attributes from a pickled state.

```python
bact
bactsig
```
depends
dependsigs
implicit
implicitsigs
sources
sourcesigs
current_version_id = 2
merge (other) → None
Merge the fields of another object into this object. Already existing information is overwritten by the other instance’s data. WARNING: If a ‘__dict__’ slot is added, it should be updated instead of replaced.

```
class SCons.Node.FS.DirNodeInfo
    Bases: NodeInfoBase
    ...
    __getstate__() → None
    Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a ‘__dict__’ slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all instances of a class.
    __setstate__(state) → None
    Restore the attributes from a pickled state. The version is discarded.
    convert(node, val) → None
    current_version_id = 2
    format(field_list=None, names: int = 0)
    fs = None
    merge (other) → None
    Merge the fields of another object into this object. Already existing information is overwritten by the other instance’s data. WARNING: If a ‘__dict__’ slot is added, it should be updated instead of replaced.
    str_to_node(s)
```
update (node) → None

class SCons.Node.FS.DiskChecker (disk_check_type, do_check_function, ignore_check_function)
    Bases: object
    Implement disk check variation.
    This Class will hold functions to determine what this particular disk checking implementation should do when enabled or disabled.
    enable (disk_check_type_list) → None
    If the current object’s disk_check_type matches any in the list passed :param disk_check_type_list: List of disk checks to enable :return:

class SCons.Node.FS.Entry (name, directory, fs)
    Bases: Base
    This is the class for generic Node.FS entries–that is, things that could be a File or a Dir, but we’re just not sure yet. Consequently, the methods in this class really exist just to transform their associated object into the right class when the time comes, and then call the same-named method in the transformed class.

class Attrs
    Bases: object
    shared
BuildInfo
    alias of BuildInfoBase
Decider (function) → None
GetTag (key)
    Return a user-defined tag.
NodeInfo
    alias of NodeInfoBase
RDirs (pathlist)
    Search for a list of directories in the Repository list.
RFindalldirs (pathlist)
    Return all of the directories for a given path list, including corresponding “backing” directories in any repositories. The Node lookups are relative to this Node (typically a directory), so memoizing result saves cycles from looking up the same path for each target in a given directory.
Tag (key, value) → None
Add a user-defined tag.

_Rfindalldirs_key (pathlist)
_getattr__ (attr)
Together with the node_bwcomp dict defined below, this method provides a simple backward compatibility layer for
the Node attributes ‘abspath’, ‘labspath’, ‘path’, ‘tpath’, ‘suffix’ and ‘path_elements’. These Node attributes used to
be directly available in v2.3 and earlier, but have been replaced by getter methods that initialize the single
variables lazily when required, in order to save memory. The redirection to the getters lets older Tools and
SConstruct continue to work without any additional changes, fully transparent to the user. Note, that __getattr__ is
only called as fallback when the requested attribute can’t be found, so there should be no speed performance
penalty involved for standard builds.

_lt__ (other)
less than operator used by sorting on py3
__str__ () → str
A Node.FS.Base object’s string representation is its path name.

_abspath
_add_child (collection, set, child) → None
Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.

_children_get ()
_children_reset () → None
_func_exists
_func_get_contents
_func_is_derived
_func_rexists
_func_sconsign
_func_target_from_source
_get_scanner (env, initial_scanner, root_node_scanner, kw)
_get_str ()
_glob1 (pattern, ondisk: bool = True, source: bool = False, strings: bool = False)

_labspath
_local
_memo
_path
_path_elements
_proxy
_save_str ()
_sconsign
_specific_sources
_tags
_tpath
add_dependency (depend)
Adds dependencies.
add_ignore (depend)
Adds dependencies to ignore.
add_prerequisite (prerequisite) → None
Adds prerequisites
add_source (source)
Adds sources.
add_to_implicit (deps) → None
add_to_waiting_parents (node) → int
Returns the number of nodes added to our waiting parents list: 1 if we add a unique waiting parent, 0 if not. (Note
that the returned values are intended to be used to increment a reference count, so don’t think you can “clean up”
this function by using True and False instead…)
add_to_waiting_s_e (node) → None
add_wkid (wkid) → None
Add a node to the list of kids waiting to be evaluated
SCons API Documentation

all_children (scan: int = 1)
    Return a list of all the node's direct children.
alter_targets ()
    Return a list of alternate targets for this Node.
always_build
attributes
binfo
build (**kw)
    Actually build the node.
    This is called by the Taskmaster after it's decided that the Node is out-of-date and must be rebuilt, and after the
    prepare() method has gotten everything, uh, prepared.
    This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe
    stuff in built().
builder
builder_set (builder) → None
built () → None
    Called just after this node is successfully built.
cached
cachedir_csig
cachesig
changed (node=None, allowcache: bool = False)
    Returns if the node is up-to-date with respect to the BuildInfo stored last time it was built. The default behavior is to
    compare it against our own previously stored BuildInfo, but the stored BuildInfo from another Node (typically one in
    a Repository) can be used instead.
    Note that we now always check every dependency. We used to short-circuit the check by returning as soon as we
    detected any difference, but we now rely on checking every dependency to make sure that any necessary Node
    information (for example, the content signature of an #included .h file) is updated.
    The allowcache option was added for supporting the early release of the executor/builder structures, right after a
    File target was built. When set to true, the return value of this changed method gets cached for File nodes. Like
    this, the executor isn't needed any longer for subsequent calls to changed().
    @see: FS.File.changed(), FS.File.release_target_info()
changed_since_last_build
check_attributes (name)
    Simple API to check if the node.attributes for name has been set
children (scan: int = 1)
    Return a list of the node's direct children, minus those that are ignored by this node.
children_are_up_to_date () → bool
    Alternate check for whether the Node is current: If all of our children were up-to-date, then this Node was
    up-to-date, too.
clear () → None
    Completely clear a Node of all its cached state (so that it can be re-evaluated by interfaces that do continuous
    integration builds).
clear_memoized_values () → None
contentsig
cwd
del_binfo () → None
    Delete the build info from this node.
depends
depends_set
dir
dirname
disambiguate (must_exist=None)
diskcheck_match () → None
duplicate
duplicate
entries
env

env_set (env, safe: bool = False) → None

executor

executor_cleanup () → None

Let the executor clean up any cached information.

exists ()

Reports whether node exists.

explain ()

for_signature ()

Return a string representation of the Node that will always be the same for this particular Node, no matter what. This is by contrast to the __str__() method, which might, for instance, return a relative path for a file Node. The purpose of this method is to generate a value to be used in signature calculation for the command line used to build a target, and we use this method instead of str() to avoid unnecessary rebuilds. This method does not need to return something that would actually work in a command line; it can return any kind of nonsense, so long as it does not change.

fs

Reference to parent Node.FS object

get_abspath ()

Get the absolute path of the file.

get_binfo ()

Fetch a node’s build information.
	node - the node whose sources will be collected cache - alternate node to use for the signature cache returns - the build signature

This no longer handles the recursive descent of the node’s children’s signatures. We expect that they’re already built and updated by someone else, if that’s what’s wanted.

get_build_env ()

Fetch the appropriate Environment to build this node.

get_build_scanner_path (scanner)

Fetch the appropriate scanner path for this node.

get_builder (default_builder=None)

Return the set builder, or a specified default value

get_cachedir_csig ()

get_contents ()

Fetch the contents of the entry. Returns the exact binary contents of the file.

get_csig ()

get_dir ()

get_env ()

get_env_scanner (env, kw={})

get_executor (create: int = 1) → Executor

Fetch the action executor for this node. Create one if there isn’t already one, and requested to do so.

get_found_includes (env, scanner, path)

Return the scanned include lines (implicit dependencies) found in this node.

The default is no implicit dependencies. We expect this method to be overridden by any subclass that can be scanned for implicit dependencies.

get Implicit_deps (env, initial_scanner, path_func, kw={})

Return a list of implicit dependencies for this node.

This method exists to handle recursive invocation of the scanner on the implicit dependencies returned by the scanner, if the scanner’s recursive flag says that we should.

get_internal_path ()

get_labspath ()

Get the absolute path of the file.

get_ninfo ()

get_path (dir=None)

Return path relative to the current working directory of the Node.FS.Base object that owns us.

get_path_elements ()

get_repath ()
Get the path of the file relative to the root SConstruct file’s directory.

`get_source_scanner(node)`
Fetch the source scanner for the specified node

NOTE: “self” is the target being built, “node” is the source file for which we want to fetch the scanner.
Implies self.has_builder() is true; again, expect to only be called from locations where this is already verified.
This function may be called very often; it attempts to cache the scanner found to improve performance.

`get_state()`
Fetch the stored implicit dependencies

`get_stored_info()`

`get_stored_implicit()`

`get_string(for_signature)`
This is a convenience function designed primarily to be used in command generators (i.e., CommandGeneratorActions or Environment variables that are callable), which are called with a for_signature argument that is nonzero if the command generator is being called to generate a signature for the command line, which determines if we should rebuild or not.
Such command generators should use this method in preference to str(Node) when converting a Node to a string, passing in the for_signature parameter, such that we will call Node.for_signature() or str(Node) properly, depending on whether we are calculating a signature or actually constructing a command line.

`get_subst_proxy()`
This method is expected to return an object that will function exactly like this Node, except that it implements any additional special features that we would like to be in effect for Environment variable substitution. The principle use is that some Nodes would like to implement a __getattr__() method, but putting that in the Node type itself has a tendency to kill performance. We instead put it in a proxy and return it from this method. It is legal for this method to return self if no new functionality is needed for Environment substitution.

`get_suffix()`

`get_target_scanner()`

`get_text_contents() → str`
Fetch the decoded text contents of a Unicode encoded Entry.
Since this should return the text contents from the file system, we check to see into what sort of subclass we should morph this Entry.

`get_tpath()`
`getmtime()`
`getsize()`

`has_builder() → bool`
Return whether this Node has a builder or not.
In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly (“if node.builder: …”). When the builder attribute is examined directly, it ends up calling __getattr__ for both the __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.

`has_explicit_builder() → bool`
Return whether this Node has an explicit builder.
This allows an internal Builder created by SCons to be marked non-explicit, so that it can be overridden by an explicit builder that the user supplies (the canonical example being directories).

`ignore`
`ignore_set`
`implicit`
`implicit_set`
`includes`

`is_conftest() → bool`
Returns true if this node is an conftest node

`is_derived() → bool`
Returns true if this node is derived (i.e. built).
This should return true only for nodes whose path should be in the variant directory when duplicate=0 and should contribute their build signatures when they are used as source files to other derived files. For example: source with source builders are not derived in this sense, and hence should not return true.

`is_explicit`
is_literal () → bool
  Always pass the string representation of a Node to the command interpreter literally.

is_sconscript () → bool
  Returns true if this node is an sconscript

is_under (dir) → bool
  is_up_to_date () → bool
  Default check for whether the Node is current: unknown Node subtypes are always out of date, so they will always
  get built.

isdire () → bool
isfile () → bool
islink () → bool
linked

lstat ()

make_ready () → None
  Get a Node ready for evaluation.
  This is called before the Taskmaster decides if the Node is up-to-date or not. Overriding this method allows for a
  Node subclass to be disambiguated if necessary, or for an implicit source builder to be attached.

missing () → bool
multiple_side_effect_has_builder () → bool
  Return whether this Node has a builder or not.
  In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly ("if
  node.builder: …"). When the builder attribute is examined directly, it ends up calling __getattr__ for both the
  __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and
  slowing things down immensely.

must_be_same (klass) → None
  Called to make sure a Node is a Dir. Since we’re an Entry, we can morph into one.

name
new_binfo ()
new_ninfo ()
ninfo
nocache
noclean
on_disk_entries
postprocess () → None
  Clean up anything we don’t need to hang onto after we’ve been built.

precious
prepare ()
  Prepare for this Node to be built.
  This is called after the Taskmaster has decided that the Node is out-of-date and must be rebuilt, but before actually
  calling the method to build the Node.
  This default implementation checks that explicit or implicit dependencies either exist or are derived, and initializes
  the BuildInfo structure that will hold the information about how this node is, uh, built.
  (The existence of source files is checked separately by the Executor, which aggregates checks for all of the targets
  built by a specific action.)
  Overriding this method allows for for a Node subclass to remove the underlying file from the file system. Note that
  subclass methods should call this base class method to get the child check and the BuildInfo structure.

prerequisites
pseudo
push_to_cache () → None
  Try to push a node into a cache

ref_count
rel_path (other)
release_target_info () → None
  Called just after this node has been marked up-to-date or was built completely.
  This is where we try to release as many target node infos as possible for clean builds and update runs, in order to
  minimize the overall memory consumption.
By purging attributes that aren’t needed any longer after a Node (=File) got built, we don’t have to care that much how many KBytes a Node actually requires...as long as we free the memory shortly afterwards.

@see: built() and File.release_target_info()
released_target_info
remove()
\> Remove this Node: no-op by default.
render_include_tree()
\> Return a text representation, suitable for displaying to the user, of the include tree for the sources of this node.
entry()
repositories
reset_executor() -> None
\> Remove cached executor; forces recompute when needed.
retrieve_from_cache() -> bool
\> Try to retrieve the node's content from a cache
This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().
\> Returns true if the node was successfully retrieved.
exists()
\> Does this node exist locally or in a repository?
file()
\> We’re a generic Entry, but the caller is actually looking for a File at this point, so morph into one.
root
rstr() -> str
\> A Node.FS.Base object’s string representation is its path name.
sbuilder
scan() -> None
\> Scan this node’s dependents for implicit dependencies.
scanner_key()
scanner_paths
searched
select_scanner(scanner)
\> Selects a scanner for this Node.
This is a separate method so it can be overridden by Node subclasses (specifically, Node.FS.Dir) that must use their own Scanner and don’t select one the Scanner.Selector that’s configured for the target.
set_always_build(always_build: int = 1) -> None
\> Set the Node’s always_build value.
set_executor(executor: Executor) -> None
\> Set the action executor for this node.
set_explicit(is_explicit) -> None
set_local() -> None
set_nocache(nocache: int = 1) -> None
\> Set the Node’s nocache value.
set_noclean(noclean: int = 1) -> None
\> Set the Node’s noclean value.
set_precious(precious: int = 1) -> None
\> Set the Node’s precious value.
set_pseudo(pseudo: bool = True) -> None
\> Set the Node’s pseudo value.
set_specific_source(source) -> None
set_src_builder(builder) -> None
\> Set the source code builder for this node.
set_state(state) -> None
side_effect
side_effects
sources
sources_set
src_builder ()
    Fetch the source code builder for this node.
    If there isn’t one, we cache the source code builder specified for the directory (which in turn will cache the value
    from its parent directory, and so on up to the file system root).
srddir
srcnode ()
    If this node is in a build path, return the node corresponding to its source file. Otherwise, return ourself.
stat ()
state
store_info
str_for_display ()
target_from_source (prefix, suffix, splitext=<function splitext>)
    Generates a target entry that corresponds to this entry (usually a source file) with the specified prefix and suffix.
    Note that this method can be overridden dynamically for generated files that need different behavior. See
    Tool/swig.py for an example.
target_peers
variant_dirs
visited () → None
    Called just after this node has been visited (with or without a build).
waiting_parents
waiting_s_e
wkids
class SCons.Node.FS.EntryProxy (subject)
    Bases: Proxy
    __get_abspath ()
    __get_base_path ()
    Return the file’s directory and file name, with the suffix stripped.
    __get_dir ()
    __get_file ()
    __get_filebase ()
    __get_posix_path ()
    Return the path with / as the path separator, regardless of platform.
    __get_reopath ()
    __get_rsrdir ()
    Returns the directory containing the source node linked to this node via VariantDir(), or the directory of this node if
    not linked.
    __get_rsrncde ()
    __get_srdir ()
    Returns the directory containing the source node linked to this node via VariantDir(), or the directory of this node if
    not linked.
    __get_srsnode ()
    __get_suffix ()
    __get_windows_path ()
    Return the path with as the path separator, regardless of platform.
dictSpecialAttrs = {'abspath': <function EntryProxy.__get_abspath>, 'base': <function EntryProxy.__get_base_path>, 'dir': <function EntryProxy.__get_dir>, 'file': <function EntryProxy.__get_file>, 'filebase': <function EntryProxy.__get_filebase>, 'posix': <function EntryProxy.__get_posix_path}, 'reopath': <function EntryProxy.__get_reopath>, 'rsrdir': <function EntryProxy.__get_rsrdir}, 'rsrncde': <function EntryProxy.__get_rsrncde>, 'srcdir': <function EntryProxy.__get_srdir}, 'srcnode': <function EntryProxy.__get_srsnode}, 'suffix': <function EntryProxy.__get_suffix}, 'win32': <function EntryProxy.__get_windows_path}, 'windows': <function EntryProxy.__get_windows_path})
get ()
    Retrieve the entire wrapped object
except SCons.Node.FS.EntryProxyAttributeError (entry_proxy, attribute)
    Bases: AttributeError
An AttributeError subclass for recording and displaying the name of the underlying Entry involved in an AttributeError exception.

```
add_note ()
    Exception.add_note(note) – add a note to the exception
```

```
args
name
attribute name
obj
```

```
with_traceback ()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.
```

```
class SCons.Node.FS.FS (path=None)
    Bases: LocalFS
Dir (name, directory=None, create: bool = True)
    Look up or create a Dir node with the specified name. If the name is a relative path (begins with ./,../, or a file name), then it is looked up relative to the supplied directory node, or to the top level directory of the FS (supplied at construction time) if no directory is supplied.
    This method will raise TypeError if a normal file is found at the specified path.
Entry (name, directory=None, create: bool = True)
    Look up or create a generic Entry node with the specified name. If the name is a relative path (begins with ./,../, or a file name), then it is looked up relative to the supplied directory node, or to the top level directory of the FS (supplied at construction time) if no directory is supplied.
File (name, directory=None, create: bool = True)
    Look up or create a File node with the specified name. If the name is a relative path (begins with ./,../, or a file name), then it is looked up relative to the supplied directory node, or to the top level directory of the FS (supplied at construction time) if no directory is supplied.
    This method will raise TypeError if a directory is found at the specified path.
Glob (pathname, ondisk: bool = True, source: bool = True, strings: bool = False, exclude=None, cwd=None)
    This is mainly a shim layer
PyPackageDir (modulename) → Dir | None
    Locate the directory of Python module modulename.
    For example 'SCons' might resolve to Windows: C:Python311Libsite-packagesSCons Linux: /usr/lib64/python3.11/site-packages/SCons
    Can be used to determine a toolpath based on a Python module name.
    This is the backend called by the public API function PyPackageDir().
Repository (*dirs) → None
    Specify Repository directories to search.
VariantDir (variant_dir, src_dir, duplicate: int = 1)
    Link the supplied variant directory to the source directory for purposes of building files.
_lookup (p, directory, fsclass, create: bool = True)
    The generic entry point for Node lookup with user-supplied data.
    This translates arbitrary input into a canonical Node.FS object of the specified fsclass. The general approach for strings is to turn it into a fully normalized absolute path and then call the root directory’s lookup_abs() method for the heavy lifting.
    If the path name begins with ‘#’, it is unconditionally interpreted relative to the top-level directory of this FS. ‘#’ is treated as a synonym for the top-level SConstruct directory, much like ‘~’ is treated as a synonym for the user’s home directory in a UNIX shell. So both ‘#foo’ and ‘/~foo’ refer to the ‘foo’ subdirectory underneath the top-level SConstruct directory.
    If the path name is relative, then the path is looked up relative to the specified directory, or the current directory (self._cwd, typically the SConscript directory) if the specified directory is None.
chdir (dir, change_os_dir: bool = False)
    Change the current working directory for lookups. If change_os_dir is true, we will also change the “real” cwd to match.
chmod (path, mode)
copy (src, dst)
copy2 (src, dst)
events (path)
get_max_drift ()
get_root (drive)
    Returns the root directory for the specified drive, creating it if necessary.
getcwd ()
getmtime (path)
getsize (path)
isdir (path) → bool
isfile (path) → bool
islink (path) → bool
link (src, dst)
listdir (path)
lstat (path)
makedirs (path, mode: int = 511, exist_ok: bool = False)
mkdir (path, mode: int = 511)
open (path)
readlink (file) → str
rename (old, new)
s scandir (path)
set_SConstruct_dir (dir) → None
set_max_drift (max_drift) → None
stat (path)
symlink (src, dst)
unlink (path)

SCons API Documentation

variant_dir_target_climb (orig, dir, tail)
    Create targets in corresponding variant directories
    Climb the directory tree, and look up path names relative to any linked variant directories we find.
    Even though this loops and walks up the tree, we don’t memoize the return value because this is really only used
to process the command-line targets.

class SCons.Node.FS.File (name, directory, fs)

    Bases: Base
    A class for files in a file system.

class Attrs
    Bases: object
    shared

BuildInfo
    alias of FileBuildInfo

Decider (function) → None

Dir (name, create: bool = True)
    Create a directory node named ‘name’ relative to the directory of this file.

DirS (pathlist)
    Create a list of directories relative to the SConscript directory of this file.

Entry (name)
    Create an entry node named ‘name’ relative to the directory of this file.

File (name)
    Create a file node named ‘name’ relative to the directory of this file.

GetTag (key)
    Return a user-defined tag.

NodeInfo
    alias of FileNodeInfo

RDirs (pathlist)
    Search for a list of directories in the Repository list.

Rfindalldirs (pathlist)
    Return all of the directories for a given path list, including corresponding “backing” directories in any repositories.
The Node lookups are relative to this Node (typically a directory), so memoizing result saves cycles from looking up
the same path for each target in a given directory.

Tag (key, value) → None
Add a user-defined tag.

_Rfindalldirs_key (pathlist)
__dmap_cache = {}
__dmap_sig_cache = {}
__getattr__ (attr)
Together with the node_bwcomp dict defined below, this method provides a simple backward compatibility layer for
the Node attributes ‘abspath’, ‘labspath’, ‘path’, ‘tpath’, ‘suffix’ and ‘path_elements’. These Node attributes used to
be directly available in v2.3 and earlier, but have been replaced by getter methods that initialize the single
variables lazily when required, in order to save memory. The redirection to the getters lets older Tools and
SConstruct continue to work without any additional changes, fully transparent to the user. Note, that __getattr__ is
only called as fallback when the requested attribute can’t be found, so there should be no speed performance
penalty involved for standard builds.

_It__ (other)
less than operator used by sorting on py3
__str__ () → str
A Node.FS.Base object’s string representation is its path name.

_abspath
_add_child (collection, set, child) → None
Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.

_add_strings_to_dependency_map (dmap)
In the case comparing node objects isn’t sufficient, we’ll add the strings for the nodes to the dependency map

_build_dependency_map (binfo)
Build mapping from file -> signature

Parameters:

- **self (self)** –
- **considered (binfo - buildinfo from node being)** –

Returns: dictionary of file->signature mappings

_children_get ()
_children_reset () → None
_createDir () → None
_func_exists
_func_get_contents
_func_is_derived
_func_rexists
_func_sconsign
_func_target_from_source
_get_found_includes_key (env, scanner, path)
_get_previous_signatures (dmap)
Return a list of corresponding csigs from previous build in order of the node/files in children.

Parameters:

- **self (self)** –
- **csig (dmap - Dictionary of file ->)** –

Returns: List of csigs for provided list of children

_get_scanner (env, initial_scanner, root_node_scanner, kw)
_get_str ()
_glob1 (pattern, ondisk: bool = True, source: bool = False, strings: bool = False)
_labspath
_local
_memo
_morph () → None
Turn a file system node into a File object.

_add_dependency (depend)
  Adds dependencies.

_add_ignore (depend)
  Adds dependencies to ignore.

_add_prerequisite (prerequisite) → None
  Adds prerequisites

_add_source (source)
  Adds sources.

_add_to_implicit (deps) → None

_add_to_waiting_parents (node) → int
  Returns the number of nodes added to our waiting parents list: 1 if we add a unique waiting parent, 0 if not. (Note that the returned values are intended to be used to increment a reference count, so don’t think you can “clean up” this function by using True and False instead…)

_add_to_waiting_s_e (node) → None

_add_wkid (wkid) → None
  Add a node to the list of kids waiting to be evaluated

_all_children (scan: int = 1)
  Return a list of all the node’s direct children.

_alter_targets ()
  Return any corresponding targets in a variant directory.

always_build

attributes

binfo

build (**kw)
  Actually build the node.
  This is called by the Taskmaster after it’s decided that the Node is out-of-date and must be rebuilt, and after the prepare() method has gotten everything, uh, prepared.
  This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().

builder

builder_set (builder) → None

built () → None
  Called just after this File node is successfully built.
  Just like for ‘release_target_info’ we try to release some more target node attributes in order to minimize the overall memory consumption.
  @see: release_target_info

cached

cachedir_csig

cachesig

changed (node=None, allowcache: bool = False) → bool
  Returns if the node is up-to-date with respect to the BuildInfo stored last time it was built.
  For File nodes this is basically a wrapper around Node.changed(), but we allow the return value to get cached after the reference to the Executor got released in release_target_info().
  @see: Node.changed()

changed_content (target, prev_ni, repo_node=None) → bool

changed_since_last_build
### changed_state

```python
changed_state(target, prev_ni, repo_node=None) → bool
```

Return True if the timestamps don’t match or if there is no previous timestamp.

- **target**: Information about the node from the previous build.
- **prev_ni**: None

### changed_timestamp_match

```python
changed_timestamp_match(target, prev_ni, repo_node=None) → bool
```

NOTE: If the timestamp hasn’t changed this will skip md5’ing the file and just copy the prev_ni provided. If the prev_ni is wrong, it will propagate it. See: https://github.com/SCons/scons/issues/2980

- **target**: Information about the node from the previous build.
- **prev_ni**: None

### changed_timestamp_newer

```python
changed_timestamp_newer(target, prev_ni, repo_node=None) → bool
```

- **target**: Information about the node from the previous build.
- **prev_ni**: None

### changed_timestamp_then_content

```python
changed_timestamp_then_content(target, prev_ni, node=None) → bool
```

- **target**: Information about the node from the previous build.
- **prev_ni**: None
- **node**: Node instance. Check this node for file

### NOTE: If the timestamp hasn’t changed this will skip md5’ing the
### file and just copy the prev_ni provided. If the prev_ni is wrong, it will propagate it. See: https://github.com/SCons/scons/issues/2980

### Parameters:

- **dependency** *(self)*
  - ...
- **target** *(target)*
  - ...
- **.sconsign** *(prev_ni - The NodeInfo object loaded from previous builds)*
  - ...
- **existence/timestamp** *(node - Node instance. Check this node for file)*
  - if specified.

### Returns:

Boolean - Indicates if node(File) has changed.

---

### check_attributes

```python
check_attributes(name)
```

Simple API to check if the node.attributes for name has been set

### children

```python
children(scan: int = 1)
```

Return a list of the node’s direct children, minus those that are ignored by this node.

### children_are_up_to_date

```python
children_are_up_to_date() → bool
```

Alternate check for whether the Node is current: If all of our children were up-to-date, then this Node was up-to-date, too.


### clear

```python
clear() → None
```

Completely clear a Node of all its cached state (so that it can be re-evaluated by interfaces that do continuous integration builds).

### clear.memoized_values

```python
clear.memoized_values() → None
```

### contentsig

```python
convert_copy_attrs = ['bsources', 'bimplicit', 'bdepends', 'bact', 'bactsig', 'ninfo']
```

### convert_old_entry

```python
convert_old_entry(old_entry)
```

### convert_sig_attrs

```python
convert_sig_attrs = ['bsourcesigs', 'bimplicitsigs', 'bdependssigs']
```

### cwd

```python
cwd
```

### del.binfo

```python
del.binfo() → None
```

### depends

```python
depends
```

### depends_set

```python
depends_set
```

### dir

```python
dir
```

### disambiguate

```python
disambiguate(must_exist=None)
```

### diskcheck_match

```python
diskcheck_match() → None
```

### do_duplicate

```python
do_duplicate(src)
```

### duplicate

```python
duplicate
```

### entries

```python
entries
```

### env

```python
env
```

### env_set

```python
env_set(env, safe: bool = False) → None
```

### executor

```python
executor
```

### executor_cleanup

```python
executor_cleanup() → None
```

Let the executor clean up any cached information.

### exists

```python
exists() → None
```

### explain

```python
explain()
```

### find_repo_file

```python
find_repo_file()
```

---
For this node, find if there exists a corresponding file in one or more repositories:

```python
find_src_builder()
for_signature()
```

Return a string representation of the Node that will always be the same for this particular Node, no matter what. This is by contrast to the `__str__()` method, which might, for instance, return a relative path for a file Node. The purpose of this method is to generate a value to be used in signature calculation for the command line used to build a target, and we use this method instead of `str()` to avoid unnecessary rebuilds. This method does not need to return something that would actually work in a command line; it can return any kind of nonsense, so long as it does not change.

```python
defs
Reference to parent Node.FS object
get_abspath()
Get the absolute path of the file.
get_binfo()
Fetch a node’s build information.
node - the node whose sources will be collected
``` cache - alternate node to use for the signature cache
returns - the build signature
This no longer handles the recursive descent of the node’s children’s signatures. We expect that they’re already built and updated by someone else, if that’s what’s wanted.

```python
defget_build_env()
Fetch the appropriate Environment to build this node.
defget_build_scanner_path(scanner)
Fetch the appropriate scanner path for this node.
defget_builder(default_builder=None)
Return the set builder, or a specified default value
defget_cachedir_bsig()
Return the signature for a cached file, including its children.
It adds the path of the cached file to the cache signature, because multiple targets built by the same action will all have the same build signature, and we have to differentiate them somehow.
Signature should normally be string of hex digits.
defget_cachedir_csig()
Fetch a Node’s content signature for purposes of computing another Node’s cachesig.
This is a wrapper around the normal get_csig() method that handles the somewhat obscure case of using CacheDir with the -n option. Any files that don’t exist would normally be “built” by fetching them from the cache, but the normal get_csig() method will try to open up the local file, which doesn’t exist because the -n option meant we didn’t actually pull the file from cachedir. But since the file does actually exist in the cachedir, we can use its contents for the csig.

defget_content_hash() → str
Compute and return the hash for this file.
defget_contents() → bytes
Return the contents of the file as bytes.
defget_contents_sig()
A helper method for get_cachedir_bsig.
It computes and returns the signature for this node’s contents.
defget_csig() → str
Generate a node’s content signature.
gdefget_dir()
gdefget_env()
gdefget_env_scanner(env, kw={})
gdefget_executor(create: int = 1) → Executor
Fetch the action executor for this node. Create one if there isn’t already one, and requested to do so.
defget_found_includes(env, scanner, path)
Return the included implicit dependencies in this file. Cache results so we only scan the file once per path regardless of how many times this information is requested.
defget_implicit_deps(env, initial_scanner, path_func, kw={})
Return a list of implicit dependencies for this node.
This method exists to handle recursive invocation of the scanner on the implicit dependencies returned by the scanner, if the scanner’s recursive flag says that we should.

get_internal_path ()
get_labspath ()
Get the absolute path of the file.
get_max_drift_csig () → str | None
Returns the content signature currently stored for this node if it’s been unmodified longer than the max_drift value, or the max_drift value is 0. Returns None otherwise.

get_ninfo ()
get_path (dir=None)
Return path relative to the current working directory of the Node.FS.Base object that owns us.
get_path_elements ()
get_relpah ()
Get the path of the file relative to the root SConstruct file’s directory.
get_size () → int
get_source_scanner (node)
Fetch the source scanner for the specified node
NOTE: “self” is the target being built, “node” is the source file for which we want to fetch the scanner.
Implies self.has_builder() is true; again, expect to only be called from locations where this is already verified.
This function may be called very often; it attempts to cache the scanner found to improve performance.
get_state ()
get_stored_implicit ()
Fetch the stored implicit dependencies
get_stored_info ()
get_string (for_signature)
This is a convenience function designed primarily to be used in command generators (i.e., CommandGeneratorActions or Environment variables that are callable), which are called with a for_signature argument that is nonzero if the command generator is being called to generate a signature for the command line, which determines if we should rebuild or not.
Such command generators should use this method in preference to str(Node) when converting a Node to a string, passing in the for_signature parameter, such that we will call Node.for_signature() or str(Node) properly, depending on whether we are calculating a signature or actually constructing a command line.
get_subst_proxy ()
This method is expected to return an object that will function exactly like this Node, except that it implements any additional special features that we would like to be in effect for Environment variable substitution. The principle use is that some Nodes would like to implement a __getattr__() method, but putting that in the Node type itself has a tendency to kill performance. We instead put it in a proxy and return it from this method. It is legal for this method to return self if no new functionality is needed for Environment substitution.
get_suffix ()
get_target_scanner ()
get_text_contents () → str
Return the contents of the file as text.
get_timestamp () → int
get_tpath ()
getmtime ()
getsze ()
has_builder () → bool
Return whether this Node has a builder or not.
In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly (“if node.builder: …”). When the builder attribute is examined directly, it ends up calling __getattr__ for both the __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.
has_explicit_builder () → bool
Return whether this Node has an explicit builder.
This allows an internal Builder created by SCons to be marked non-explicit, so that it can be overridden by an explicit builder that the user supplies (the canonical example being directories).

has_src_builder () → bool
Return whether this Node has a source builder or not.
If this Node doesn’t have an explicit source code builder, this is where we figure out, on the fly, if there’s a transparent source code builder for it.
Note that if we found a source builder, we also set the self.builder attribute, so that all of the methods that actually build this file don’t have to do anything different.

hash_chunksize = 65536
ignore
ignore_set
implicit
implicit_set
includes
is_conftest () → bool
Returns true if this node is an conftest node
is_derived () → bool
Returns true if this node is derived (i.e. built).
This should return true only for nodes whose path should be in the variant directory when duplicate=0 and should contribute their build signatures when they are used as source files to other derived files. For example: source with source builders are not derived in this sense, and hence should not return true.

is_explicit
is_literal () → bool
Always pass the string representation of a Node to the command interpreter literally.

is_sconscript () → bool
Returns true if this node is an sconscript

is_under (dir) → bool
is_up_to_date () → bool
Check for whether the Node is current.
In all cases self is the target we’re checking to see if it’s up to date

isdir () → bool
isfile () → bool
islink () → bool
linked
lstat ()

make_ready () → None
Get a Node ready for evaluation.
This is called before the Taskmaster decides if the Node is up-to-date or not. Overriding this method allows for a Node subclass to be disambiguated if necessary, or for an implicit source builder to be attached.
missing () → bool
multiple_side_effect_has_builder () → bool
Return whether this Node has a builder or not.
In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly (“if node.builder: ...”). When the builder attribute is examined directly, it ends up calling __getattr__ for both the __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.

must_be_same (klass)
This node, which already existed, is being looked up as the specified klass. Raise an exception if it isn’t.

name
new_binfo ()
new_ninfo ()
ninfo
nocache
noclean
on_disk_entries
postprocess () → None
Clean up anything we don’t need to hang onto after we’ve been built.

precious
prepare ()
   Prepare for this file to be created.
prerequisites
pseudo
push_to_cache () → None
   Try to push the node into a cache
ref_count
rel_path (other)
release_target_info () → None
   Called just after this node has been marked up-to-date or was built completely.
   This is where we try to release as many target node infos as possible for clean builds and update runs, in order to
   minimize the overall memory consumption.
   We’d like to remove a lot more attributes like self.sources and self.sources_set, but they might get used in a next
   build step. For example, during configuration the source files for a built E(*)o file are used to figure out which linker
   to use for the resulting Program (gcc vs. g++)! That’s why we check for the ‘keep_targetinfo’ attribute, config Nodes
   and the Interactive mode just don’t allow an early release of most variables.
   In the same manner, we can’t simply remove the self.attributes here. The smart linking relies on the shared flag,
   and some parts of the java Tool use it to transport information about nodes...
   @see: built() and Node.release_target_info()
released_target_info
remove ()
   Remove this file.
render_include_tree ()
   Return a text representation, suitable for displaying to the user, of the include tree for the sources of this node.
rentry ()
repositories
reset_executor () → None
   Remove cached executor; forces recompute when needed.
retrieve_from_cache () → bool
   Try to retrieve the node’s content from a cache
   This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe
   stuff in built().
   Returns True if the node was successfully retrieved.
exists ()
   Does this node exist locally or in a repository?
rfile ()
root
rstr ()
   A Node.FS.Base object’s string representation is its path name.
sbuilder
scan () → None
   Scan this node’s dependents for implicit dependencies.
scanner_key ()
scanner_paths
searched
select_scanner (scanner)
   Selects a scanner for this Node.
   This is a separate method so it can be overridden by Node subclasses (specifically, Node.FS.Dir) that must use
   their own Scanner and don’t select one the Scanner.Selector that’s configured for the target.
set_always_build (always_build: int = 1) → None
   Set the Node’s always_build value.
set_executor (executor: Executor) → None
   Set the action executor for this node.
set_explicit (is_explicit) → None
set_local () \rightarrow None

set_nocache (nocache: int = 1) \rightarrow None
    Set the Node’s nocache value.

set_noclean (noclean: int = 1) \rightarrow None
    Set the Node’s noclean value.

set_precious (precious: int = 1) \rightarrow None
    Set the Node’s precious value.

set_pseudo (pseudo: bool = True) \rightarrow None
    Set the Node’s pseudo value.

set_specific_source (source) \rightarrow None
    Set the source code builder for this node.

set_state (state) \rightarrow None

side_effect
side_effects
sources
sources_set
src_builder ()
    Fetch the source code builder for this node.
    If there isn’t one, we cache the source code builder specified for the directory (which in turn will cache the value
    from its parent directory, and so on up to the file system root).

srcdir
srcnode ()
    If this node is in a build path, return the node corresponding to its source file. Otherwise, return ourself.

stat ()
state
store_info
str_for_display ()
target_from_source (prefix, suffix, splitext=<function splitext>)
    Generates a target entry that corresponds to this entry (usually a source file) with the specified prefix and suffix.
    Note that this method can be overridden dynamically for generated files that need different behavior. See
    Tool/swig.py for an example.

target_peers
variant_dirs
visited () \rightarrow None
    Called just after this node has been visited (with or without a build).

waiting_parents
waiting_s_e
wkids

class SCons.Node.FS.FileBuildInfo
Bases: BuildInfoBase
This is info loaded from sconsign.

Attributes unique to FileBuildInfo:

dependency_map : Caches file->csig mapping
    for all dependencies. Currently this is only used when using MD5-timestamp decider. It’s used to ensure that
    we copy the correct csig from the previous build to be written to .sconsign when current build is done.
    Previously the matching of csig to file was strictly by order they appeared in bdepends, bsources, or
    bimplicit, and so a change in order or count of any of these could yield writing wrong csig, and then false
    positive rebuilds
__getstate__() ()
    Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a
    ‘__dict__’ slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all
    instances of a class.
__setstate__(state) \rightarrow None
    Restore the attributes from a pickled state.
bact
bactsig
bdepends
bdependsigs
bimplicit
bimplicitigs
bsources
bsourcesigs
convert_from_sconsign (dir, name) → None
   Converts a newly-read FileBuildInfo object for in-SCons use
   For normal up-to-date checking, we don’t have any conversion to perform— but we’re leaving this method here to
   make that clear.
convert_to_sconsign () → None
   Converts this FileBuildInfo object for writing to a .sconsign file
   This replaces each Node in our various dependency lists with its usual string representation: relative to the
top-level SConstruct directory, or an absolute path if it’s outside.
current_version_id = 2
dependency_map
format (names: int = 0)
merge (other) → None
   Merge the fields of another object into this object. Already existing information is overwritten by the other instance’s
data. WARNING: If a ‘.dict’ slot is added, it should be updated instead of replaced.
prepare_dependencies () → None
   Prepares a FileBuildInfo object for explaining what changed
   The bsources, bdepends and bimplicit lists have all been stored on disk as paths relative to the top-level
SConstruct directory. Convert the strings to actual Nodes (for use by the --debug=explain code and
--implicit-cache).
exception SCons.Node.FS.FileBuildInfoFileToCsigMappingError
   Bases: Exception
   add_note ()
      Exception.add_note(note) – add a note to the exception
   args
   with_traceback ()
      Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.
class SCons.Node.FS.FileFinder
   Bases: object
   _find_file_key (filename, paths, verbose=None)
   filedir_lookup (p, fd=None)
      A helper method for find_file() that looks up a directory for a file we’re trying to find. This only creates the Dir Node
if it exists on-disk, since if the directory doesn’t exist we know we won’t find any files in it… :-)
   It would be more compact to just use this as a nested function with a default keyword argument (see the
commented-out version below), but that doesn’t work unless you have nested scopes, so we define it here just so
this work under Python 1.5.2.
   find_file (filename, paths, verbose=None)
      Find a node corresponding to either a derived file or a file that exists already.
      Only the first file found is returned, and none is returned if no file is found.
      filename: A filename to find paths: A list of directory path nodes to search in. Can be represented as a list, a tuple,
or a callable that is called with no arguments and returns the list or tuple.
      returns The node created from the found file.
class SCons.Node.FS.FileNodeInfo
   Bases: NodeInfoBase
   __getstate__ ()
      Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a
‘.dict’ slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all
instances of a class.
   __setstate__ (state) → None
SCons API Documentation

- Restore the attributes from a pickled state.
- Convert (node, val) → None
- csig
- Current_version_id = 2
- Field_list = ['csig', 'timestamp', 'size']
- Format (field_list=None, names: int = 0)
- Fs = None
- Merge (other) → None
  - Merge the fields of another object into this object. Already existing information is overwritten by the other instance’s data. WARNING: If a '__dict__' slot is added, it should be updated instead of replaced.
- Size
- Str_to_node (s)
- Timestamp
- Update (node) → None

SCons.Node.FS.LinkFunc (target, source, env) → int
  - Relative paths cause problems with symbolic links, so we use absolute paths, which may be a problem for people who want to move their soft-linked src-trees around. Those people should use the 'hard-copy' mode, softlinks cannot be used for that; at least I have no idea how ...

Class: SCons.Node.FS.LocalFS
  - Bases: object
  - This class implements an abstraction layer for operations involving a local file system. Essentially, this wraps any function in the os, os.path or shutil modules that we use to actually go do anything with or to the local file system.
  - Note that there’s a very good chance we’ll refactor this part of the architecture in some way as we really implement the interface(s) for remote file system Nodes. For example, the right architecture might be to have this be a subclass instead of a base class. Nevertheless, we’re using this as a first step in that direction.
  - We’re not using chdir() yet because the calling subclass method needs to use os.chdir() directly to avoid recursion. Will we really need this one?

SCons.Node.FS.LocalString (target, source, env) → str

SCons.Node.FS.MkdirFunc (target, source, env) → int

Class: SCons.Node.FS.RootDir (drive, fs)
  - Bases: Dir
  - A class for the root directory of a file system.
  - This is the same as a Dir class, except that the path separator ('/' or ' ') is actually part of the name, so we don’t need to add a separator when creating the path names of entries within this directory.

Class: Attrs
Bases: object
shared
BuildInfo
  alias of DirBuildInfo
Decider (function) → None
Dir (name, create: bool = True)
  Looks up or creates a directory node named ‘name’ relative to this directory.
Entry (name)
  Looks up or creates an entry node named ‘name’ relative to this directory.
File (name)
  Looks up or creates a file node named ‘name’ relative to this directory.
GetTag (key)
  Return a user-defined tag.
NodeInfo
  alias of DirNodeInfo
RDirs (pathlist)
  Search for a list of directories in the Repository list.
_Rfindalldirs (pathlist)
  Return all of the directories for a given path list, including corresponding “backing” directories in any repositories.
  The Node lookups are relative to this Node (typically a directory), so memoizing result saves cycles from looking up the same path for each target in a given directory.
Tag (key, value) → None
  Add a user-defined tag.
__getattr__ (attr)
  Together with the node_bwcomp dict defined below, this method provides a simple backward compatibility layer for the Node attributes ‘abspath’, ‘labspath’, ‘path’, ‘tpath’, ‘suffix’ and ‘path_elements’. These Node attributes used to be directly available in v2.3 and earlier, but have been replaced by getter methods that initialize the single variables lazily when required, in order to save memory. The redirection to the getters lets older Tools and SConstruct continue to work without any additional changes, fully transparent to the user. Note, that __getattr__ is only called as fallback when the requested attribute can’t be found, so there should be no speed performance penalty involved for standard builds.
__lt__ (other)
  less than operator used by sorting on py3
_abspath
_add_child (collection, set, child) → None
  Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.
_children_get ()
_children_reset () → None
_create ()
  Create this directory, silently and without worrying about whether the builder is the default or not.
_func_exists
_func_get_contents
_func_is-derived
_func_rexists
_func_sconsign
_func_target_from-source
_get_scanner (env, initial_scanner, root_node_scanner, kw)
_get_str ()
_glob1 (pattern, ondisk: bool = True, source: bool = False, strings: bool = False)
  Globs for and returns a list of entry names matching a single pattern in this directory.
  This searches any repositories and source directories for corresponding entries and returns a Node (or string) relative to the current directory if an entry is found anywhere.
  TODO: handle pattern with no wildcard. Python’s glob.glob uses a separate __glob0 function to do this.
_labspath
_local
_lookupDict

_lookup_abs (p, klass, create: bool = True)
Fast (?) lookup of a normalized absolute path.
This method is intended for use by internal lookups with already-normalized path data. For general-purpose
lookups, use the FS.Entry(), FS.Dir() or FS.File() methods.
The caller is responsible for making sure we’re passed a normalized absolute path; we merely let Python’s
dictionary look up and return the One True Node.FS object for the path.
If a Node for the specified “p” doesn’t already exist, and “create” is specified, the Node may be created after
recursive invocation to find or create the parent directory or directories.

_memo

_morph () → None
Turn a file system Node (either a freshly initialized directory object or a separate Entry object) into a proper
directory object.
Set up this directory’s entries and hook it into the file system tree. Specify that directories (this Node) don’t use
signatures for calculating whether they’re current.

_path

_path_elements

_proxy

_rel_path_key (other)

_save_str ()

_sconsign

_specific_sources

_srcdir_find_file_key (filename)

tags

_tpath

.abspath

addRepository (dir) → None
add_dependency (depend)
  Adds dependencies.
add_ignore (depend)
  Adds dependencies to ignore.
add_prerequisite (prerequisite) → None
  Adds prerequisites
add_source (source)
  Adds sources.
add_to_implicit (deps) → None
add_to_waiting_parents (node) → int
  Returns the number of nodes added to our waiting parents list: 1 if we add a unique waiting parent, 0 if not. (Note
  that the returned values are intended to be used to increment a reference count, so don’t think you can “clean up”
  this function by using True and False instead…)
add_to_waiting_s_e (node) → None
add_wkid (wkid) → None
  Add a node to the list of kids waiting to be evaluated
all_children (scan: int = 1)
  Return a list of all the node’s direct children.
alter_targets ()
  Return any corresponding targets in a variant directory.
always_build
attributes
binfo
build (**kw) → None
  A null “builder” for directories.
builder
builder_set (builder) → None
built () → None
  Called just after this node is successfully built.
cached
cachedir_csigin
cachesig

changed (node=None, allowcache: bool = False)
Returns if the node is up-to-date with respect to the BuildInfo stored last time it was built. The default behavior is to compare it against our own previously stored BuildInfo, but the stored BuildInfo from another Node (typically one in a Repository) can be used instead.

Note that we now always check every dependency. We used to short-circuit the check by returning as soon as we detected any difference, but we now rely on checking every dependency to make sure that any necessary Node information (for example, the content signature of an #included .h file) is updated.

The allowcache option was added for supporting the early release of the executor/builder structures, right after a File target was built. When set to true, the return value of this changed method gets cached for File nodes. Like this, the executor isn’t needed any longer for subsequent calls to changed().

@see: FS.File.changed(), FS.File.release_target_info()

changed_since_last_build
check_attributes (name)
  Simple API to check if the node.attributes for name has been set

called_nodes (scan: int = 1)
  Return a list of the node’s direct children, minus those that are ignored by this node.

children_are_up_to_date () → bool
  Alternate check for whether the Node is current: If all of our children were up-to-date, then this Node was up-to-date, too.


clear () → None
  Completely clear a Node of all its cached state (so that it can be re-evaluated by interfaces that do continuous integration builds).

clear_memoized_values () → None

clear

cwd
  Delete the build info from this node.

depend
  depends_set
dir

dir_on_disk (name)
dirname

disambiguate (must_exist=None)
diskcheck_match () → None
do_duplicate (src) → None
duplicate

entries

entry_abspath (name)
entry_exists_on_disk (name)
  Searches through the file/dir entries of the current directory, and returns True if a physical entry with the given name could be found.
  @see reny.entry_exists_on_disk

entry_labspath (name)
entry_path (name)
entry_tpath (name)

env
  env_set (env, safe: bool = False) → None

evaluator
  executor_cleanup () → None
  Let the executor clean up any cached information.

exists ()
  Reports whether node exists.
explain ()
file_on_disk (name)
for_signature ()
    Return a string representation of the Node that will always be the same for this particular Node, no matter what. This is by contrast to the __str__() method, which might, for instance, return a relative path for a file Node. The purpose of this method is to generate a value to be used in signature calculation for the command line used to build a target, and we use this method instead of str() to avoid unnecessary rebuilds. This method does not need to return something that would actually work in a command line; it can return any kind of nonsense, so long as it does not change.

fs
    Reference to parent Node.FS object
getRepositories ()
    Returns a list of repositories for this directory.
get_abspath () → str
    Get the absolute path of the file.
get_all_rdirs ()
get_binfo ()
    Fetch a node’s build information.
    node - the node whose sources will be collected cache - alternate node to use for the signature cache returns - the build signature
    This no longer handles the recursive descent of the node’s children’s signatures. We expect that they’re already built and updated by someone else, if that’s what’s wanted.
get_build_env ()
    Fetch the appropriate Environment to build this node.
get_build_scanner_path (scanner)
    Fetch the appropriate scanner path for this node.
get_builder (default_builder=None)
    Return the set builder, or a specified default value
get_cachedir_csig ()
get_contents ()
    Return content signatures and names of all our children separated by new-lines. Ensure that the nodes are sorted.
get_csig ()
    Compute the content signature for Directory nodes. In general, this is not needed and the content signature is not stored in the DirNodeInfo. However, if get_contents on a Dir node is called which has a child directory, the child directory should return the hash of its contents.
get_dir ()
get_env ()
get_env_scanner (env, kw={})
get_executor (create: int = 1) → Executor
    Fetch the action executor for this node. Create one if there isn’t already one, and requested to do so.
get_found_includes (env, scanner, path)
    Return this directory’s implicit dependencies.
    We don’t bother caching the results because the scan typically shouldn’t be requested more than once (as opposed to scanning .h file contents, which can be requested as many times as the files is #included by other files).
get_implicit_deps (env, initial_scanner, path_func, kw={})
    Return a list of implicit dependencies for this node.
    This method exists to handle recursive invocation of the scanner on the implicit dependencies returned by the scanner, if the scanner’s recursive flag says that we should.
get_internal_path ()
get_labspath () → str
    Get the absolute path of the file.
get_ninfo ()
get_path (dir=None)
    Return path relative to the current working directory of the Node.FS.Base object that owns us.
get_path_elements ()
get_repath ()
    Get the path of the file relative to the root SConstruct file's directory.

get_source_scanner (node)
    Fetch the source scanner for the specified node
    NOTE: “self” is the target being built, “node” is the source file for which we want to fetch the scanner.
    Implies self.has_builder() is true; again, expect to only be called from locations where this is already verified.
    This function may be called very often; it attempts to cache the scanner found to improve performance.

get_state ()
get_stored_implicit ()
    Fetch the stored implicit dependencies

get_stored_info ()
get_string (for_signature)
    This is a convenience function designed primarily to be used in command generators (i.e.,
    CommandGeneratorActions or Environment variables that are callable), which are called with a for_signature
    argument that is nonzero if the command generator is being called to generate a signature for the command line,
    which determines if we should rebuild or not.
    Such command generators should use this method in preference to str(Node) when converting a Node to a string,
    passing in the for_signature parameter, such that we will call Node.for_signature() or str(Node) properly,
    depending on whether we are calculating a signature or actually constructing a command line.

get_subst_proxy ()
    This method is expected to return an object that will function exactly like this Node, except that it implements any
    additional special features that we would like to be in effect for Environment variable substitution. The principle use
    is that some Nodes would like to implement a __getattr__() method, but putting that in the Node type itself has a
    tendency to kill performance. We instead put it in a proxy and return it from this method. It is legal for this method to
    return self if no new functionality is needed for Environment substitution.

get_suffix ()
get_target_scanner ()
get_text_contents ()
    We already emit things in text, so just return the binary version.

get_timestamp () → int
    Return the latest timestamp from among our children

glob (pathname, ondisk: bool = True, source: bool = False, strings: bool = False, exclude=None) → list
    Returns a list of Nodes (or strings) matching a pathname pattern.
    Pathname patterns follow POSIX shell syntax:

    * matches everything
    ? matches any single character
    [seq] matches any character in seq (ranges allowed)
    [!seq] matches any char not in seq

    The wildcard characters can be escaped by enclosing in brackets. A leading dot is not matched by a wildcard, and
    needs to be explicitly included in the pattern to be matched. Matches also do not span directory separators.
    The matches take into account Repositories, returning a local Node if a corresponding entry exists in a Repository
    (either an in-memory Node or something on disk).
    The underlying algorithm is adapted from a rather old version of glob.glob() function in the Python standard library
    (heavily modified), and uses fnmatch.fnmatch() under the covers.
    This is the internal implementation of the external Glob API.
Parameters:

- **pattern** – pathname pattern to match.
- **ondisk** – if false, restricts matches to in-memory Nodes. By default, matches entries that exist on-disk in addition to in-memory Nodes.
- **source** – if true, corresponding source Nodes are returned if globbing in a variant directory. The default behavior is to return Nodes local to the variant directory.
- **strings** – if true, returns the matches as strings instead of Nodes. The strings are path names relative to this directory.
- **exclude** – if not None, must be a pattern or a list of patterns following the same POSIX shell semantics. Elements matching at least one pattern from exclude will be excluded from the result.

**has_builder()** → bool
Return whether this Node has a builder or not.
In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly (“if node.builder: …”). When the builder attribute is examined directly, it ends up calling __getattr__ for both the __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.

**has_explicit_builder()** → bool
Return whether this Node has an explicit builder.
This allows an internal Builder created by SCons to be marked non-explicit, so that it can be overridden by an explicit builder that the user supplies (the canonical example being directories).

**ignore**
**ignore_set**
**implicit**
**implicit_set**
**includes**

**is_conftest()** → bool
Returns true if this node is an conftest node

**is-derived()** → bool
Returns true if this node is derived (i.e. built).
This should return true only for nodes whose path should be in the variant directory when duplicate=0 and should contribute their build signatures when they are used as source files to other derived files. For example: source with source builders are not derived in this sense, and hence should not return true.

**is_explicit**
**is_literal()** → bool
Always pass the string representation of a Node to the command interpreter literally.

**is_sconscript()** → bool
Returns true if this node is an sconscript

**is_under(dir)** → bool
**is_up_to_date()** → bool
If any child is not up-to-date, then this directory isn’t, either.

**isdir()** → bool
**isfile()** → bool
**islink()** → bool

**link(srcdir, duplicate)** → None
Set this directory as the variant directory for the supplied source directory.

**linked**
**lstat()**

**make_ready()** → None
Get a Node ready for evaluation.
This is called before the Taskmaster decides if the Node is up-to-date or not. Overriding this method allows for a Node subclass to be disambiguated if necessary, or for an implicit source builder to be attached.

**missing()** → bool
**multiple_side_effect_has_builder()**
Return whether this Node has a builder or not.

In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly ("if node.builder: ..."). When the builder attribute is examined directly, it ends up calling _getattr_ for both the _len_ and _bool_ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.

**must_be_same**(klass) \rightarrow None

This node, which already existed, is being looked up as the specified klass. Raise an exception if it isn't.

**name**

**new_binfo** ()

**new_ninfo** ()

**ninfo**

**nocache**

**noclean**

**on_disk_entries**

**path**

**postprocess** () \rightarrow None

Clean up anything we don't need to hang onto after we've been built.

**precious**

**prepare** () \rightarrow None

Prepare for this Node to be built.

This is called after the Taskmaster has decided that the Node is out-of-date and must be rebuilt, but before actually calling the method to build the Node.

This default implementation checks that explicit or implicit dependencies either exist or are derived, and initializes the BuildInfo structure that will hold the information about how this node is, uh, built.

(The existence of source files is checked separately by the Executor, which aggregates checks for all of the targets built by a specific action.)

Overriding this method allows for a Node subclass to remove the underlying file from the file system. Note that subclass methods should call this base class method to get the child check and the BuildInfo structure.

**prerequisites**

**pseudo**

**push_to_cache** () \rightarrow None

Try to push a node into a cache

**rdir** ()

**ref_count**

**rel_path**(other)

Return a path to "other" relative to this directory.

**release_target_info** () \rightarrow None

Called just after this node has been marked up-to-date or was built completely.

This is where we try to release as many target node infos as possible for clean builds and update runs, in order to minimize the overall memory consumption.

By purging attributes that aren't needed any longer after a Node (=File) got built, we don't have to care that much how many KBytes a Node actually requires...as long as we free the memory shortly afterwards.

@see: built() and File.release_target_info()

**released_target_info**

**remove** ()

Remove this Node: no-op by default.

**render_include_tree** ()

Return a text representation, suitable for displaying to the user, of the include tree for the sources of this node.

**rentry** ()

**rentry_exists_on_disk**(name)

Searches through the file/dir entries of the current and all its remote directories (repos), and returns True if a physical entry with the given name could be found. The local directory (self) gets searched first, so repositories take a lower precedence regarding the searching order.

@see entry_exists_on_disk

**repositories**

**reset_executor** () \rightarrow None
Remove cached executor; forces recompute when needed.
retrieve_from_cache () → bool
   Try to retrieve the node's content from a cache
   This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().
   Returns true if the node was successfully retrieved.
exists ()
   Does this node exist locally or in a repository?
rfile ()
root
rstr () → str
   A Node.FS.Base object's string representation is its path name.
sbuilder
scan () → None
   Scan this node's dependents for implicit dependencies.
scanner_key ()
   A directory does not get scanned.
scanner_paths
sconsign ()
   Return the .sconsign file info for this directory.
searched
select_scanner (scanner)
   Selects a scanner for this Node.
   This is a separate method so it can be overridden by Node subclasses (specifically, Node.FS.Dir) that must use their own Scanner and don't select one the Scanner.Selector that's configured for the target.
set_always_build (always_build: int = 1) → None
   Set the Node's always_build value.
set_executor (executor: Executor) → None
   Set the action executor for this node.
set_explicit (is_explicit) → None
set_local () → None
set_nocache (nocache: int = 1) → None
   Set the Node's nocache value.
set_noclean (noclean: int = 1) → None
   Set the Node's noclean value.
set_precious (precious: int = 1) → None
   Set the Node's precious value.
set_pseudo (pseudo: bool = True) → None
   Set the Node's pseudo value.
set_specific_source (source) → None
set_src_builder (builder) → None
   Set the source code builder for this node.
set_state (state) → None
side_effect
side_effects
sources
sources_set
src_builder ()
   Fetch the source code builder for this node.
   If there isn't one, we cache the source code builder specified for the directory (which in turn will cache the value from its parent directory, and so on up to the file system root).
srddir
srcdir_duplicate (name)
srcdir_find_file (filename)
srcdir_list ()
srncnode ()
Dir has a special need for srcnode()...if we have a srcdir attribute set, then that is our srcnode.

stat ()
state
store_info
str_for_display ()
target_from_source (prefix, suffix, splitext=<function splitext>)

Generates a target entry that corresponds to this entry (usually a source file) with the specified prefix and suffix.

Note that this method can be overridden dynamically for generated files that need different behavior. See Tool/swig.py for an example.
target_peers
up ()
variant_dirs
visited () → None
Called just after this node has been visited (with or without a build).
waiting_parents
waiting_s_e
walk (func, arg) → None
Walk this directory tree by calling the specified function for each directory in the tree.
This behaves like the os.path.walk() function, but for in-memory Node.FS.Dir objects. The function takes the same arguments as the functions passed to os.path.walk():

    func(arg, dirname, fnames)

Except that “dirname” will actually be the directory Node, not the string. The ‘.’ and ‘.’ entries are excluded from fnames. The fnames list may be modified in-place to filter the subdirectories visited or otherwise impose a specific order. The “arg” argument is always passed to func() and may be used in any way (or ignored, passing None is common).

wkids
SCons.Node.FS.UnlinkFunc (target, source, env) → int
class SCons.Node.FS._Null
Bases: object
SCons.Node.FS._classEntry
alias of Entry
SCons.Node.FS._copy_func (fs, src, dest) → None
SCons.Node.FS._hardlink_func (fs, src, dst) → None
SCons.Node.FS._my_normcase (x)
SCons.Node.FS._my_splitdrive (p)
SCons.Node.FS._softlink_func (fs, src, dst) → None
SCons.Node.FS.diskcheck_types ()
SCons.Node.FS.do_diskcheck_match (node, predicate, errorfmt)
SCons.Node.FS.find_file (filename, paths, verbose=None)

Find a node corresponding to either a derived file or a file that exists already.
Only the first file found is returned, and none is returned if no file is found.
filename: A filename to find paths: A list of directory path nodes to search in. Can be represented as a list, a tuple, or a callable that is called with no arguments and returns the list or tuple.
returns The node created from the found file.
SCons.Node.FS.get_MkdirBuilder ()
SCons.Node.FS.get_default_fs ()
SCons.Node.FS.has_glob_magic (s) → bool
SCons.Node.FS.ignore_diskcheck_match (node, predicate, errorfmt) → None
SCons.Node.FS.initialize_do_splitdrive () → None
SCons.Node.FS.invalidate_node_memos (targets) → None
Invalidate the memoized values of all Nodes (files or directories) that are associated with the given entries. Has been added to clear the cache of nodes affected by a direct execution of an action (e.g. Delete/Copy/Chmod). Existing Node caches become inconsistent if the action is run through Execute(). The argument targets can be a single Node object or filename, or a sequence of Nodes/filenames.
SCons.Node.FS.needs_normpath_match (string, pos=0, endpos=9223372036854775807)
Matches zero or more characters at the beginning of the string.
SCons API Documentation

SCons.Node.FS.save_strings(val) → None
SCons.Node.FS.sconsign_dir(node)
    Return the .sconsign file info for this directory, creating it first if necessary.
SCons.Node.FS.sconsign_none(node)
SCons.Node.FS.set_diskcheck(enabled_checkers) → None
SCons.Node.FS.set_duplicate(duplicate)

SCons.Node.Python module

Python nodes.

class SCons.Node.Python.Value(value, built_value=None, name=None)
    Bases: Node
    A Node class for values represented by Python expressions.
    Values are typically passed on the command line or generated by a script, but not from a file or some other source.
    Changed in version 4.0: the name parameter was added.

class Attrs
    Bases: object
    shared
BuildInfo
    alias of ValueBuildInfo
Decider(function) → None
GetTag(key)
    Return a user-defined tag.
NodeInfo
    alias of ValueNodeInfo
Tag(key, value) → None
    Add a user-defined tag.
    _add_child(collection, set, child) → None
        Adds 'child' to 'collection', first checking 'set' to see if it's already present.
    _children_get() → None
    _children_reset() → None
    _func_exists
    _func_get_contents
    _func_is_derived
    _func_rexists
    _func_target_from_source
    _get_scanner(env, initial_scanner, root_node_scanner, kw)
    _memo
    _specific_sources
    _tags
add_dependency(depend)
    Adds dependencies.
add_ignore(depend)
    Adds dependencies to ignore.
add_prerequisite(prerequisite) → None
    Adds prerequisites
add_source(source) → None
    Adds sources.
add_to_implicit(deps) → None
add_to_waiting_parents(node) → int
    Returns the number of nodes added to our waiting parents list: 1 if we add a unique waiting parent, 0 if not. (Note that the returned values are intended to be used to increment a reference count, so don’t think you can “clean up” this function by using True and False instead…)
add_to_waiting_s_e(node) → None
add_wkid(wkid) → None
    Add a node to the list of kids waiting to be evaluated
all_children(scan: int = 1)
Return a list of all the node’s direct children.

```
alter_targets ()
```

Return a list of alternate targets for this Node.

```
always_build
```

attributes

```
binfo
build (**kw) → None
```

Actually build the node.

This is called by the Taskmaster after it’s decided that the Node is out-of-date and must be rebuilt, and after the prepare() method has gotten everything, uh, prepared.

This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().

```
builder
builder_set (builder) → None
```

```
built () → None
```

Called just after this node is successfully built.

```
cached
changed (node= None, allowcache: bool = False)
```

Returns if the node is up-to-date with respect to the BuildInfo stored last time it was built. The default behavior is to compare it against our own previously stored BuildInfo, but the stored BuildInfo from another Node (typically one in a Repository) can be used instead.

Note that we now **always** check every dependency. We used to short-circuit the check by returning as soon as we detected any difference, but we now rely on checking every dependency to make sure that any necessary Node information (for example, the content signature of an #included .h file) is updated.

The allowcache option was added for supporting the early release of the executor/builder structures, right after a File target was built. When set to true, the return value of this changed method gets cached for File nodes. Like this, the executor isn’t needed any longer for subsequent calls to changed().

@see: FS.File.changed(), FS.File.release_target_info()

```
changed_since_last_build
check_attributes (name)
```

Simple API to check if the node.attributes for name has been set

```
children (scan: int = 1)
```

Return a list of the node’s direct children, minus those that are ignored by this node.

```
children_are_up_to_date () → bool
```

Alternate check for whether the Node is current: If all of our children were up-to-date, then this Node was up-to-date, too.


```
clear () → None
del_binfo () → None
clear_memoized_values () → None
del_binfo () → None
depends
depends_set
disambiguate (must_exist= None)
env
env_set (env, safe: bool = False) → None
executor
executor_cleanup () → None
```

Let the executor clean up any cached information.

```
exists () → bool
```

Reports whether node exists.

```
explain ()
```

```
for_signature ()
```

53
Return a string representation of the Node that will always be the same for this particular Node, no matter what. This is by contrast to the \_str\_() method, which might, for instance, return a relative path for a file Node. The purpose of this method is to generate a value to be used in signature calculation for the command line used to build a target, and we use this method instead of str() to avoid unnecessary rebuilds. This method does not need to return something that would actually work in a command line; it can return any kind of nonsense, so long as it does not change.

get\_abspath ()

Return an absolute path to the Node. This will return simply str(Node) by default, but for Node types that have a concept of relative path, this might return something different.

get\_binfo ()

Fetch a node's build information.

node - the node whose sources will be collected cache - alternate node to use for the signature cache returns - the build signature

This no longer handles the recursive descent of the node's children's signatures. We expect that they're already built and updated by someone else, if that's what's wanted.

get\_build\_env ()

Fetch the appropriate Environment to build this node.

get\_build\_scanner\_path (scanner)

Fetch the appropriate scanner path for this node.

get\_builder (default\_builder=None)

Return the set builder, or a specified default value

get\_cachedir\_csig ()

Get contents for signature calculations.

get\_csig (calc=None)

Because we're a Python value node and don't have a real timestamp, we get to ignore the calculator and just use the value contents.

Returns string. Ideally string of hex digits. (Not bytes)

get\_env ()

get\_env\_scanner (env, kw={})

get\_executor (create: int = 1) \rightarrow Executor

Fetch the action executor for this node. Create one if there isn't already one, and requested to do so.

get\_found\_includes (env, scanner, path)

Return the scanned include lines (implicit dependencies) found in this node.

The default is no implicit dependencies. We expect this method to be overridden by any subclass that can be scanned for implicit dependencies.

get\_implicit\_deps (env, initial\_scanner, path\_func, kw={})

Return a list of implicit dependencies for this node.

This method exists to handle recursive invocation of the scanner on the implicit dependencies returned by the scanner, if the scanner's recursive flag says that we should.

get\_ninfo ()

get\_source\_scanner (node)

Fetch the source scanner for the specified node

NOTE: “self” is the target being built, “node” is the source file for which we want to fetch the scanner.

Implies self.has\_builder() is true; again, expect to only be called from locations where this is already verified.

This function may be called very often; it attempts to cache the scanner found to improve performance.

get\_state ()

get\_stored\_implicit ()

Fetch the stored implicit dependencies

get\_stored\_info ()

get\_string (for\_signature)

This is a convenience function designed primarily to be used in command generators (i.e., CommandGeneratorActions or Environment variables that are callable), which are called with a for\_signature argument that is nonzero if the command generator is being called to generate a signature for the command line, which determines if we should rebuild or not.
Such command generators should use this method in preference to str(Node) when converting a Node to a string, passing in the for_signature parameter, such that we will call Node.for_signature() or str(Node) properly, depending on whether we are calculating a signature or actually constructing a command line.

get_subst_proxy ()
This method is expected to return an object that will function exactly like this Node, except that it implements any additional special features that we would like to be in effect for Environment variable substitution. The principle use is that some Nodes would like to implement a __getattr__() method, but putting that in the Node type itself has a tendency to kill performance. We instead put it in a proxy and return it from this method. It is legal for this method to return self if no new functionality is needed for Environment substitution.

get_suffix () → str
get_target.Scanner ()
get_text_contents () → str

By the assumption that the node.built_value is a deterministic product of the sources, the contents of a Value are the concatenation of all the contents of its sources. As the value need not be built when get_contents() is called, we cannot use the actual node.built_value.

has_builder () → bool
Return whether this Node has a builder or not.

In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly ("if node.builder: ... "). When the builder attribute is examined directly, it ends up calling __getattr__ for both the __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.

has_explicit_builder () → bool
Return whether this Node has an explicit builder.

This allows an internal Builder created by SCons to be marked non-explicit, so that it can be overridden by an explicit builder that the user supplies (the canonical example being directories).

ignore
ignore_set
implicit
implicit_set
includes
is_conftest () → bool
Returns true if this node is an conftest node

is_derived () → bool
Returns true if this node is derived (i.e. built).

This should return true only for nodes whose path should be in the variant directory when duplicate=0 and should contribute their build signatures when they are used as source files to other derived files. For example: source with source builders are not derived in this sense, and hence should not return true.

is_explicit
is_literal () → bool
Always pass the string representation of a Node to the command interpreter literally.

is_sconscript () → bool
Returns true if this node is an sconscript

is_under (dir) → bool
is_up_to_date () → bool
Alternate check for whether the Node is current: If all of our children were up-to-date, then this Node was up-to-date, too.


linked
make_ready () → None
Get a Node ready for evaluation.

This is called before the Taskmaster decides if the Node is up-to-date or not. Overriding this method allows for a Node subclass to be disambiguated if necessary, or for an implicit source builder to be attached.

missing () → bool
multiple_side_effect_has_builder () → bool
Return whether this Node has a builder or not.
In Boolean tests, this turns out to be a *lot* more efficient than simply examining the builder attribute directly ("if node.builder: ..."). When the builder attribute is examined directly, it ends up calling __getattr__ for both the __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.

```python
new_binfo ()
new_ninfo ()
ninfo
nocache
noclean
postprocess () → None
    Clean up anything we don’t need to hang onto after we’ve been built.
precious
prepare ()
    Prepare for this Node to be built.
    This is called after the Taskmaster has decided that the Node is out-of-date and must be rebuilt, but before actually calling the method to build the Node.
    This default implementation checks that explicit or implicit dependencies either exist or are derived, and initializes the BuildInfo structure that will hold the information about how this node is, uh, built.
    (The existence of source files is checked separately by the Executor, which aggregates checks for all of the targets built by a specific action.)
    Overriding this method allows for for a Node subclass to remove the underlying file from the file system. Note that subclass methods should call this base class method to get the child check and the BuildInfo structure.
prerequisites
pseudo
push_to_cache () → None
    Try to push a node into a cache
read ()
    Return the value. If necessary, the value is built.
ref_count
release_target_info () → None
    Called just after this node has been marked up-to-date or was built completely.
    This is where we try to release as many target node infos as possible for clean builds and update runs, in order to minimize the overall memory consumption.
    By purging attributes that aren’t needed any longer after a Node (=File) got built, we don’t have to care that much how many KBytes a Node actually requires...as long as we free the memory shortly afterwards.
    @see: built() and File.release_target_info()
remove ()
    Remove this Node: no-op by default.
render_include_tree ()
    Return a text representation, suitable for displaying to the user, of the include tree for the sources of this node.
reset_executor () → None
    Remove cached executor; forces recompute when needed.
retrieve_from_cache () → bool
    Try to retrieve the node’s content from a cache
    This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().
    Returns true if the node was successfully retrieved.
rexists ()
    Does this node exist locally or in a repository?
scan () → None
    Scan this node’s dependents for implicit dependencies.
scanner_key ()
select_scanner (scanner)
    Selects a scanner for this Node.
    This is a separate method so it can be overridden by Node subclasses (specifically, Node.FS.Dir) that *must* use their own Scanner and don’t select one the Scanner.Selector that’s configured for the target.
set_always_build(always_build: int = 1) → None
Set the Node’s always_build value.

set_executor(executor: Executor) → None
Set the action executor for this node.

set_explicit(is_explicit) → None
Set the Node’s explicit value.

set_nocache(nocache: int = 1) → None
Set the Node’s nocache value.

set_noclean(noclean: int = 1) → None
Set the Node’s noclean value.

set_precious(precious: int = 1) → None
Set the Node’s precious value.

set_pseudo(pseudo: bool = True) → None
Set the Node’s pseudo value.

set_specific_source(source) → None

set_state(state) → None

side_effect
side_effects

sources

sources_set

state

store_info

str_for_display()

target_peers

visited() → None
Called just after this node has been visited (with or without a build).

waiting_parents

waiting_s_e

wkids

write(built_value) → None
Set the value of the node.

class SCons.Node.Python.ValueBuildInfo

Bases: BuildInfoBase

__getstate__() ()
Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a ‘__dict__’ slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all instances of a class.

__setstate__(state) → None
Restore the attributes from a pickled state.

bact

bactsig

bdepends

bdependssigs

bimplicit

bimplicitsigs

bsources

bsourcesigs

current_version_id = 2

merge(other) → None
Merge the fields of another object into this object. Already existing information is overwritten by the other instance’s data. WARNING: If a ‘__dict__’ slot is added, it should be updated instead of replaced.


Bases: NodeInfoBase

__getstate__() ()
Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a ‘__dict__’ slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all instances of a class.
SCons API Documentation

__setstate__(state) → None
    Restore the attributes from a pickled state.
convert(node, val) → None
csigt
    current_version_id = 2
    field_list = ['csigt']
format(field_list=None, names: int = 0)
merge(other) → None
    Merge the fields of another object into this object. Already existing information is overwritten by the other instance's
data. WARNING: If a '__dict__' slot is added, it should be updated instead of replaced.
str_to_node(s)
update(node) → None

SCons.Node.Python.ValueWithMemo(value, built_value=None, name=None)
    Memoized Value node factory.
    Changed in version 4.0: the name parameter was added.

Module contents

The Node package for the SCons software construction utility.

This is, in many ways, the heart of SCons.

A Node is where we encapsulate all of the dependency information about any thing that SCons can build, or about any thing which SCons can use to build some other thing. The canonical "thing," of course, is a file, but a Node can also represent something remote (like a web page) or something completely abstract (like an Alias).

Each specific type of "thing" is specifically represented by a subclass of the Node base class: Node.FS.File for files, Node.Alias for aliases, etc. Dependency information is kept here in the base class, and information specific to files/aliases/etc. is in the subclass. The goal, if we've done this correctly, is that any type of "thing" should be able to depend on any other type of "thing."

SCons.Node.Annotate(node) → None
class SCons.Node.BuildInfoBase
    Bases: object
    The generic base class for build information for a Node.
    This is what gets stored in a .sconsign file for each target file. It contains a NodelInfo instance for this node (signature information that's specific to the type of Node) and direct attributes for the generic build stuff we have to track: sources, explicit dependencies, implicit dependencies, and action information.
__getstate__() → None
    Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a '__dict__' slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all instances of a class.
__setstate__(state) → None
    Restore the attributes from a pickled state.
bact
bactsig
bdepends
bdependsigs
bimplicit
bimplicitcerts
bsources
bsourcesigs
current_version_id = 2
merge(other) → None
    Merge the fields of another object into this object. Already existing information is overwritten by the other instance's data. WARNING: If a '__dict__' slot is added, it should be updated instead of replaced.
class SCons.Node.Node
    Bases: object
    The base Node class, for entities that we know how to build, or use to build other Nodes.
class Attrs
    Bases: object
    shared
BuildInfo
    alias of BuildInfoBase
Decider (function) → None
GetTag (key)
    Return a user-defined tag.
NodeInfo
    alias of NodeInfoBase
Tag (key, value) → None
    Add a user-defined tag.
    _add_child(collection, set, child) → None
        Adds 'child' to 'collection', first checking 'set' to see if it's already present.
    _children_get ()
    _children_reset () → None
    _func_exists
    _func_get_contents
    _func_is_derived
    _func_rexists
    _func_target_from_source
    _get_scanner (env, initial_scanner, root_node_scanner, kw)
    _memo
    _specific_sources
    _tags
    add_dependency (depend)
        Adds dependencies.
    add_ignore (depend)
        Adds dependencies to ignore.
    add_prerequisite (prerequisite) → None
        Adds prerequisites.
    add_source (source)
        Adds sources.
    add_to_implicit (deps) → None
    add_to_waiting_parents (node) → int
        Returns the number of nodes added to our waiting parents list: 1 if we add a unique waiting parent, 0 if not. (Note that the returned values are intended to be used to increment a reference count, so don't think you can "clean up" this function by using True and False instead…)
    add_to_waiting_s_e (node) → None
    add_wkid (wkid) → None
        Add a node to the list of kids waiting to be evaluated.
    all_children (scan: int = 1)
        Return a list of all the node's direct children.
    alter_targets ()
        Return a list of alternate targets for this Node.
    always_build
    attributes
    binfo
    build (**kw)
        Actually build the node.
        This is called by the Taskmaster after it's decided that the Node is out-of-date and must be rebuilt, and after the prepare() method has gotten everything, uh, prepared.
        This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().
    builder
    builder_set (builder) → None
Built () → None
   Called just after this node is successfully built.

Cached

Changed (node=Node, allowcache: bool = False)
   Returns if the node is up-to-date with respect to the BuildInfo stored last time it was built. The default behavior is to compare it against our own previously stored BuildInfo, but the stored BuildInfo from another Node (typically one in a Repository) can be used instead.
   Note that we now always check every dependency. We used to short-circuit the check by returning as soon as we detected any difference, but we now rely on checking every dependency to make sure that any necessary Node information (for example, the content signature of an #included .h file) is updated.
   The allowcache option was added for supporting the early release of the executor/builder structures, right after a File target was built. When set to true, the return value of this changed method gets cached for File nodes. Like this, the executor isn’t needed any longer for subsequent calls to changed().
   @see: FS.File.changed(), FS.File.release_target_info()

ChangedSinceLastBuild

CheckAttributes (name)
   Simple API to check if the node.attributes for name has been set

Children (scan: int = 1)
   Return a list of the node’s direct children, minus those that are ignored by this node.

ChildrenAreUpToDate () → bool
   Alternate check for whether the Node is current: If all of our children were up-to-date, then this Node was up-to-date, too.

Clear () → None
   Completely clear a Node of all its cached state (so that it can be re-evaluated by interfaces that do continuous integration builds).

ClearMemoizedValues () → None
   Delete the build info from this node.

Depends

DependsSet

Disambiguate (must_exist=None)

Env

EnvSet (env, safe: bool = False) → None

Executor

ExecutorCleanup () → None
   Let the executor clean up any cached information.

Exists () → bool
   Reports whether node exists.

Explanation

ForSignature ()
   Return a string representation of the Node that will always be the same for this particular Node, no matter what. This is by contrast to the __str__() method, which might, for instance, return a relative path for a file Node. The purpose of this method is to generate a value to be used in signature calculation for the command line used to build a target, and we use this method instead of str() to avoid unnecessary rebuilds. This method does not need to return something that would actually work in a command line; it can return any kind of nonsense, so long as it does not change.

GetAbsPath ()
   Return an absolute path to the Node. This will return simply str(Node) by default, but for Node types that have a concept of relative path, this might return something different.

GetBinfo ()
   Fetch a node’s build information.
   node - the node whose sources will be collected cache - alternate node to use for the signature cache returns - the build signature
   This no longer handles the recursive descent of the node’s children’s signatures. We expect that they’re already built and updated by someone else, if that’s what’s wanted.
get_build_env ()
    Fetch the appropriate Environment to build this node.
get_build_scanner_path (scanner)
    Fetch the appropriate scanner path for this node.
get_builder (default_builder=None)
    Return the set builder, or a specified default value
get_cachedir_csig ()
get_contents ()
    Fetch the contents of the entry.
gcsg ()
get_env ()
get_env_scanner (env, kw={})
get_executor (create: int = 1) → Executor
    Fetch the action executor for this node. Create one if there isn’t already one, and requested to do so.
get_found_includes (env, scanner, path)
    Return the scanned include lines (implicit dependencies) found in this node.
    The default is no implicit dependencies. We expect this method to be overridden by any subclass that can be
    scanned for implicit dependencies.
get_implicit_deps (env, initial_scanner, path_func, kw={})
    Return a list of implicit dependencies for this node.
    This method exists to handle recursive invocation of the scanner on the implicit dependencies returned by the
    scanner, if the scanner’s recursive flag says that we should.
get_ninfo ()
get_source_scanner (node)
    Fetch the source scanner for the specified node
    NOTE: “self” is the target being built, “node” is the source file for which we want to fetch the scanner.
    Implies self.has_builder() is true; again, expect to only be called from locations where this is already verified.
    This function may be called very often; it attempts to cache the scanner found to improve performance.
get_state ()
get_stored_implicit ()
    Fetch the stored implicit dependencies
get_stored_info ()
get_string (for_signature)
    This is a convenience function designed primarily to be used in command generators (i.e.,
    CommandGeneratorActions or Environment variables that are callable), which are called with a for_signature
    argument that is nonzero if the command generator is being called to generate a signature for the command line,
    which determines if we should rebuild or not.
    Such command generators should use this method in preference to str(Node) when converting a Node to a string,
    passing in the for_signature parameter, such that we will call Node.for_signature() or str(Node) properly,
    depending on whether we are calculating a signature or actually constructing a command line.
get_subst_proxy ()
    This method is expected to return an object that will function exactly like this Node, except that it implements any
    additional special features that we would like to be in effect for Environment variable substitution. The principle use
    is that some Nodes would like to implement a __getattr__() method, but putting that in the Node type itself has a
    tendency to kill performance. We instead put it in a proxy and return it from this method. It is legal for this method to
    return self if no new functionality is needed for Environment substitution.
get_suffix () → str
get_target_scanner ()
has_builder () → bool
    Return whether this Node has a builder or not.
    In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly ("if
    node.builder: "...”). When the builder attribute is examined directly, it ends up calling __getattr__ for both the
    __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and
    slowing things down immensely.
has_explicit_builder () → bool
    Return whether this Node has an explicit builder.
This allows an internal Builder created by SCons to be marked non-explicit, so that it can be overridden by an explicit builder that the user supplies (the canonical example being directories).

ignore
ignore_set
implicit
implicit_set
includes
is_conftest () → bool
    Returns true if this node is an conftest node
is_derived () → bool
    Returns true if this node is derived (i.e. built).
    This should return true only for nodes whose path should be in the variant directory when duplicate=0 and should contribute their build signatures when they are used as source files to other derived files. For example: source with source builders are not derived in this sense, and hence should not return true.
is_explicit
is_literal () → bool
    Always pass the string representation of a Node to the command interpreter literally.
is_sconscript () → bool
    Returns true if this node is an sconscript
is_up_to_date () → bool
    Default check for whether the Node is current: unknown Node subtypes are always out of date, so they will always get built.
linked
make_ready () → None
    Get a Node ready for evaluation.
    This is called before the Taskmaster decides if the Node is up-to-date or not. Overriding this method allows for a Node subclass to be disambiguated if necessary, or for an implicit source builder to be attached.
missing () → bool
multiple_side_effect_has_builder () → bool
    Return whether this Node has a builder or not.
    In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly ("if node.builder: ..."). When the builder attribute is examined directly, it ends up calling __getattr__ for both the __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.
new_binfo ()
nnew_ninfo ()
ninfo
nocache
noclean
postprocess () → None
    Clean up anything we don’t need to hang onto after we’ve been built.
precious
prepare ()
    Prepare for this Node to be built.
    This is called after the Taskmaster has decided that the Node is out-of-date and must be rebuilt, but before actually calling the method to build the Node.
    This default implementation checks that explicit or implicit dependencies either exist or are derived, and initializes the BuildInfo structure that will hold the information about how this node is, uh, built.
    (The existence of source files is checked separately by the Executor, which aggregates checks for all of the targets built by a specific action.)
    Overriding this method allows for for a Node subclass to remove the underlying file from the file system. Note that subclass methods should call this base class method to get the child check and the BuildInfo structure.
prerequisites
pseudo
push_to_cache () → None
    Try to push a node into a cache
ref_count
release_target_info () -> None
   Called just after this node has been marked up-to-date or was built completely.
   This is where we try to release as many target node infos as possible for clean builds and update runs, in order to
   minimize the overall memory consumption.
   By purging attributes that aren’t needed any longer after a Node (=File) got built, we don’t have to care that much
   how many KBytes a Node actually requires...as long as we free the memory shortly afterwards.
   @see: built() and File.release_target_info()
remove ()
   Remove this Node: no-op by default.
render_include_tree ()
   Return a text representation, suitable for displaying to the user, of the include tree for the sources of this node.
reset_executor () -> None
   Remove cached executor; forces recompute when needed.
retrieve_from_cache () -> bool
   Try to retrieve the node’s content from a cache
   This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe
   stuff in built().
   Returns true if the node was successfully retrieved.
exists ()
   Does this node exist locally or in a repository?
scan () -> None
   Scan this node’s dependents for implicit dependencies.
scanner_key ()
select_scanner (scanner)
   Selects a scanner for this Node.
   This is a separate method so it can be overridden by Node subclasses (specifically, Node.FS.Dir) that must use
   their own Scanner and don’t select one the Scanner.Selector that’s configured for the target.
set_always_build (always_build: int = 1) -> None
   Set the Node’s always_build value.
set_executor (executor: Executor) -> None
   Set the action executor for this node.
set_explicit (is_explicit) -> None
set_nocache (nocache: int = 1) -> None
   Set the Node’s nocache value.
set_noclean (noclean: int = 1) -> None
   Set the Node’s noclean value.
set_precious (precious: int = 1) -> None
   Set the Node’s precious value.
set_pseudo (pseudo: bool = True) -> None
   Set the Node’s pseudo value.
set_specific_source (source) -> None
set_state (state) -> None
side_effect
side_effects
sources
sources_set
state
store_info
target_peers
visited () -> None
   Called just after this node has been visited (with or without a build).
waiting_parents
waiting_s_e
wkids
class SCons.Node.NodeInfoBase
Bases: object
The generic base class for signature information for a Node.
Node subclasses should subclass NodeInfoBase to provide their own logic for dealing with their own Node-specific
signature information.
__getstate__()  
Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a
‘__dict__’ slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all
instances of a class.
__setstate__(state) → None  
Restore the attributes from a pickled state. The version is discarded.
convert (node, val) → None  
current_version_id = 2
format (field_list=None, names: int = 0)
merge (other) → None  
Merge the fields of another object into this object. Already existing information is overwritten by the other instance’s
data. WARNING: If a ‘__dict__’ slot is added, it should be updated instead of replaced.
update (node) → None
class SCons.Node.NodeList (initlist=None)
Bases: UserList
__abc_impl = <abc._abc_data object>
append (item)
  S.append(value) – append value to the end of the sequence
clear () → None -- remove all items from S
copy ()
count (value) → integer -- return number of occurrences of value
extend (other)
  S.extend(iterable) – extend sequence by appending elements from the iterable
index (value[, start[, stop]]) → integer -- return first index of value.
    Raises ValueError if the value is not present.
    Supporting start and stop arguments is optional, but recommended.
insert (i, item)
  S.insert(index, value) – insert value before index
pop ([, index]) → item -- remove and return item at index (default last).
    Raise IndexError if list is empty or index is out of range.
remove (item)
  S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.
reverse ()
  S.reverse() – reverse IN PLACE
sort (*args, **kwd)
class SCons.Node.Walker (node, kids_func=<function get_children>, cycle_func=<function
drop_cycle>, eval_func=<function do_nothing>)
Bases: object
An iterator for walking a Node tree.
This is depth-first, children are visited before the parent. The Walker object can be initialized with any node, and
returns the next node on the descent with each get_next() call. get the children of a node instead of calling ‘children’.
‘cycle_func’ is an optional function that will be called when a cycle is detected.
This class does not get caught in node cycles caused, for example, by C header file include loops.
get_next ()  
Return the next node for this walk of the tree.
    This function is intentionally iterative, not recursive, to sidestep any issues of stack size limitations.
is_done () → bool
SCons.Node.changed_since_last_build_alias (node, target, prev_ni, repo_node=None) → bool
SCons.Node.changed_since_last_build_entry (node, target, prev_ni, repo_node=None) → bool
SCons.Node.changed_since_last_build_node (node, target, prev_ni, repo_node=None) → bool
Must be overridden in a specific subclass to return True if this Node (a dependency) has changed since the last time
it was used to build the specified target. prev_ni is this Node’s state (for example, its file timestamp, length, maybe
content signature) as of the last time the target was built.

Note that this method is called through the dependency, not the target, because a dependency Node must be able to
use its own logic to decide if it changed. For example, File Nodes need to obey if we’re configured to use timestamps,
but Python Value Nodes never use timestamps and always use the content. If this method were called through the
target, then each Node’s implementation of this method would have to have more complicated logic to handle all the
different Node types on which it might depend.

SCons.Node.changed_since_last_build_python (node, target, prev_ni, repo_node=None) → bool
SCons.Node.changed_since_last_build_state_changed (node, target, prev_ni, repo_node=None) → bool
SCons.Node.classname (obj)
SCons.Node.decide_source (node, target, prev_ni, repo_node=None) → bool
SCons.Node.decide_target (node, target, prev_ni, repo_node=None) → bool
SCons.Node.do_nothing (node, parent) → None
SCons.Node.do_nothing_node (node) → None
SCons.Node.exists_always (node) → bool
SCons.Node.exists_base (node) → bool
SCons.Node.exists_entry (node) → bool

Return if the Entry exists. Check the file system to see what we should turn into first. Assume a file if there’s no
directory.
SCons.Node.exists_file (node) → bool
SCons.Node.existsnone (node) → bool
SCons.Node.get_children (node, parent)
SCons.Node.get_contents_dir (node)

Return content signatures and names of all our children separated by new-lines. Ensure that the nodes are sorted.
SCons.Node.get_contents_entry (node)

Fetch the contents of the entry. Returns the exact binary contents of the file.
SCons.Node.get_contents_file (node)
SCons.Node.get_contents_none (node)
SCons.Node.ignore_cycle (node, stack) → None
SCons.Node.is_derived_node (node) → bool

Returns true if this node is derived (i.e. built).
SCons.Node.is_derived_none (node)
SCons.Node.reexists_base (node)
SCons.Node.reexists_node (node)
SCons.Node.reexists_none (node)
SCons.Node.store_info_file (node) → None
SCons.Node.store_info_pass (node) → None
SCons.Node.target_from_source_base (node, prefix, suffix, splitext)
SCons.Node.target_from_source_none (node, prefix, suffix, splitext)

SCons.Platform package

Submodules

SCons.Platform.aix module

Platform-specific initialization for IBM AIX systems.

There normally shouldn’t be any need to import this module directly. It will usually be imported through the generic
SCons.Platform.aix.generate (env) → None
SCons.Platform.aix.get_xlc (env, xlc=None, packages=[])
There normally shouldn’t be any need to import this module directly. It will usually be imported through the generic SCons.Platform.Platform() selection method.
SCons.Platform.cygwin.generate (env) → None

SCons.Platform.darwin module

Platform-specific initialization for Mac OS X systems.
There normally shouldn’t be any need to import this module directly. It will usually be imported through the generic SCons.Platform.Platform() selection method.
SCons.Platform.darwin.generate (env) → None

SCons.Platform.hpux module

Platform-specific initialization for HP-UX systems.
There normally shouldn’t be any need to import this module directly. It will usually be imported through the generic SCons.Platform.Platform() selection method.
SCons.Platform.hpux.generate (env) → None

SCons.Platform.irix module

Platform-specific initialization for SGI IRIX systems.
There normally shouldn’t be any need to import this module directly. It will usually be imported through the generic SCons.Platform.Platform() selection method.
SCons.Platform.irix.generate (env) → None

SCons.Platform.mingw module

Platform-specific initialization for the MinGW system.

SCons.Platform.os2 module

Platform-specific initialization for OS/2 systems.
There normally shouldn’t be any need to import this module directly. It will usually be imported through the generic SCons.Platform.Platform() selection method.
SCons.Platform.os2.generate (env) → None

SCons.Platform.posix module

Platform-specific initialization for POSIX (Linux, UNIX, etc.) systems.
There normally shouldn’t be any need to import this module directly. It will usually be imported through the generic SCons.Platform.Platform() selection method.
SCons.Platform.posix.escape (arg)
    escape shell special characters
SCons.Platform.posix.exec_popen3 (l, env, stdout, stderr)
SCons.Platform.posix.exec_subprocess (l, env)
SCons.Platform.posix.generate (env) → None
SCons.Platform.posix.piped_env_spawn (sh, escape, cmd, args, env, stdout, stderr)
SCons.Platform.posix.subprocess_spawn (sh, escape, cmd, args, env)

SCons.Platform.sunos module

Platform-specific initialization for Sun systems.
There normally shouldn’t be any need to import this module directly. It will usually be imported through the generic SCons.Platform.Platform() selection method.
SCons.Platform.sunos.generate (env) → None
SCons API Documentation

SCons.Platform.virtualenv module

"Platform" support for a Python virtualenv.

SCons.Platform.virtualenv.ImportVirtualenv(env) → None

Copies virtualenv-related environment variables from OS environment to env[\"ENV\"] and prepends virtualenv's PATH to env[\"ENV\"][\"PATH\"].

SCons.Platform.virtualenv.IsInVirtualenv(path)

Returns True, if path is under virtualenv's home directory. If not, or if we don't use virtualenv, returns False.

SCons.Platform.virtualenv._enable_virtualenv_default()

Modify environment such that SCons will take into account its virtualenv when running external tools.

SCons.Platform.virtualenv._ignore_virtualenv_default()

SCons.Platform.virtualenv._inject_venv_path(env, path_list=None) → None

SCons.Platform.virtualenv._inject_venv_variables(env) → None

SCons.Platform.virtualenv._is_path_in(path, base) → bool

Returns true if path is located under the base directory.

SCons.Platform.virtualenv._running_in_virtualenv()

Returns True if scons is executed within a virtualenv

SCons.Platform.virtualenv.select_paths_in_venv(path_list)

Returns a list of paths from path_list which are under virtualenv's home directory.

SCons.Platform.win32 module

Platform-specific initialization for Win32 systems.

There normally shouldn't be any need to import this module directly. It will usually be imported through the generic SCons.Platform.Platform() selection method.

class SCons.Platform.win32.ArchDefinition(arch, synonyms=[])

Bases: object

Determine which windows CPU were running on. A class for defining architecture-specific settings and logic.

SCons.Platform.win32.escape(x)

SCons.Platform.win32.exec_spawn(l, env)

SCons.Platform.win32.generate(env)

SCons.Platform.win32.get_architecture(arch=None)

SCons.Platform.win32.get_program_files_dir()

SCons.Platform.win32.get_system_root()

SCons.Platform.win32.piped_spawn(sh, escape, cmd, args, env, stdout, stderr)

SCons.Platform.win32.spawn(sh, escape, cmd, args, env)

SCons.Platform.win32.spawnve(mode, file, args, env)

Module contents

SCons platform selection.

Looks for modules that define a callable object that can modify a construction environment as appropriate for a given platform.

Note that we take a more simplistic view of "platform" than Python does. We're looking for a single string that determines a set of tool-independent variables with which to initialize a construction environment. Consequently, we'll examine both sys.platform and os.name (and anything else that might come in to play) in order to return some specification which is unique enough for our purposes.
Note that because this subsystem just selects a callable that can modify a construction environment, it’s possible for people to define their own “platform specification” in an arbitrary callable function. No one needs to use or tie in to this subsystem in order to roll their own platform definition.

SCons.Platform.DefaultToolList (platform, env)
Select a default tool list for the specified platform.

SCons.Platform.Platform (name='darwin')
Select a canned Platform specification.

class SCons.Platform.PlatformSpec (name, generate)
Bases: object
class SCons.Platform.TempFileMunge (cmd, cmdstr=None)
Bases: object
Convert long command lines to use a temporary file.

You can set an Environment variable (usually TEMPFILE) to this, then call it with a string argument, and it will perform temporary file substitution on it. This is used to circumvent limitations on the length of command lines.

Example:

```python
env['TEMPFILE'] = TempFileMunge
env['LINKCOM'] = '${TEMPFILE(''$LINK $TARGET $SOURCES',''$LINKCOMSTR')}'
```

By default, the name of the temporary file used begins with a prefix of '@'. This may be configured for other tool chains by setting the TEMPFILEPREFIX variable. Example:

```python
env['TEMPFILEPREFIX'] = '-@'  # diab compiler
env['TEMPFILEPREFIX'] = '-via'  # arm tool chain
env['TEMPFILEPREFIX'] = ''  # (the empty string) PC Lint
```

You can configure the extension of the temporary file through the TEMPFILESUFFIX variable, which defaults to `.lnk` (see comments in the code below). Example:

```
env['TEMPFILESUFFIX'] = '.lnt'  # PC Lint
```

Entries in the temporary file are separated by the value of the TEMPFILEARGJOIN variable, which defaults to an OS-appropriate value.

A default argument escape function is SCons.Subst.quote_spaces. If you need to apply extra operations on a command argument before writing to a temporary file (fix Windows slashes, normalize paths, etc.), please set TEMPFILEARGESCFUNC variable to a custom function. Example:

```python
import sys
import re
from SCons.Subst import quote_spaces

WINPATHSEP_RE = re.compile(r'\([^\''\]|$)')

def tempfile_arg_esc_func(arg):
    arg = quote_spaces(arg)
    if sys.platform != "win32":  
        return arg  
    # GCC requires double Windows slashes, let's use UNIX separator
    return WINPATHSEP_RE.sub(r'/\', arg)

env['TEMPFILEARGESCFUNC'] = tempfile_arg_esc_func
```
SCons API Documentation

```python
def _print_cmd_str(target, source, env, cmdstr):
    None

SCons.Platform.platform_default()
    Return the platform string for our execution environment.
    The returned value should map to one of the SCons/Platform/*.py files. Since scons is architecture independent,
    though, we don’t care about the machine architecture.

SCons.Platform.platform_module(name="darwin")
    Return the imported module for the platform.
    This looks for a module name that matches the specified argument. If the name is unspecified, we fetch the
    appropriate default for our execution environment.

SCons.Scanner package

Submodules

SCons.Scanner.C module

Dependency scanner for C/C++ code.

Two scanners are defined here: the default CScanner, and the optional CConditionalScanner, which must be explicitly
selected by calling add_scanner() for each affected suffix.

SCons.Scanner.C.CConditionalScanner()
    Return an advanced conditional Scanner instance for scanning source files
    Interprets C/C++ Preprocessor conditional syntax (#ifdef, #if, defined, #else, #elif, etc.).

SCons.Scanner.C.Scanner()
    Return a prototype Scanner instance for scanning source files that use the C pre-processor

class SCons.Scanner.C.SConsCPPConditionalScanner(*args, **kwargs)
    Bases: PreProcessor
    SCons-specific subclass of the cpp.py module’s processing.
    We subclass this so that: 1) we can deal with files represented by Nodes, not strings; 2) we can keep track of the files
    that are missing.

    _call__(file)
        Pre-processes a file.
        This is the main public entry point.
    _do_if Else_condition(condition) → None
        Common logic for evaluating the conditions on #if, #ifdef and #ifndef lines.
    _match_tuples(tuples)
    _parse_tuples(contents)
    _process_tuples(tuples, file=None)
    all include(t) → None
    do define(t) → None
        Default handling of a #define line.
    do_elif(t) → None
        Default handling of a #elif line.
    do_else(t) → None
        Default handling of a #else line.
    do endif(t) → None
        Default handling of a #endif line.
    do_if(t) → None
        Default handling of a #if line.
    do_ifdef(t) → None
        Default handling of a #ifdef line.
    do ifndef(t) → None
        Default handling of a #ifndef line.
    do_import(t) → None
        Default handling of a #import line.
    do include(t) → None
        Default handling of a #include line.
```
do_include_next (t) → None
  Default handling of a #include line.

do_nothing (t) → None
  Null method for when we explicitly want the action for a specific preprocessor directive to do nothing.

do_undef (t) → None
  Default handling of a #undef line.

eval_expression (t)
  Evaluates a C preprocessor expression.
  This is done by converting it to a Python equivalent and eval()ing it in the C preprocessor namespace we use to track #define values.

finalize_result (fname)

find_include_file (t)
  Finds the #include file for a given preprocessor tuple.

initialize_result (fname) → None

process_contents (contents)
  Pre-processes a file contents.
  Is used by tests

process_file (file)
  Pre-processes a file.
  This is the main internal entry point.

read_file (file) → str

resolve_include (t)
  Resolve a tuple-ized #include line.
  This handles recursive expansion of values without "" or <> surrounding the name until an initial " or < is found, to handle #include FILE where FILE is a #define somewhere else.

restore () → None
  Pops the previous dispatch table off the stack and makes it the current one.

save () → None
  Pushes the current dispatch table on the stack and re-initializes the current dispatch table to the default.

scons_current_file (t) → None

start_handling_includes (t=None) → None
  Causes the PreProcessor object to start processing #import, #include and #include_next lines.
  This method will be called when a #if, #ifdef, #ifndef or #elif evaluates True, or when we reach the #else in a #if, #ifdef, #ifndef or #elif block where a condition already evaluated False.

stop_handling_includes (t=None) → None
  Causes the PreProcessor object to stop processing #import, #include and #include_next lines.
  This method will be called when a #if, #ifdef, #ifndef or #elif evaluates False, or when we reach the #else in a #if, #ifdef, #ifndef or #elif block where a condition already evaluated True.

tupleize (contents)
  Turns the contents of a file into a list of easily-processed tuples describing the CPP lines in the file.
  The first element of each tuple is the line’s preprocessor directive (#if, #include, #define, etc., minus the initial ‘#’).
  The remaining elements are specific to the type of directive, as pulled apart by the regular expression.

class SCons.Scanner.C.SConsCPPConditionalScannerWrapper (name, variable)
  Bases: object
  The SCons wrapper around a cpp.py scanner.
  This is the actual glue between the calling conventions of generic SCons scanners, and the (subclass of) cpp.py class that knows how to look for #include lines with reasonably real C-preprocessor-like evaluation of #if/#ifdef/#else/#elif lines.

recursively.process (nodes)

select (node)

class SCons.Scanner.C.SConsCPPScanner (*args, **kwargs)
  Bases: PreProcessor
  SCons-specific subclass of the cpp.py module’s processing.
  We subclass this so that: 1) we can deal with files represented by Nodes, not strings; 2) we can keep track of the files that are missing.

__call__ (file)
Pre-processes a file.
This is the main public entry point.

_do_if_else_condition (condition) → None
Common logic for evaluating the conditions on #if, #ifdef and #ifndef lines.

_match_tuples (tuples)
_parse_tuples (contents)
_process_tuples (tuples, file=None)

all_include (t) → None
do_define (t) → None
Default handling of a #define line.
do_elif (t) → None
Default handling of a #elif line.
do_else (t) → None
Default handling of a #else line.
do endif (t) → None
Default handling of a #endif line.
do if (t) → None
Default handling of a #if line.
do ifndef (t) → None
Default handling of a #ifndef line.
do ifdef (t) → None
Default handling of a #ifdef line.
do import (t) → None
Default handling of a #import line.
do include (t) → None
Default handling of a #include line.
do include_next (t) → None
Default handling of a #include line.
do nothing (t) → None
Null method for when we explicitly want the action for a specific preprocessor directive to do nothing.
do undef (t) → None
Default handling of a #undef line.
eval_expression (t)
Evaluates a C preprocessor expression.
This is done by converting it to a Python equivalent and eval()ing it in the C preprocessor namespace we use to track #define values.

finalize_result (fname)
find_include_file (t)
Finds the #include file for a given preprocessor tuple.
initialize_result (fname) → None
process_contents (contents)
Pre-processes a file contents.
Is used by tests
process_file (file)
Pre-processes a file.
This is the main internal entry point.
read_file (file) → str
resolve_include (t)
Resolve a tuple-ized #include line.
This handles recursive expansion of values without “<” or “>” surrounding the name until an initial “<” or “>” is found, to handle #include FILE where FILE is a #define somewhere else.
restore () → None
Pops the previous dispatch table off the stack and makes it the current one.
save () → None
Pushes the current dispatch table on the stack and re-initializes the current dispatch table to the default.
SCons API Documentation

start_handling_includes (t=None) → None
Causes the PreProcessor object to start processing #import, #include and #include_next lines.
This method will be called when a #if, #ifdef, #ifndef or #elif evaluates True, or when we reach the #else in a #if,
#ifdef, #ifndef or #elif block where a condition already evaluated False.

stop_handling_includes (t=None) → None
Causes the PreProcessor object to stop processing #import, #include and #include_next lines.
This method will be called when a #if, #ifdef, #ifndef or #elif evaluates False, or when we reach the #else in a #if,
#ifdef, #ifndef or #elif block where a condition already evaluated True.

tupleize (contents)
Turns the contents of a file into a list of easily-processed tuples describing the CPP lines in the file.
The first element of each tuple is the line’s preprocessor directive (#if, #include, #define, etc., minus the initial ‘#’).
The remaining elements are specific to the type of directive, as pulled apart by the regular expression.

class SCons.Scanner.C.SConsCPPScannerWrapper (name, variable)
Bases: object
The SCons wrapper around a cpp.py scanner.
This is the actual glue between the calling conventions of generic SCons scanners, and the (subclass of) cpp.py
class that knows how to look for #include lines with reasonably real C-preprocessor-like evaluation of
#if/#ifdef/#else/#elif lines.

tupleize (contents)

SCons.Scanner.C.dictify_CPPDEFINES (env) → dict
Returns CPPDEFINES converted to a dict.
This should be similar to processDefines(). Unfortunately, we can’t do the simple thing of calling that routine and
passing the result to the dict() constructor, because it turns the defines into a list of “name=value” pairs, which the
dict constructor won’t consume correctly. Also cannot just call dict on CPPDEFINES itself - it’s fine if it’s stored in the
converted form (currently deque of tuples), but CPPDEFINES could be in other formats too.
So we have to do all the work here - keep concepts in sync with processDefines.

SCons.Scanner.D module
Scanner for the Digital Mars “D” programming language.
Coded by Andy Friesen, 17 Nov 2003

class SCons.Scanner.D.D
Bases: Classic
__call__ (node, env, path=()) → list
Scans a single object.

Parameters:

• node – the node that will be passed to the scanner function
• env – the environment that will be passed to the scanner function.
• path – tuple of paths from the path_function

Returns: A list of direct dependency nodes for the specified node.

static _recurse_all_nodes (nodes)
static _recurse_no_nodes (nodes)
add_scanner (skey, scanner) → None
add_skey (skey) → None
Add a skey to the list of skeys

static find_include (include, source_dir, path)
find_include_names (node)
get_skeys (env=None)
path (env, dir=None, target=None, source=None)
scan (node, path=())
select (node)
static sort_key (include)
SCons.Scanner.D.DScanner ()
Return a prototype Scanner instance for scanning D source files
SCons.Scanner.Dir module

SCons.Scanner.Dir.DirEntryScanner (**kwargs)
Return a prototype Scanner instance for “scanning” directory Nodes for their in-memory entries
SCons.Scanner.Dir.DirScanner (**kwargs)
Return a prototype Scanner instance for scanning directories for on-disk files
SCons.Scanner.Dir.do_not_scan (k)
SCons.Scanner.Dir.only_dirs (nodes)
“Scans” a Node.FS.Dir for its in-memory entries.
SCons.Scanner.Dir.scan_in_memory (node, env, path=())
Scans a directory for on-disk files and directories therein.
Looking up the entries will add these to the in-memory Node tree representation of the file system, so all we have to
do is just that and then call the in-memory scanning function.

SCons.Scanner.Fortran module

Dependency scanner for Fortran code.
class SCons.Scanner.Fortran.F90Scanner (name, suffixes, path_variable, use_regex, incl_regex, def_regex, *args, **kwargs)
Bases: Classic
A Classic Scanner subclass for Fortran source files which takes into account both USE and INCLUDE statements.
This scanner will work for both F77 and F90 (and beyond) compilers.
Currently, this scanner assumes that the include files do not contain USE statements. To enable the ability to deal
with USE statements in include files, add logic right after the module names are found to loop over each include file,
search for and locate each USE statement, and append each module name to the list of dependencies. Caching the
search results in a common dictionary somewhere so that the same include file is not searched multiple times would
be a smart thing to do.
__call__ (node, env, path=()) → list
Scans a single object.

Parameters:
• node – the node that will be passed to the scanner function
• env – the environment that will be passed to the scanner function.

Returns: A list of direct dependency nodes for the specified node.
static _recurse_all_nodes (nodes)
static _recurse_no_nodes (nodes)
add_scanner (skey, scanner) → None
add_skey (skey) → None
  Add a skey to the list of skeys
static find_include (include, source_dir, path)
find_include_names (node)
get_skeys (env=None)
path (env, dir=None, target=None, source=None)
scan (node, env, path=())
select (node)
static sort_key (include)
SCons.Scanner.Fortran.FortranScan (path_variable: str = 'FORTRANPATH')
Return a prototype Scanner instance for scanning source files for Fortran USE & INCLUDE statements

SCons.Scanner.IDL module

Dependency scanner for IDL (Interface Definition Language) files.
SCons.Scanner.IDL.IDLScan ()
Return a prototype Scanner instance for scanning IDL source files
SCons API Documentation

SCons.Scanner.Java module

SCons.Scanner.Java.JavaScanner ()
Scanner for .java files.
New in version 4.4.

SCons.Scanner.Java._collect_classes (classlist, dirname, files) → None
Return a list of substituted path elements.
If paths is a string, it is split on the search-path separator. Otherwise, substitution is done on string-valued list
elements but they are not split.

Note helps support behavior like pulling in the external CLASSPATH and setting it directly into JAVACLASSPATH,
however splitting on os.pathsep makes the interpretation system-specific (this is warned about in the manpage
entry for JAVACLASSPATH).

SCons.Scanner.Java.scan (node, env, libpath=()) → list
Scan for files both on JAVACLASSPATH and JAVAPROCESSORPATH.

JAVACLASSPATH/JAVAPROCESSORPATH path can contain:

- Explicit paths to JAR/Zip files
- Wildcards (*)
- Directories which contain classes in an unnamed package
- Parent directories of the root package for classes in a named package

Class path entries that are neither directories nor archives (.zip or JAR files) nor the asterisk (*) wildcard character
are ignored.

SCons.Scanner.LaTeX module

Dependency scanner for LaTeX code.

class SCons.Scanner.LaTeX.FindENVPathDirs (variable)
Bases: object
A class to bind a specific E[*]PATH variable name to a function that will return all of the E[*]path directories.

class SCons.Scanner.LaTeX.LaTeX (name, suffixes, graphics_extensions, *args, **kwargs)
Bases: ScannerBase
Class for scanning LaTeX files for included files.
Unlike most scanners, which use regular expressions that just return the included file name, this returns a tuple
consisting of the keyword for the inclusion ("include", "includegraphics", "input", or "bibliography"), and then the file
name itself. Based on a quick look at LaTeX documentation, it seems that we should append .tex suffix for the
"include" keywords, append .tex if there is no extension for the "input" keyword, and need to add .bib for the
"bibliography" keyword that does not accept extensions by itself.
Finally, if there is no extension for an "includegraphics" keyword latex will append .ps or .eps to find the file, while
pdflatex may use .pdf, .jpg, .tif, .mps, or .png.
The actual subset and search order may be altered by DeclareGraphicsExtensions command. This complication is
ignored. The default order corresponds to experimentation with teTeX:

```
$ latex --version
pdfeTeX 3.141592-1.21a-2.2 (Web2C 7.5.4)
kpathsea version 3.5.4
```

The order is:

[.eps, .ps] for latex [.png', '.pdf', '.jpg', '.tif']

Another difference is that the search path is determined by the type of the file being searched: env[TEXINPUTS] for
"input" and "include" keywords env[TEXINPUTS] for "includegraphics" keyword env[TEXINPUTS] for
"listinputlisting" keyword env[BIBINPUTS] for "bibliography" keyword env[BSTINPUTS] for "bibliographystyle"
keyword env[INDEXSTYLE] for "makeindex" keyword, no scanning support needed just allows user to set it if
needed.
_call__ (node, env, path=()) → list

Scans a single object.

**Parameters:**

- **node** – the node that will be passed to the scanner function
- **env** – the environment that will be passed to the scanner function.
- **path** – tuple of paths from the path_function

**Returns:** A list of direct dependency nodes for the specified node.

__latex_names__ (include_type, filename)

static _recurse_all_nodes (nodes)

static _recurse_no_nodes (nodes)

add_scanner (skey, scanner) → None

Add a skey to the list of skeys

canonical_text (text)

  Standardize an input TeX-file contents.

**Currently:**

- removes comments, unwrapping comment-wrapped lines.

  ```
  env_variables = ['TEXINPUTS', 'BIBINPUTS', 'BSTINPUTS', 'INDEXSTYLE']
  find_include = (include, source_dir, path)
  get_skeys (env=None)
  keyword_paths =
    {'addbibresource': 'BIBINPUTS', 'addglobalbib': 'BIBINPUTS', 'addsectionbib': 'BIBINPUTS',
     'bibliography': 'BIBINPUTS', 'bibliographystyle': 'BSTINPUTS', 'bibliographystyle': 'BSTINPUTS',
     'index': 'INDEXSTYLE', 'include': 'TEXINPUTS',
     'includegraphics': 'TEXINPUTS', 'input': 'TEXINPUTS',
     'inputlisting': 'TEXINPUTS', 'makeindex': 'INDEXSTYLE',
     'usepackage': 'TEXINPUTS'}
  path (env, dir=None, target=None, source=None)
  scan (node, subdir: str = '.')
  scan_recursion (node, path=())
  ```

  do a recursive scan of the top level target file This lets us search for included files based on the directory of the
  main file just as latex does

  select (node)

  static sort_key (include)

  two_arg_commands = ['import', 'subimport', 'includefrom', 'subincludefrom', 'inputfrom', 'subinputfrom']

SCons.Scanner.LaTeX.LaTeXScanner ()

  Return a prototype Scanner instance for scanning LaTeX source files when built with latex.

SCons.Scanner.LaTeX.PDFLaTeXScanner ()

  Return a prototype Scanner instance for scanning LaTeX source files when built with pdflatex.

class SCons.Scanner.LaTeX._Null

  Bases: object

  SCons.Scanner.LaTeX._null

  alias of _Null

SCons.Scanner.LaTeX.modify_env_var (env, var, abspath)

SCons.Scanner.Prog module

Dependency scanner for program files.

SCons.Scanner.Prog.ProgramScanner (**kwargs)

  Return a prototype Scanner instance for scanning executable files for static-lib dependencies

SCons.Scanner.Prog._subst_libs (env, libs)

  Substitute environment variables and split into list.

SCons.Scanner.Prog.scan (node, env, libpath=())

  Scans program files for static-library dependencies.
It will search the LIBPATH environment variable for libraries specified in the LIBS variable, returning any files it finds as dependencies.

SCons.Scanner.RC module

Dependency scanner for RC (Interface Definition Language) files.
SCons.Scanner.RC.RCScan ()
  Return a prototype Scanner instance for scanning RC source files
SCons.Scanner.RC.no_tlb (nodes)
  Filter out .tlb files as they are binary and shouldn’t be scanned.

SCons.Scanner.SWIG module

Dependency scanner for SWIG code.
SCons.Scanner.SWIG.SWIGScanner ()

Module contents

The Scanner package for the SCons software construction utility.
SCons.Scanner.Base
  alias of ScannerBase

class SCons.Scanner.Classic (name, suffixes, path_variable, regex, *args, **kwargs)
  Bases: Current
  A Scanner subclass to contain the common logic for classic CPP-style include scanning, but which can be customized to use different regular expressions to find the includes.
  Note that in order for this to work “out of the box” (without overriding the find_include() and sort_key1() methods), the regular expression passed to the constructor must return the name of the include file in group 0.
  __call__ (node, env, path=()) → list
    Scans a single object.

Parameters:
  * node – the node that will be passed to the scanner function
  * env – the environment that will be passed to the scanner function.

Returns:
  A list of direct dependency nodes for the specified node.

static _recurse_all_nodes (nodes)
static _recurse_no_nodes (nodes)
add_scanner (skey, scanner) → None
add_skey (skey) → None
  Add a skey to the list of skyes
static find_include (include, source_dir, path)
find_include_names (node)
get_skeys (env=None)
path (env, dir=None, target=None, source=None)
scan (node, path=())
select (node)
static sort_key (include)

class SCons.Scanner.ClassicCPP (name, suffixes, path_variable, regex, *args, **kwargs)
  Bases: Classic
  A Classic Scanner subclass which takes into account the type of bracketing used to include the file, and uses classic CPP rules for searching for the files based on the bracketing.
  Note that in order for this to work, the regular expression passed to the constructor must return the leading bracket in group 0, and the contained filename in group 1.
  __call__ (node, env, path=()) → list
    Scans a single object.
Parameters:

- **node** – the node that will be passed to the scanner function
- **env** – the environment that will be passed to the scanner function.
- **path** – tuple of paths from the *path_function*

Returns: A list of direct dependency nodes for the specified node.

```python
static _recurse_all_nodes (nodes)
static _recurse_no_nodes (nodes)
add_scanner (skey, scanner) → None
add_skey (skey) → None
```

Add a skey to the list of skeys

```python
static find_include (include, source_dir, path)
find_include_names (node)
get_skeys (env=None)
path (env, dir=None, target=None, source=None)
scan (node, path=())
select (node)
```

```python
static sort_key (include)
class SCons.Scanner.Current (*args, **kwargs)
```

A class for scanning files that are source files (have no builder) or are derived files and are current (which implies that they exist, either locally or in a repository).

```python
__call__ (node, env, path=()) → list
```

Scans a single object.

Parameters:

- **node** – the node that will be passed to the scanner function
- **env** – the environment that will be passed to the scanner function.
- **path** – tuple of paths from the *path_function*

Returns: A list of direct dependency nodes for the specified node.

```python
static _recurse_all_nodes (nodes)
static _recurse_no_nodes (nodes)
add_scanner (skey, scanner) → None
add_skey (skey) → None
```

Get a skey to the list of skeys

```python
get_skeys (env=None)
path (env, dir=None, target=None, source=None)
select (node)
```

```python
class SCons.Scanner.FindPathDirs (variable)
```

Class to bind a specific E[*]PATH variable name to a function that will return all of the E[*]path directories.

```python
SCons.Scanner.Scanner (function, *args, **kwargs)
```

Factory function to create a Scanner Object.

```python
creates the appropriate Scanner based on the type of “function”.
```

TODO: Deprecate this some day. We’ve moved the functionality inside the ScannerBase class and really don’t need this factory function any more. It was, however, used by some of our Tool modules, so the call probably ended up in various people's custom modules patterned on SCons code.

```python
class SCons.Scanner.ScannerBase (function, name: str = 'NONE', argument=<class 'SCons.Scanner._Null'>, skeys=<class 'SCons.Scanner._Null'>, path_function=None, node_class=<class 'SCons.Node.FS.Base'>, node_factory=None, scan_check=None, recursive=None)
```

Bases: object

Base class for dependency scanners.

Implements straightforward, single-pass scanning of a single file.

A Scanner is usually set up with a scanner function (and optionally a path function), but can also be a kind of dispatcher which passes control to other Scanners.
A scanner function takes three arguments: a Node to scan for dependencies, the construction environment to use, and an optional tuple of paths (as generated by the optional path function). It must return a list containing the Nodes for all the direct dependencies of the file.

The optional path function is called to return paths that can be searched for implicit dependency files. It takes five arguments: a construction environment, a Node for the directory containing the SConscript file that defined the primary target, a list of target nodes, a list of source nodes, and the optional argument for this instance.

Examples:

```python
s = Scanner(my_scanner_function)
s = Scanner(function=my_scanner_function)
s = Scanner(function=my_scanner_function, argument='foo')
```

Parameters:

- **function** – either a scanner function taking two or three arguments and returning a list of File Nodes; or a mapping of keys to other Scanner objects.
- **name** – an optional name for identifying this scanner object (defaults to “NONE”).
- **argument** – an optional argument that will be passed to both function and path_function.
- **skeys** – an optional list argument that can be used to determine if this scanner can be used for a given Node. In the case of File nodes, for example, the skeys would be file suffixes.
- **path_function** – an optional function which returns a tuple of the directories that can be searched for implicit dependency files. May also return a callable which is called with no args and returns the tuple (supporting Bindable class).
- **node_class** – optional class of Nodes which this scan will return. If not specified, defaults to SCons.Node.FS.Base. If node_class is None, then this scanner will not enforce any Node conversion and will return the raw results from function.
- **node_factory** – optional factory function to be called to translate the raw results returned by function into the expected node_class objects.
- **scan_check** – optional function to be called to first check whether this node really needs to be scanned.
- **recursive** – optional specifier of whether this scanner should be invoked recursively on all of the implicit dependencies it returns (for example #include lines in C source files, which may refer to header files which should themselves be scanned). May be a callable, which will be called to filter the list of nodes found to select a subset for recursive scanning (the canonical example being only recursively scanning subdirectories within a directory). The default is to not do recursive scanning.

```python
__call__ (node, env, path=()) → list
Scans a single object.
```

Parameters:

- **node** – the node that will be passed to the scanner function
- **env** – the environment that will be passed to the scanner function.
- **path** – tuple of paths from the path_function

Returns: A list of direct dependency nodes for the specified node.

```python
static _recurse_all_nodes (nodes)
static _recurse_no_nodes (nodes)
add_scanner (skey, scanner) → None
add_skey (skey) → None
Add a skey to the list of skeys
get_skeys (env=None)
path (env, dir=None, target=None, source=None)
```
select (node)

class SCons.Scanner.Selector (mapping, *args, **kwargs)
    Bases: ScannerBase
    A class for selecting a more specific scanner based on the scanner_key() (suffix) for a specific Node.
    TODO: This functionality has been moved into the inner workings of the ScannerBase class, and this class will be
    deprecated at some point. (It was never exposed directly as part of the public interface, although it is used by the
    Scanner() factory function that was used by various Tool modules and therefore was likely a template for custom
    modules that may be out there.)

static _recurse_all_nodes (nodes)
static _recurse_no_nodes (nodes)

add_scanner (skey, scanner) → None
add_skey (skey) → None
    Add a skey to the list of skeys
get_skeys (env=None)
p=ath (env, dir=None, target=None, source=None)
select (node)

class SCons.Scanner._Null
    Bases: object
    SCons.Scanner._null
    alias of _Null

SCons.Script package

Submodules

SCons.Script.Interactive module

class SCons.Script.Interactive.SConsInteractiveCmd (**kw)
    Bases: Cmd
    build [TARGETS] Build the specified TARGETS and their dependencies. ‘b’ is a synonym. clean [TARGETS] Clean
    (remove) the specified TARGETS and their dependencies. ‘c’ is a synonym. exit Exit SCons interactive mode. help
    [COMMAND] Prints help for the specified COMMAND. ‘h’ and ‘?’ are synonyms. shell [COMMANDLINE] Execute
    COMMANDLINE in a subshell. ‘sh’ and ‘!’ are synonyms. version Prints SCons version information.

_do_one_help (arg) → None
_doc_to_help (obj)
_strip_initial_spaces (s)
cmdloop (intro=None)
    Repeatedly issue a prompt, accept input, parse an initial prefix off the received input, and dispatch to action
    methods, passing them the remainder of the line as argument.
columnize (list, displaywidth=80)
    Display a list of strings as a compact set of columns.
    Each column is only as wide as necessary. Columns are separated by two spaces (one was not legible enough).
complete (text, state)
    Return the next possible completion for ‘text’.
    If a command has not been entered, then complete against command list. Otherwise try to call
    complete_<command> to get list of completions.
complete_help (**args)
completesdefault (**ignored)
    Method called to complete an input line when no command-specific complete_*() method is available.
    By default, it returns an empty list.
completenames (text, **ignored)
default (argv) → None
    Called on an input line when the command prefix is not recognized.
    If this method is not overridden, it prints an error message and returns.
do_EOF (argv) → None
do_build (argv) → None
SCons API Documentation

build [TARGETS] Build the specified TARGETS and their dependencies. 'b' is a synonym.
do_clean (argv)
clean [TARGETS] Clean (remove) the specified TARGETS and their dependencies. 'c' is a synonym.
do_exit (argv) → None
   exit Exit SCons interactive mode.
do_help (argv) → None
   help [COMMAND] Prints help for the specified COMMAND. 'h' and '?' are synonyms.
do_shell (argv) → None
   shell [COMMANDLINE] Execute COMMANDLINE in a subshell. 'sh' and '!' are synonyms.
do_version (argv) → None
   version Prints SCons version information.
doc_header = 'Documented commands (type help <topic>):'
doc_leader = ''
emptyline ()
   Called when an empty line is entered in response to the prompt.
   If this method is not overridden, it repeats the last nonempty command entered.
ge_names ()
identchars = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_'
intro = None
lastcmd = ''
misc_header = 'Miscellaneous help topics:'
nohelp = '*** No help on %s'
onecmd (line)
   Interpret the argument as though it had been typed in response to the prompt.
   This may be overridden, but should not normally need to be; see the precmd() and postcmd() methods for useful
   execution hooks. The return value is a flag indicating whether interpretation of commands by the interpreter should
   stop.
parseline (line)
   Parse the line into a command name and a string containing the arguments. Returns a tuple containing (command,
   args, line). 'command' and 'args' may be None if the line couldn't be parsed.
postcmd (stop, line)
   Hook method executed just after a command dispatch is finished.
postloop ()
   Hook method executed once when the cmdloop() method is about to return.
prenm (line)
   Hook method executed just before the command line is interpreted, but after the input prompt is generated and
   issued.
preloop ()
   Hook method executed once when the cmdloop() method is called.
print_topics (header, cmds, cmdlen, maxcol)
   'Cmd' '
   prompt = '(Cmd) '
ruler = '='
synonyms = {'b': 'build', 'c': 'clean', 'h': 'help', 'scons': 'build', 'sh': 'shell'}
undoc_header = 'Undocumented commands:'
use_rawinput = 1
SCons.Script.Interactive.interact (fs, parser, options, targets, target_top) → None

SCons.Script.Main module

The main() function used by the scons script.

Architecturally, this is the scons script, and will likely only be called from the external "scons" wrapper. Consequently,
anything here should not be, or be considered, part of the build engine. If it's something that we expect other software
to want to use, it should go in some other module. If it's specific to the "scons" script invocation, it goes here.
SCons.Script.Main.AddOption (*args, **kw)
class SCons.Script.Main.BuildTask (tm, targets, top, node)
   Bases: OutOfDateTask
An SCons build task.

LOGGER = None

_abc_impl = <_abc._abc_data object>

_exception_raise ()

Raises a pending exception that was recorded while getting a Task ready for execution.

_no_exception_to_raise () → None

display (message) → None

Hook to allow the calling interface to display a message.

This hook gets called as part of preparing a task for execution (that is, a Node to be built). As part of figuring out what Node should be built next, the actual target list may be altered, along with a message describing the alteration. The calling interface can subclass Task and provide a concrete implementation of this method to see those messages.

do_failed (status: int = 2) → None

exc_clear () → None

Clears any recorded exception.

This also changes the "exception_raise" attribute to point to the appropriate do-nothing method.

exc_info ()

Returns info about a recorded exception.

exception_set (exception=None) → None

Records an exception to be raised at the appropriate time.

This also changes the "exception_raise" attribute to point to the method that will, in fact

execute () → None

Called to execute the task.

This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in prepare(), executed() or failed().

executed ()

Called when the task has been successfully executed and the Taskmaster instance wants to call the Node's callback methods.

This may have been a do-nothing operation (to preserve build order), so we must check the node's state before deciding whether it was "built", in which case we call the appropriate Node method. In any event, we always call "visited()", which will handle any post-visit actions that must take place regardless of whether or not the target was an actual built target or a source Node.

executed_with_callbacks () → None

Called when the task has been successfully executed and the Taskmaster instance wants to call the Node's callback methods.

This may have been a do-nothing operation (to preserve build order), so we must check the node's state before deciding whether it was "built", in which case we call the appropriate Node method. In any event, we always call "visited()", which will handle any post-visit actions that must take place regardless of whether or not the target was an actual built target or a source Node.

executed_without_callbacks () → None

Called when the task has been successfully executed and the Taskmaster instance doesn't want to call the Node's callback methods.

fail_continue () → None

Explicit continue-the-build failure.

This sets failure status on the target nodes and all of their dependent parent nodes.

Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

fail_stop () → None

Explicit stop-the-build failure.

This sets failure status on the target nodes and all of their dependent parent nodes.

Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

failed () → None

Default action when a task fails: stop the build.

Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().
get_target ()
Fetch the target being built or updated by this task.

make_ready () → None
Make a task ready for execution

make_ready_all () → None
Marks all targets in a task ready for execution.
This is used when the interface needs every target Node to be visited—the canonical example being the “scons -c” option.

make_ready_current ()
Marks all targets in a task ready for execution if any target is not current.
This is the default behavior for building only what’s necessary.

needs_execute () → bool
Returns True (indicating this Task should be executed) if this Task’s target state indicates it needs executing, which has already been determined by an earlier up-to-date check.

postprocess () → None
Post-processes a task after it's been executed.
This examines all the targets just built (or not, we don’t care if the build was successful, or even if there was no build because everything was up-to-date) to see if they have any waiting parent Nodes, or Nodes waiting on a common side effect, that can be put back on the candidates list.

prepare ()
Called just before the task is executed.
This is mainly intended to give the target Nodes a chance to unlink underlying files and make all necessary directories before the Action is actually called to build the targets.

trace_message (node, description: str = 'node') → None

class SCons.Script.Main.CleanTask (tm, targets, top, node)
Bases: AlwaysTask
An SCons clean task.

LOGGER = None

_abc_impl = <_abc._abc_data object>

_clean_targets (remove: bool = True) → None

_exception_raise ()
Raises a pending exception that was recorded while getting a Task ready for execution.

_get_files_to_clean ()

_no_exception_to_raise () → None

display (message) → None

Hook to allow the calling interface to display a message.
This hook gets called as part of preparing a task for execution (that is, a Node to be built). As part of figuring out what Node should be built next, the actual target list may be altered, along with a message describing the alteration. The calling interface can subclass Task and provide a concrete implementation of this method to see those messages.

exc_clear () → None
Clears any recorded exception.
This also changes the “exception_raise” attribute to point to the appropriate do-nothing method.

exc_info ()
Returns info about a recorded exception.

exception_set (exception=None) → None
Records an exception to be raised at the appropriate time.
This also changes the “exception_raise” attribute to point to the method that will, in fact

execute () → None
Called to execute the task.
This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in prepare(), executed() or failed().

executed () → None
Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s callback methods.

executed_with_callbacks () → None
Called when the task has been successfully executed and the Taskmaster instance wants to call the Node's callback methods.
This may have been a do-nothing operation (to preserve build order), so we must check the node's state before deciding whether it was "built", in which case we call the appropriate Node method. In any event, we always call "visited()", which will handle any post-visit actions that must take place regardless of whether or not the target was an actual built target or a source Node.

executed_without_callbacks () → None
Called when the task has been successfully executed and the Taskmaster instance doesn't want to call the Node's callback methods.

fail_continue () → None
Explicit continue-the-build failure.
This sets failure status on the target nodes and all of their dependent parent nodes.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

fail_stop () → None
Explicit stop-the-build failure.
This sets failure status on the target nodes and all of their dependent parent nodes.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

failed () → None
Default action when a task fails: stop the build.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

fs_delete (path, pathstr, remove: bool = True)

get_target ()
Fetch the target being built or updated by this task.

make_ready () → None
Marks all targets in a task ready for execution.
This is used when the interface needs every target Node to be visited—the canonical example being the "scons -c" option.

make_ready_all () → None
Marks all targets in a task ready for execution.
This is used when the interface needs every target Node to be visited—the canonical example being the "scons -c" option.

make_ready_current ()
Marks all targets in a task ready for execution if any target is not current.
This is the default behavior for building only what’s necessary.

needs_execute () → bool
Always returns True (indicating this Task should always be executed).
Subclasses that need this behavior (as opposed to the default of only executing Nodes that are out of date w.r.t. their dependencies) can use this as follows:

class MyTaskSubclass(SCons.Taskmaster.Task):
    needs_execute = SCons.Taskmaster.AlwaysTask.needs_execute

postprocess () → None
Post-processes a task after it’s been executed.
This examines all the targets just built (or not, we don’t care if the build was successful, or even if there was no build because everything was up-to-date) to see if they have any waiting parent Nodes, or Nodes waiting on a common side effect, that can be put back on the candidates list.

prepare () → None
Called just before the task is executed.
This is mainly intended to give the target Nodes a chance to unlink underlying files and make all necessary directories before the Action is actually called to build the targets.

remove () → None

show () → None

trace_message (node, description: str = 'node') → None
SCons API Documentation

SCons.Script.Main.DebugOptions (json=None)
API to allow specifying options to SCons debug logic Currently only json is supported which changes the json file written by –debug=json from the default

class SCons.Script.Main.FakeOptionParser
Bases: object
A do-nothing option parser, used for the initial OptionsParser value.
During normal SCons operation, the OptionsParser is created right away by the main() function. Certain test scripts however, can introspect on different Tool modules, the initialization of which can try to add a new, local option to an otherwise uninitialized OptionsParser object. This allows that introspection to happen without blowing up.
class FakeOptionValues
Bases: object

add_local_option (*args, **kw) → None
values = <SCons.Script.Main.FakeOptionParser.FakeOptionValues object>

SCons.Script.Main.GetBuildFailures ()
SCons.Script.Main.GetOption (name)
SCons.Script.Main.PrintHelp (file=None, local_only: bool = False) → None
SCons.Script.Main.Progress (*args, **kw) → None
class SCons.Script.Main.Progressor (obj, interval: int = 1, file=None, overwrite: bool = False)
Bases: object
count = 0
erase_previous () → None
prev = ''
replace_string (node) → None
spinner (node) → None
string (node) → None
target_string = '$TARGET'
write (s) → None
class SCons.Script.Main.QuestionTask (tm, targets, top, node)
Bases: AlwaysTask
An SCons task for the -q (question) option.
LOGGER = None
_abc_impl = <_abc._abc_data object>
_exception_raise ()
Raises a pending exception that was recorded while getting a Task ready for execution.
_no_exception_to_raise () → None
display (message) → None
Hook to allow the calling interface to display a message.
This hook gets called as part of preparing a task for execution (that is, a Node to be built). As part of figuring out what Node should be built next, the actual target list may be altered, along with a message describing the alteration. The calling interface can subclass Task and provide a concrete implementation of this method to see those messages.
exc_clear () → None
Clears any recorded exception.
This also changes the “exception_raise” attribute to point to the appropriate do-nothing method.
exc_info ()
Returns info about a recorded exception.
exception_set (exception=None) → None
Records an exception to be raised at the appropriate time.
This also changes the “exception_raise” attribute to point to the method that will, in fact
execute () → None
Called to execute the task.
This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in prepare(), executed() or failed().
executed () → None
Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s callback methods.
This may have been a do-nothing operation (to preserve build order), so we must check the node’s state before
deciding whether it was “built”, in which case we call the appropriate Node method. In any event, we always call
“visited()”, which will handle any post-visit actions that must take place regardless of whether or not the target was
an actual built target or a source Node.

executed_with_callbacks () → None
Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s
callback methods.

executed_without_callbacks () → None
Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s
callback methods.

fail_continue () → None
Explicit continue-the-build failure.
This sets failure status on the target nodes and all of their dependent parent nodes.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on
up-to-date nodes when using Configure().

fail_stop () → None
Explicit stop-the-build failure.
This sets failure status on the target nodes and all of their dependent parent nodes.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on
up-to-date nodes when using Configure().

failed () → None
Default action when a task fails: stop the build.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on
up-to-date nodes when using Configure().

get_target ()
Fetch the target being built or updated by this task.

make_ready ()
Marks all targets in a task ready for execution if any target is not current.
This is the default behavior for building only what’s necessary.

make_ready_all () → None
Marks all targets in a task ready for execution.
This is used when the interface needs every target Node to be visited—the canonical example being the “scons -c”
option.

make_ready_current ()
Marks all targets in a task ready for execution if any target is not current.
This is the default behavior for building only what’s necessary.

needs_execute () → bool
Always returns True (indicating this Task should always be executed).
Subclasses that need this behavior (as opposed to the default of only executing Nodes that are out of date w.r.t.
their dependencies) can use this as follows:

    class MyTaskSubclass(SCons.Taskmaster.Task):
        needs_execute = SCons.Taskmaster.AlwaysTask.needs_execute

postprocess () → None
Post-processes a task after it’s been executed.
This examines all the targets just built (or not, we don’t care if the build was successful, or even if there was no
build because everything was up-to-date) to see if they have any waiting parent Nodes, or Nodes waiting on a
common side effect, that can be put back on the candidates list.

prepare () → None
Called just before the task is executed.
This is mainly intended to give the target Nodes a chance to unlink underlying files and make all necessary
directories before the Action is actually called to build the targets.
trace_message(node, description: str = 'node') → None

exception SCons.Script.Main.SConsPrintHelpException
    Bases: Exception
    add_note() — add a note to the exception
    args
    with_traceback() — set self.__traceback__ to tb and return self.

SCons.Script.Main.SetOption(name, value)

    Bases: object
    display(t) → None
    get_all_children(node)
    getDerivedChildren(node)

SCons.Script.Main.ValidateOptions(throw_exception: bool = False) → None
    Validate options passed to SCons on the command line.
    Checks that all options given on the command line are known to this instance of SCons. Call after all of the cli options have been set up through AddOption() calls. For example, if you added an option --xyz and you call SCons with --xyy you can cause SCons to issue an error message and exit by calling this function.

    Parameters:
    throw_exception — if an invalid option is present on the command line, raises an exception if this optional parameter evaluates true; if false (the default), issue a message and exit with error status.

    Raises:
    SConsBadOptionError — If throw_exception is true and there are invalid options on the command line.

    New in version 4.5.0.

SCons.Script.Main._SConstruct_exists(dirname: str, repositories: List[str], filelist: List[str]) → str | None
    Check that an SConstruct file exists in a directory.

    Parameters:
    dirname — the directory to search. If empty, look in cwd.
    repositories — a list of repositories to search in addition to the project directory tree.
    filelist — names of SConstruct file(s) to search for. If empty list, use the built-in list of names.

    Returns: The path to the located SConstruct file, or None.

SCons.Script.Main._build_targets(fs, options, targets, target_top)
SCons.Script.Main._create_path(plist)
SCons.Script.Main._exec_main(parser, values) → None
SCons.Script.Main._load_all_site_scons_dirs(topdir, verbose: bool = False) → None
    Load all of the predefined site_scons dir. Order is significant; we load them in order from most generic (machine-wide) to most specific (topdir). The verbose argument is only for testing.
SCons.Script.Main._load_site_scons_dir(topdir, site_dir_name=None)
    Load the site directory under topdir.
    If a site dir name is supplied use it, else use default “site_scons” Prepend site dir to sys.path. If a “site_tools” subdir exists, prepend to toolpath. Import “site_init.py” from site dir if it exists.
SCons.Script.Main._main(parser)
SCons.Script.Main._scons_internal_error() → None
    Handle all errors but user errors. Print out a message telling the user what to do in this case and print a normal trace.
SCons.Script.Main._scons_internal_warning(e) → None
    Slightly different from _scons_user_warning in that we use the current call stack rather than sys.exc_info() to get our stack trace. This is used by the warnings framework to print warnings.
SCons.Script.Main._scons_syntax_error(e) → None
    Handle syntax errors. Print out a message and show where the error occurred.
SCons.Script.Main._scons_user_error(e) → None
Handle user errors. Print out a message and a description of the error, along with the line number and routine where it occurred. The file and line number will be the deepest stack frame that is not part of SCons itself.

SCons.Script.Main._scons_user_warning (e) → None

Handle user warnings. Print out a message and a description of the warning, along with the line number and routine where it occurred. The file and line number will be the deepest stack frame that is not part of SCons itself.

SCons.Script.Main._set_debug_values (options) → None

SCons.Script.Main.find_deepest_user_frame (tb)

Find the deepest stack frame that is not part of SCons.

Input is a "pre-processed" stack trace in the form returned by traceback.extract_tb() or traceback.extract_stack()

SCons.Script.Main.main () → None

SCons.Script.Main.path_string (label, module) → str

SCons.Script.Main.python_version_deprecated (version=sys.version_info(major=3, minor=11, micro=8, releaselevel='final', serial=0))

SCons.Script.Main.python_version_string ()

SCons.Script.Main.python_version_unsupported (version=sys.version_info(major=3, minor=11, micro=8, releaselevel='final', serial=0))

SCons.Script.Main.revert_io () → None

SCons.Script.Main.test_load_all_site_scons_dirs (d) → None

SCons.Script.Main.version_string (label, module)

SCons.Script.SConsOptions module

SCons.Script.SConsOptions.Parser (version)

Returns a parser object initialized with the standard SCons options.

Add options in the order we want them to show up in the -H help text, basically alphabetical. Each op.add_option() call should have a consistent format:

```python
op.add_option("-L", "--long-option-name",
            nargs=1, type="string",
            dest="long_option_name", default='foo',
            action="callback", callback=opt_long_option,
            help="help text goes here",
            metavar="VAR")
```

Even though the optparse module constructs reasonable default destination names from the long option names, we’re going to be explicit about each one for easier readability and so this code will at least show up when grepping the source for option attribute names, or otherwise browsing the source code.

exception SCons.Script.SConsOptions.SConsBadOptionError (opt_str, parser=None)

Bases: BadOptionError

Exception used to indicate that invalid command line options were specified

Variables:

- **opt_str** (str) – The offending option specified on command line which is not recognized
- **parser** (OptionParser) – The active argument parser

add_note ()

Exception.add_note(note) – add a note to the exception

args

with_traceback ()

Exception.with_traceback(tb) – set self._traceback to tb and return self.

class SCons.Script.SConsOptions.SConsIndentHelpFormatter (indent_increment=2, max_help_position=24, width=None, short_first=1)

Bases: IndentedHelpFormatter

NO_DEFAULT_VALUE = 'none'

_format_text (text)

Format a paragraph of free-form text for inclusion in the help output at the current indentation level.

dedent ()
Translates heading to “SCons Options.”
Heading of “Options” changed to “SCons Options.” Unfortunately, we have to do this here, because those titles are hard-coded in the optparse calls.

Customized option formatter.
A copy of the normal optparse.IndentedHelpFormatter.format_option() method. This has been snarfed so we can modify text wrapping to our liking:

- add our own regular expression that doesn’t break on hyphens (so things like --no-print-directory don’t get broken).
- wrap the list of options themselves when it’s too long (the wrapper.fill(opts) call below).
- set the subsequent_indent when wrapping the help_text.
The help for each option consists of two parts:
  - the opt strings and metavars e.g. (“-x”, or “-fFILENAME, –file=FILENAME”)
  - the user-supplied help string e.g. (“turn on expert mode”, “read data from FILENAME”)
If possible, we write both of these on the same line:

```
-x      turn on expert mode
```

But if the opt string list is too long, we put the help string on a second line, indented to the same column it would start in if it fit on the first line:

```
-fFILENAME, --file=FILENAME
      read data from FILENAME
```

Return a comma-separated list of option strings & metavariables.
Formats the usage message.

class SCons.Script.SConsOptions.SConsOption (*opts, **attrs)
Bases: Option

ACTIONS = ('store', 'store_const', 'store_true', 'store_false', 'append', 'append_const', 'count', 'callback', 'help', 'version')

ALWAYS_TYPED_ACTIONS = ('store', 'append')

ATTRS = ['action', 'type', 'dest', 'default', 'nargs', 'const', 'choices', 'callback', 'callback_args', 'callback_kargs', 'help', 'metavar']

CHECK_METHODS = [Function Option._check_action>, <function Option._check_type>, <function Option._check_dest>, <function Option._check_nargs>, <function Option._check_callback>, <function SConsOption._check_nargs Optional>]

CONST_ACTIONS = ('store_const', 'append_const', 'store', 'append', 'callback')
STORE_ACTIONS = ('store', 'store_const', 'store_true', 'store_false', 'append', 'append_const', 'count')
TYPED_ACTIONS = ('store', 'append', 'callback')
TYPES = ('string', 'int', 'long', 'float', 'complex', 'choice')
TYPE_CHECKER = { 'choice': <function check_choice>, 'complex': <function check_builtin>, 'float': <function check_builtin>, 'int': <function check_builtin>, 'long': <function check_builtin> }

_check_action ()
_check_callback ()
_check_choice ()
_check_const ()
_check_dest ()
_check_nargs ()
_check_nargs_optional ()
_check_opt_strings (opts)
_check_type ()
_set_attrs (attrs)
_set_opt_strings (opts)
check_value (opt, value)
convert_value (opt, value)
generate_opt_string ()
process (opt, value, values, parser)
take_action (action, dest, opt, value, values, parser)
takes_value ()
class SCons.Script.SConsOptions.SConsOptionGroup (parser, title, description=None)
Bases: OptionGroup
A subclass for SCons-specific option groups.
The only difference between this and the base class is that we print the group's help text flush left, underneath their
own title but lined up with the normal "SCons Options".
_check_conflict (option)
create_option_list ()
_create_option_mappings ()
_share_option_mappings (parser)
add_option (Option)
add_option (opt_str, ..., kwarg=val, ...) → None
add_options (option_list)
destroy ()
    see OptionParser.destroy().
format_description (formatter)
format_help (formatter)
    Format an option group's help text.
The title is dedented so it's flush with the "SCons Options" title we print at the top.
format_option_help (formatter)
get_description ()
get_option (opt_str)
has_option (opt_str)
remove_option (opt_str)
set_conflict_handler (handler)
set_description (description)
set_title (title)
class SCons.Script.SConsOptions.SConsOptionParser (usage=None, option_list=None,
    option_class=<class 'optparse.Option'>, version=None, conflict_handler='error',
    description=None, formatter=None, add_help_option=True, prog=None, epilog=None)
Bases: OptionParser
    _add_help_option ()
    _add_version_option ()
    _check_conflict (option)
    _create_option_list ()

89
SCons API Documentation

_create_option_mappings ()
_get_all_options ()
_get_args (args)
_init_parsing_state ()
_match_long_opt (opt: string) → string
   Determine which long option string ‘opt’ matches, ie. which one it is an unambiguous abbreviation for. Raises
   BadOptionError if ‘opt’ doesn’t unambiguously match any long option string.
_populate_option_list (option_list, add_help=True)
_process_args (largs, rargs, values)
   __process_args__(largs : [string],
   rargs : [string], values : Values)
   Process command-line arguments and populate ‘values’, consuming options and arguments from ‘rargs’. If
   ‘allow_interspersed_args’ is false, stop at the first non-option argument. If true, accumulate any interspersed
   non-option arguments in ‘largs’.
_process_long_opt (rargs, values)
SCons-specific processing of long options.
   This is copied directly from the normal optparse._process_long_opt() method, except that, if configured to
do so, we catch the exception thrown when an unknown option is encountered and just stick it back on the
   "leftover" arguments for later (re-)processing. This is because we may see the option definition later, while
   processing SConscript files.
_process_short_opts (rargs, values)
_share_option_mappings (parser)
add_local_option (*args, **kw)
   Adds a local option to the parser.
   This is initiated by an AddOption() call to add a user-defined command-line option. We add the option to a separate
   option group for the local options, creating the group if necessary.
add_option (Option)
add_option (opt_str, ..., kwarg=val, ...) → None
add_option_group (*args, **kwargs)
add_options (option_list)
check_values (values: Values, args: [string])
   → (values : Values, args : [string])
   Check that the supplied option values and leftover arguments are valid. Returns the option values and leftover
   arguments (possibly adjusted, possibly completely new – whatever you like). Default implementation just returns
   the passed-in values; subclasses may override as desired.
destroy ()
   Declare that you are done with this OptionParser. This cleans up reference cycles so the OptionParser (and all
   objects referenced by it) can be garbage-collected promptly. After calling destroy(), the OptionParser is unusable.
disable_interspersed_args ()
   Set parsing to stop on the first non-option. Use this if you have a command processor which runs another
   command that has options of its own and you want to make sure these options don’t get confused.
enable_interspersed_args ()
   Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is
   the default behavior. See also disable_interspersed_args() and the class documentation description of the attribute
   allow_interspersed_args.
error (msg)
   overridden OptionValueError exception handler
exit (status=0, msg=None)
expand_prog_name (s)
format_description (formatter)
format_epilog (formatter)
format_help (formatter=None)
format_local_option_help (formatter=None, file=None)
   Return the help for the project-level ("local") options.
   New in version 4.6.0.
SCons API Documentation

format_option_help (formatter=None)
get_default_values ()
get_description ()
get_option (opt_str)
get_option_group (opt_str)
get_prog_name ()
get_usage ()
get_version ()
has_option (opt_str)
parse_args (args=None, values=None)
    parse_args(args : [string] = sys.argv[1:],
               values : Values = None)
        -> (values : Values, args : [string])
        Parse the command-line options found in ‘args’ (default: sys.argv[1:]). Any errors result in a call to ‘error()’, which
        by default prints the usage message to stderr and calls sys.exit() with an error message. On success returns a pair
        (values, args) where ‘values’ is a Values instance (with all your option values) and ‘args’ is the list of arguments left
        over after parsing options.
preserve_unknown_options = False
print_help (file: file = stdout)
    Print an extended help message, listing all options and any help text provided with them, to ‘file’ (default stdout).
print_local_option_help (file=None)
    Print help for just project-defined options.
    Writes to file (default stdout).
    New in version 4.6.0.
print_usage (file: file = stdout)
    Print the usage message for the current program (self.usage) to ‘file’ (default stdout). Any occurrence of the string
    “%prog” in self.usage is replaced with the name of the current program (basename of sys.argv[0]). Does nothing if
    self.usage is empty or not defined.
print_version (file: file = stdout)
    Print the version message for this program (self.version) to ‘file’ (default stdout). As with print_usage(), any
    occurrence of “%prog” in self.version is replaced by the current program’s name. Does nothing if self.version is
    empty or undefined.
raise_exception_on_error = False
remove_option (opt_str)
reparse_local_options () -> None
    Re-parse the leftover command-line options.
    Parse options stored in self.largs, so that any value overridden on the command line is immediately available if the
    user turns around and does a GetOption() right away.
    We mimic the processing of the single args in the original OptionParser _process_args(), but here we allow exact
    matches for longOpts only (no partial argument names!). Otherwise there could be problems in add_local_option() below.
    When called from there, we try to reparse the command-line arguments that
        1. haven’t been processed so far (self.largs), but
        2. are possibly not added to the list of options yet.
    So, when we only have a value for “–myargument” so far, a command-line argument of “–myarg=test” would set it,
    per the behaviour of _match_long_opt(), which allows for partial matches of the option name, as long as the
    common prefix appears to be unique. This would lead to further confusion, because we might want to add another
    option “–myarg” later on (see issue #2929).
set_conflict_handler (handler)
set_default (dest, value)
set_defaults (**kwargs)
set_description (description)
set_process_default_values (process)
set_usage (usage)
standard_option_list = []
class SCons.Script.SConsOptions.SConsValues (defaults)
Bases: Values

Holder class for uniform access to SCons options, regardless of whether they can be set on the command line or in
the SConscript files (using the SetOption() function).

A SCons option value can originate three different ways:

1. set on the command line;
2. set in an SConscript file;
3. the default setting (from the the op.add_option() calls in the Parser() function, below).

The command line always overrides a value set in a SConscript file, which in turn always overrides default settings.
Because we want to support user-specified options in the SConscript file itself, though, we may not know about all of
the options when the command line is first parsed, so we can't make all the necessary precedence decisions at the
time the option is configured.

The solution implemented in this class is to keep these different sets of settings separate (command line, SConscript
file, and default) and to override the __getattr__() method to check them in turn. This should allow the rest of the
code to just fetch values as attributes of an instance of this class, without having to worry about where they came
from.

Note that not all command line options are settable from SConscript files, and the ones that must be explicitly
added to the “settable” list in this class, and optionally validated and coerced in the set_option() method.

__getattr__(attr)
Fetches an options value, checking first for explicit settings from the command line (which are direct attributes),
then the SConscript file settings, then the default values.

__update__ (dict, mode)
__update_careful__ (dict)
Update the option values from an arbitrary dictionary, but only use keys from dict that already have a
 corresponding attribute in self. Any keys in dict without a corresponding attribute are silently ignored.

__update_loose__ (dict)
Update the option values from an arbitrary dictionary, using all keys from the dictionary regardless of whether they
have a corresponding attribute in self or not.

guarantee_value (attr, value)
read_file (filename, mode='careful')
read_module (modname, mode='careful')

Sets an option from an SConscript file.

Raises: UserError – invalid or malformed option (“error in your script”)

settable = ['clean', 'diskcheck', 'duplicate', 'experimental', 'hash_chunksize', 'hash_format', 'help', 'implicit_cache',
'implicit_deps_changed', 'implicit_deps_unchanged', 'max_drift', 'md5_chunksize', 'no_exec', 'no_progress',
'num_jobs', 'random', 'silent', 'stack_size', 'warn', 'disable_execute_ninja', 'disable_ninja', 'skip_ninja_regen']

SCons.Script.SConsOptions.diskcheck_convert (value)
A frame on the SConstruct/SConscript call stack
SCons.Script.SConscript.Return (*vars, **kw)

class SCons.Script.SConscript.SConsEnvironment (platform=None, tools=None, toolpath=None, variables=None, parse_flags=None, **kw)
Bases: Base
An Environment subclass that contains all of the methods that are particular to the wrapper SCons interface and which aren’t (or shouldn’t be) part of the build engine itself.
Note that not all of the methods of this class have corresponding global functions, there are some private methods.

Action (*args, **kw)
AddMethod (function, name=None) → None
  Adds the specified function as a method of this construction environment with the specified name. If the name is omitted, the default name is the name of the function itself.
AddPostAction (files, action)
AddPreAction (files, action)
Alias (target, source=[], action=None, **kw)
AlwaysBuild (*targets)
Append (**kw) → None
  Append values to construction variables in an Environment.
The variable is created if it is not already present.
AppendENVPath (name, newpath, envname: str = 'ENV', sep=':', delete_existing: bool = False) → None
  Append path elements to the path name in the envname dictionary for this environment. Will only add any particular path once, and will normpath and normcase all paths to help assure this. This can also handle the case where the env variable is a list instead of a string.
  If delete_existing is False, a newpath element already in the path will not be moved to the end (it will be left where it is).
AppendUnique (delete_existing: bool = False, **kw) → None
  Append values to existing construction variables in an Environment, if they’re not already there. If delete_existing is True, removes existing values first, so values move to end.
Builder (**kw)
CacheDir (path, custom_class=None) → None
Clean (targets, files) → None
Clone (tools=[], toolpath=None, parse_flags=None, **kw)
  Return a copy of a construction Environment.
The copy is like a Python “deep copy”—that is, independent copies are made recursively of each objects—except that a reference is copied when an object is not deep-copyable (like a function). There are no references to any mutable objects in the original Environment.
Command (target, source, action, **kw)
  Builds the supplied target files from the supplied source files using the supplied action. Action may be any type that the Builder constructor will accept for an action.

Configure (*args, **kw)
Decider (function)
Default (*targets) → None
Depends (target, dependency)
  Explicitly specify that ‘target’s depend on ‘dependency’.
Detect (progs)
  Return the first available program from one or more possibilities.

Parameters:  progs (str or list) – one or more command names to check for

Dictionary (*args)
  Return construction variables from an environment.

Parameters:  *args (optional) – variable names to look up

Returns:  If args omitted, the dictionary of all construction variables. If one arg, the corresponding value is returned. If more than one arg, a list of values is returned.

Raises:  KeyError – if any of args is not in the construction environment.
Dir(name, *args, **kw)
Dump(key=None, format: str = 'pretty')
Return construction variables serialized to a string.

Parameters:
- key (optional) – if None, format the whole dict of variables. Else format the value of key (Default value = None)
- format (str, optional) – specify the format to serialize to. “pretty” generates a pretty-printed string, “json” a JSON-formatted string. (Default value = “pretty”)

static EnsurePythonVersion(major, minor) → None
Exit abnormally if the Python version is not late enough.
static EnsureSConsVersion(major, minor, revision: int = 0) → None
Exit abnormally if the SCons version is not late enough.
Entry(name, *args, **kw)
Environment(**kw)
Execute(action, *args, **kw)
Directly execute an action through an Environment

static Exit(value: int = 0) → None
Export(*vars, **kw)→ None
File(name, *args, **kw)
FindFile(file, dirs)
FindInstalledFiles()
returns the list of all targets of the Install and InstallAs Builder.
FindIxes(paths, prefix, suffix)
Search a list of paths for something that matches the prefix and suffix.

Parameters:
- paths – the list of paths or nodes.
- prefix – construction variable for the prefix.
- suffix – construction variable for the suffix.

Returns: the matched path or None
FindSourceFiles(node: str = '.') → list
Return a list of all source files.
Flatten(sequence)
GetBuildPath(files)
static GetLaunchDir() → None
GetOption(name)
Glob(pattern, ondisk: bool = True, source: bool = False, strings: bool = False, exclude=None)
Help(text, append: bool = False, keep_local: bool = False) → None
Update the help text.
The previous help text has text appended to it, except on the first call. On first call, the values of append and keep_local are considered to determine what is appended to.

Parameters:
- text – string to add to the help text.
- append – on first call, if true, keep the existing help text (default False).
- keep_local – on first call, if true and append is also true, keep only the help text from AddOption calls.

Changed in version 4.6.0: The keep_local parameter was added.
Ignore(target, dependency)
Ignore a dependency.
Import(*vars)
Literal(string)
Local(*targets)
MergeFlags(args, unique: bool = True) → None
Merge flags into construction variables.
Merges the flags from `args` into this construction environment. If `args` is not a dict, it is first converted to one with flags distributed into appropriate construction variables. See `ParseFlags()`.

As a side effect, if `unique` is true, a new object is created for each modified construction variable by the loop at the end. This is silently expected by the `Override()` `parse_flags` functionality, which does not want to share the list (or whatever) with the environment being overridden.

**Parameters:**

- `args` – flags to merge
- `unique` – merge flags rather than appending (default: True). When merging, path variables are retained from the front, other construction variables from the end.

**NoCache (`*targets`)**
Tags a target so that it will not be cached

**NoClean (`*targets`)**
Tags a target so that it will not be cleaned by `-c`

**Override (`overrides`)**
 Produce a modified environment whose variables are overridden by the overrides dictionaries. “overrides” is a dictionary that will override the variables of this environment.
This function is much more efficient than `Clone()` or creating a new `Environment` because it doesn’t copy the construction environment dictionary, it just wraps the underlying construction environment, and doesn’t even create a wrapper object if there are no overrides.

**ParseConfig (`command`, `function`=None, `unique`: `bool` = True)**
Parse the result of running a command to update construction vars.
Use `function` to parse the output of running `command` in order to modify the current environment.

**Parameters:**

- `command` – a string or a list of strings representing a command and its arguments.
- `function` – called to process the result of `command`, which will be passed as `args`. If `function` is omitted or `None`, `MergeFlags()` is used. Takes 3 args `(env, args, unique)`
- `unique` – whether no duplicate values are allowed (default true)

**ParseDepends (`filename`, `must_exist`=None, `only_one`: `bool` = False)**
Parse a mkdep-style file for explicit dependencies. This is completely abusable, and should be unnecessary in the “normal” case of proper SCons configuration, but it may help make the transition from a Make hierarchy easier for some people to swallow. It can also be genuinely useful when using a tool that can write a .d file, but for which writing a scanner would be too complicated.

**ParseFlags (`*flags`) → dict**
Return a dict of parsed flags.
Parse `flags` and return a dict with the flags distributed into the appropriate construction variable names. The flags are treated as a typical set of command-line flags for a GNU-style toolchain, such as might have been generated by one of the (foo)-config scripts, and used to populate the entries based on knowledge embedded in this method - the choices are not expected to be portable to other toolchains.
If one of the `flags` strings begins with a bang (exclamation mark), it is assumed to be a command and the rest of the string is executed; the result of that evaluation is then added to the dict.

**Platform (`platform`)**

**Precious (`*targets`)**
Mark targets as precious: do not delete before building.

**Prepend (`**kw`) → None**
Prepend values to construction variables in an Environment.
The variable is created if it is not already present.

**PrependENVPath (`name`, `newpath`, `envname`: `str` = `ENV`, `sep`: `str` = `:`, `delete_existing`: `bool` = True) → None**
Prepend path elements to the path `name` in the `envname` dictionary for this environment. Will only add any particular path once, and will normpath and normcase all paths to help assure this. This can also handle the case where the env variable is a list instead of a string.
If `delete_existing` is False, a `newpath` component already in the path will not be moved to the front (it will be left where it is).
PrependUnique (delete_existing: bool = False, **kw) → None
Prepend values to existing construction variables in an Environment, if they’re not already there. If delete_existing is True, removes existing values first, so values move to front.

Pseudo (*targets)
Mark targets as pseudo: must not exist.

PyPackageDir (modulename)

RemoveMethod (function) → None
Removes the specified function’s MethodWrapper from the added_methods list, so we don’t re-bind it when making a clone.

Replace (**kw) → None
Replace existing construction variables in an Environment with new construction variables and/or values.

ReplaceIxes (path, old_prefix, old_suffix, new_prefix, new_suffix)
Replace old_prefix with new_prefix and old_suffix with new_suffix.

env - Environment used to interpolate variables. path - the path that will be modified. old_prefix - construction variable for the old prefix. old_suffix - construction variable for the old suffix. new_prefix - construction variable for the new prefix. new_suffix - construction variable for the new suffix.

Repository (*dirs, **kw) → None
Specify Repository directories to search.

Requires (target, prerequisite)
Specify that prerequisite must be built before target. Creates an order-only relationship, not a full dependency. prerequisite must exist before target can be built, but a change to prerequisite does not trigger a rebuild of target.

SConscript (*ls, **kw)
Execute SCons configuration files.

Parameters: *ls (str or list) – configuration file(s) to execute.
Keyword Arguments:

• dirs (list) – execute SConscript in each listed directory.
• name (str) – execute script ‘name’ (used only with ‘dirs’).
• exports (list or dict) – locally export variables the called script(s) can import.
• variant_dir (str) – mirror sources needed for the build in a variant directory to allow building in it.
• duplicate (bool) – physically duplicate sources instead of just adjusting paths of derived files (used only with ‘variant_dir’) (default is True).
• must_exist (bool) – fail if a requested script is missing (default is False, default is deprecated).

Returns: list of variables returned by the called script
Raises: UserError – a script is not found and such exceptions are enabled.

static SConscriptChdir (flag: bool) → None
SConsignFile (name=’.sconsign’, dbm_module=None) → None
Scanner (*args, **kw)
SetDefault (**kw) → None
SetOption (name, value) → None
SideEffect (side_effect, target)
Tell scons that side_effects are built as side effects of building targets.

Split (arg)
This function converts a string or list into a list of strings or Nodes. This makes things easier for users by allowing files to be specified as a white-space separated list to be split.

The input rules are:

• A single string containing names separated by spaces. These will be split apart at the spaces.
• A single Node instance
**A list containing either strings or Node instances. Any strings in the list are not split at spaces.**

In all cases, the function returns a list of Nodes and strings.

**Tool** (*tool*, *toolpath=None*, **kwargs) → Tool

Find and run tool module *tool*.

Changed in version 4.2: returns the tool module rather than *None*.

**Value** (*value*, *built_value=None*, *name=None*)

Return a Value (Python expression) node.

Changed in version 4.0: the *name* parameter was added.

**VariantDir** (*variant_dir*, *src_dir*, *duplicate: int = 1*) → None

WhereIs (**prog**, *path=None*, *pathext=None*, *reject=None*)

Find prog in the path.

_canonicalize_ (*path*)

Allow Dirs and strings beginning with # for top-relative.

Note this uses the current env’s fs (in self).

_changed_build_ (*dependency*, *target*, *prev_ni*, *repo_node=None*) → bool

_changed_content_ (*dependency*, *target*, *prev_ni*, *repo_node=None*) → bool

_changed_timestamp_match_ (*dependency*, *target*, *prev_ni*, *repo_node=None*) → bool

_changed_timestamp_newer_ (*dependency*, *target*, *prev_ni*, *repo_node=None*) → bool

_changed_timestamp_then_content_ (*dependency*, *target*, *prev_ni*, *repo_node=None*) → bool

_find_toolpath_dir_ (*tp*)

_get_SConscript_filenames_ (*ls*, *kw*)

Convert the parameters passed to SConscript() calls into a list of files and export variables. If the parameters are invalid, throws SCons.Errors.UserError. Returns a tuple (l, e) where l is a list of SConscript filenames and e is a list of exports.

static _get_major_minor_revision_ (*version_string*)

Split a version string into major, minor and (optionally) revision parts.

This is complicated by the fact that a version string can be something like 3.2b1.

_gsm_ ()

_init_special_ () → None

Initial the dispatch tables for special handling of special construction variables.

_update_ (*other*) → None

Private method to update an environment’s consvar dict directly.

Bypasses the normal checks that occur when users try to set items.

_update_onlynew_ (*other*) → None

Private method to add new items to an environment’s consvar dict.

Only adds items from *other* whose keys do not already appear in the existing dict; values from *other* are not used for replacement. Bypasses the normal checks that occur when users try to set items.

arg2nodes (*args*, *node_factory=<class 'SCons.Environment._Null'>*, *lookup_list=<class 'SCons.Environment._Null'>*, **kw*)

Converts *args* to a list of nodes.

**Parameters:**

- **just** (*args - filename strings or nodes to convert; nodes are*) – added to the list without further processing.
- **not** (*node_factory - optional factory to create the nodes; if*) – specified, will use this environment’s `fs.File` method.
- **to** (*lookup_list - optional list of lookup functions to call*) – attempt to find the file referenced by each *args*.
- **add.** (*kw - keyword arguments that represent additional nodes to*) –

_backtick_ (*command*) → str

Emulate command substitution.

Provides behavior conceptually like POSIX Shell notation for running a command in backquotes (backticks) by running *command* and returning the resulting output string.

This is not really a public API any longer, it is provided for the use of ParseFlags() (which supports it using a syntax of !command) and ParseConfig().
### Raises: **OSError** – if the external command returned non-zero exit status.

- `get (key, default=None)`
  - Emulates the `get()` method of dictionaries.
- `get_CacheDir ()`  
  - Fetch the builder with the specified name from the environment.
- `get_builder (name)`  
  - Return a factory function for creating Nodes for this construction environment.
- `get_scanner (key)`  
  - Find the appropriate scanner given a key (usually a file suffix).
- `gvars ()`  
  - Emulates the `items()` method of dictionaries.
- `keys ()`  
  - Emulates the `keys()` method of dictionaries.
- `lvars ()`  
  - Emulates the `values()` method of dictionaries.
- `scanner_map_delete (kw=None) -> None`  
  - Delete the cached scanner map (if we need to).
- `setdefault (key, default=None)`  
  - Emulates the `setdefault()` method of dictionaries.

#### `subst (string, raw: int = 0, target=None, source=None, conv=None, executor: Executor | None = None, overrides: dict | None = None)`

Recursively interpolates construction variables from the Environment into the specified string, returning the expanded result. Construction variables are specified by a $ prefix in the string and begin with an initial underscore or alphabetic character followed by any number of underscores or alphanumeric characters. The construction variable names may be surrounded by curly braces to separate the name from trailing characters.

#### `subst_kw (kw, raw: int = 0, target=None, source=None)`

Calls through to `SCons.Subst.scons_subst_list()`.

#### `subst_path (path, target=None, source=None)`

Substitute a path list.

#### `validate_CacheDir_class (custom_class=None)`

Validate the passed custom CacheDir class, or if no args are passed, validate the custom CacheDir class from the environment.
SCons API Documentation

SCons.Script.SConscript._SConscript (fs, *files, **kw)
SCons.Script.SConscript.annotate (node)
   Annotate a node with the stack frame describing the SConscript file and line number that created it.
SCons.Script.SConscript.compute_exports (exports)
   Compute a dictionary of exports given one of the parameters to the Export() function or the exports argument to SConscript().
SCons.Script.SConscript.get_DefaultEnvironmentProxy ()
SCons.Script.SConscript.get_calling_namespaces ()
   Return the locals and globals for the function that called into this module in the current call stack.
SCons.Script.SConscript.handle_missing_SConscript (f: str, must_exist: bool = True) → None
   Take appropriate action on missing file in SConscript() call.
   Print a warning or raise an exception on missing file, unless missing is explicitly allowed by the must_exist parameter or by a global flag.

   Parameters:
   • f – path to missing configuration file
   • must_exist – if true (the default), fail. If false do nothing, allowing a build to declare it’s okay to be missing.

   Raises: UserError – if must_exist is true or if global SCons.Script._no_missing_sconscript is true.

Module contents

The main() function used by the scons script.

Architecturally, this is the scons script, and will likely only be called from the external “scons” wrapper. Consequently, anything here should not be, or be considered, part of the build engine. If it’s something that we expect other software to want to use, it should go in some other module. If it’s specific to the “scons” script invocation, it goes here.
SCons.Script.HelpFunction (text, append: bool = False, keep_local: bool = False) → None
   The implementation of the Help method.
   See Help().
   Changed in version 4.6.0: The keep_local parameter was added.

class SCons.Script.TargetList (initlist=None)
   Bases: UserList
   _abc_impl = <abc._abc_data object>
   _add_Default (list) → None
   _clear () → None
   _do_nothing (*args, **kw) → None
   append (item)
      S.append(value) – append value to the end of the sequence
   clear () → None -- remove all items from S
   copy ()
   count (value) → integer -- return number of occurrences of value
   extend (other)
      S.extend(iterable) – extend sequence by appending elements from the iterable
   index (value[, start[, stop]]) → integer -- return first index of value.
      Raises ValueError if the value is not present.
   insert (i, item)
      S.insert(index, value) – insert value before index
   pop ([, index]) → item -- remove and return item at index (default last).
      Raise IndexError if list is empty or index is out of range.
   remove (item)
      S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.
   reverse ()
      S.reverse() – reverse IN PLACE
   sort (*args, **kwds)
SCons.Script.Variables (files=None, args={})
**SCons API Documentation**

- `SCons.Script._Add_Arguments(alist) → None`
- `SCons.Script._Add_Targets(tlist) → None`
- `SCons.Script._Get_Default_Targets(d, fs)`
- `SCons.Script._Set_Default_Targets(env, tlist) → None`
- `SCons.Script._Set_Default_Targets_Has_Been_Called(d, fs)`
- `SCons.Script._Set_Default_Targets_Has_Not_Been_Called(d, fs)`
- `SCons.Script.set_missing_sconscript_error(flag: bool = True) → bool`

**Returns:** previous value

### SCons.Taskmaster package

**Submodules**

#### SCons.Taskmaster.Job module

Serial and Parallel classes to execute build tasks.

**The Jobs class provides a higher level interface to start, stop, and wait on jobs.**

**class SCons.Taskmaster.Job.InterruptState**

**Bases:** object

**set () → None**


**Bases:** object

An instance of this class initializes N jobs, and provides methods for starting, stopping, and waiting on all N jobs.

- `_reset_sig_handler () → None`
- `_setup_sig_handler () → None`

Setup an interrupt handler so that SCons can shutdown cleanly in various conditions:

- **a. SIGINT:** Keyboard interrupt
- **b. SIGTERM:** Kill or system shutdown
- **c. SIGHUP:** Controlling shell exiting

We handle all of these cases by stopping the taskmaster. It turns out that it’s very difficult to stop the build process by throwing asynchronously an exception such as KeyboardInterrupt. For example, the python Condition variables (threading.Condition) and queues do not seem to be asynchronous-exception-safe. It would require adding a whole bunch of try/finally block and except KeyboardInterruption all over the place.

Note also that we have to be careful to handle the case when SCons forks before executing another process. In that case, we want the child to exit immediately.

**run (postfunc=<function Jobs.<lambda>>) → None**

Run the jobs.

postfunc() will be invoked after the jobs has run. It will be invoked even if the jobs are interrupted by a keyboard interrupt (well, in fact by a signal such as either SIGINT, SIGTERM or SIGHUP). The execution of postfunc() is protected against keyboard interrupts and is guaranteed to run to completion.

**were_interrupted ()**

Returns whether the jobs were interrupted by a signal.

**class SCons.Taskmaster.Job.LegacyParallel (taskmaster, num, stack_size)**

**Bases:** object

This class is used to execute tasks in parallel, and is somewhat less efficient than Serial, but is appropriate for parallel builds.

This class is thread safe.

**start ()**

Start the job. This will begin pulling tasks from the taskmaster and executing them, and return when there are no more tasks. If a task fails to execute (i.e. execute() raises an exception), then the job will stop.

**class SCons.Taskmaster.Job.NewParallel (taskmaster, num, stack_size)**
class FakeCondition (lock)
    Bases: object
    notify ()
    notify_all ()
    wait ()

class FakeLock
    Bases: object
    lock ()
    unlock ()

class State (value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)
    Bases: Enum
    COMPLETED = 3
    READY = 0
    SEARCHING = 1
    STALLED = 2
    __contains__ (member)
        Return True if member is a member of this enum raises TypeError if member is not an enum member
        note: in 3.12 TypeError will no longer be raised, and True will also be returned if member is the value of a
        member in this enum
    __getitem__ (name)
        Return the member matching name.
    __iter__ ()
        Return members in definition order.
    __len__ ()
        Return the number of members (no aliases)

class Worker (owner)
    Bases: Thread
    _bootstrap ()
    _bootstrap_inner ()
    _delete ()
        Remove current thread from the dict of currently running threads.
    _initialized = False
    _reset_internal_locks (is_alive)
    _set_ident ()
    _set_native_id ()
    _set_tstate_lock ()
        Set a lock object which will be released by the interpreter when the underlying thread state (see pystate.h) gets
        deleted.
    _stop ()
    _wait_for_tstate_lock (block=True, timeout=-1)
    daemon
        A boolean value indicating whether this thread is a daemon thread.
        This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the
        creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread
        default to daemon = False.
        The entire Python program exits when only daemon threads are left.
    getName ()
        Return a string used for identification purposes only.
        This method is deprecated, use the name attribute instead.
    ident
        Thread identifier of this thread or None if it has not been started.
        This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits
        and another thread is created. The identifier is available even after the thread has exited.
    isDaemon ()
        Return whether this thread is a daemon.
This method is deprecated, use the daemon attribute instead.

**is_alive()**

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. See also the module function enumerate().

**join(timeout=None)**

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates — either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened — if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

**property name**

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

**property native_id**

Native integral thread ID of this thread, or None if it has not been started.

This is a non-negative integer. See the get_native_id() function. This represents the Thread ID as reported by the kernel.

**run() → None**

Method representing the thread’s activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object’s constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

**setDaemon(daemonic)**

Set whether this thread is a daemon.

This method is deprecated, use the .daemon property instead.

**setName(name)**

Set the name string for this thread.

This method is deprecated, use the name attribute instead.

**start()**

Start the thread’s activity.

It must be called at most once per thread object. It arranges for the object’s run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

```python
_adjust_stack_size()
_maybe_start_worker() → None
_restore_stack_size(prev_size) → None
_setup_logging()
_start_worker() → None
_work()
start() → None
trace_message(message) → None
```

**class SCons.Taskmaster.Job.Serial(taskmaster)**

Bases: object

This class is used to execute tasks in series, and is more efficient than Parallel, but is only appropriate for non-parallel builds. Only one instance of this class should be in existence at a time.

This class is not thread safe.

**start()**

Start the job. This will begin pulling tasks from the taskmaster and executing them, and return when there are no more tasks. If a task fails to execute (i.e. execute() raises an exception), then the job will stop.

**class SCons.Taskmaster.Job.ThreadPool(num, stack_size, interrupted)**
Bases: object

This class is responsible for spawning and managing worker threads.

cleanup () → None

Shuts down the thread pool, giving each worker thread a chance to shut down gracefully.

get ()

Remove and return a result tuple from the results queue.

preparation_failed (task) → None

Put task into request queue.

class SCons.Taskmaster.Job.Worker (requestQueue, resultsQueue, interrupted)

Bases: Thread

A worker thread waits on a task to be posted to its request queue, dequeues the task, executes it, and posts a tuple including the task and a boolean indicating whether the task executed successfully.

_bootstrap ()

_bootstrap_inner ()

_delete ()

Remove current thread from the dict of currently running threads.

_initialized = False

_reset_internal_locks (is_alive)

_set_ident ()

_set_native_id ()

_set_tstate_lock ()

Set a lock object which will be released by the interpreter when the underlying thread state (see pystate.h) gets deleted.

_stop ()

_wait_for_tstate_lock (block=True, timeout=-1)

property daemon

A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

getName ()

Return a string used for identification purposes only.

This method is deprecated, use the name attribute instead.

property ident

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isDaemon ()

Return whether this thread is a daemon.

This method is deprecated, use the daemon attribute instead.

is_alive ()

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. See also the module function enumerate().

join (timeout=None)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.
join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

property name
A string used for identification purposes only. It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

property native_id
Native integral thread ID of this thread, or None if it has not been started. This is a non-negative integer. See the get_native_id() function. This represents the Thread ID as reported by the kernel.

run()
Method representing the thread’s activity. You may override this method in a subclass. The standard run() method invokes the callable object passed to the object’s constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

setDaemon(daemonic)
Set whether this thread is a daemon. This method is deprecated, use the .daemon property instead.

setName(name)
Set the name string for this thread. This method is deprecated, use the name attribute instead.

start()
Start the thread’s activity. It must be called at most once per thread object. It arranges for the object’s run() method to be invoked in a separate thread of control. This method will raise a RuntimeError if called more than once on the same thread object.

Module contents

Generic Taskmaster module for the SCons build engine.

This module contains the primary interface(s) between a wrapping user interface and the SCons build engine. There are two key classes here:

**Taskmaster**

This is the main engine for walking the dependency graph and calling things to decide what does or doesn’t need to be built.

**Task**

This is the base class for allowing a wrapping interface to decide what does or doesn’t actually need to be done. The intention is for a wrapping interface to subclass this as appropriate for different types of behavior it may need.

The canonical example is the SCons native Python interface, which has Task subclasses that handle its specific behavior, like printing “foo’ is up to date” when a top-level target doesn’t need to be built, and handling the -c option by removing targets as its “build” action. There is also a separate subclass for suppressing this output when the -q option is used.

The Taskmaster instantiates a Task object for each (set of) target(s) that it decides need to be evaluated and/or built.

```python
class SCons.Taskmaster.AlwaysTask (tm, targets, top, node):
    Bases: Task
    LOGGER = None
    _abc_impl = <abc._abc_data object>
    _exception_raise ()
    Raises a pending exception that was recorded while getting a Task ready for execution.
    _no_exception_to_raise () → None
    display (message) → None
    Hook to allow the calling interface to display a message.
    This hook gets called as part of preparing a task for execution (that is, a Node to be built). As part of figuring out what Node should be built next, the actual target list may be altered, along with a message describing the
```
alteration. The calling interface can subclass Task and provide a concrete implementation of this method to see those messages.

```python
exc_clear () → None
Clears any recorded exception.
This also changes the “exception_raise” attribute to point to the appropriate do-nothing method.
```

```python
exc_info ()
Returns info about a recorded exception.
```

```python
exception_set (exception=None) → None
Records an exception to be raised at the appropriate time.
This also changes the “exception_raise” attribute to point to the method that will, in fact
```

```python
execute ()
Called to execute the task.
This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in prepare(), executed() or failed().
```

```python
executed () → None
Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s callback methods.
This may have been a do-nothing operation (to preserve build order), so we must check the node’s state before deciding whether it was “built”, in which case we call the appropriate Node method. In any event, we always call “visited()”, which will handle any post-visit actions that must take place regardless of whether or not the target was an actual built target or a source Node.
```

```python
executed_with_callbacks () → None
Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s callback methods.
This may have been a do-nothing operation (to preserve build order), so we must check the node’s state before deciding whether it was “built”, in which case we call the appropriate Node method. In any event, we always call “visited()”, which will handle any post-visit actions that must take place regardless of whether or not the target was an actual built target or a source Node.
```

```python
executed_without_callbacks () → None
Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s callback methods.
```

```python
fail_continue () → None
Explicit continue-the-build failure.
This sets failure status on the target nodes and all of their dependent parent nodes.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().
```

```python
fail_stop () → None
Explicit stop-the-build failure.
This sets failure status on the target nodes and all of their dependent parent nodes.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().
```

```python
failed () → None
Default action when a task fails: stop the build.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().
```

```python
get_target ()
Fetch the target being built or updated by this task.
```

```python
make_ready ()
Marks all targets in a task ready for execution if any target is not current.
This is the default behavior for building only what’s necessary.
```

```python
make_ready_all () → None
Marks all targets in a task ready for execution.
This is used when the interface needs every target Node to be visited–the canonical example being the “scons -c” option.
```

```python
make_ready_current ()
Marks all targets in a task ready for execution if any target is not current.
This is the default behavior for building only what's necessary.

needs_execute () → bool

Always returns True (indicating this Task should always be executed).

Subclasses that need this behavior (as opposed to the default of only executing Nodes that are out of date w.r.t. their dependencies) can use this as follows:

```python
class MyTaskSubclass(SCons.Taskmaster.Task):
    needs_execute = SCons.Taskmaster.AlwaysTask.needs_execute
```

postprocess () → None

Post-processes a task after it's been executed.

This examines all the targets just built (or not, we don't care if the build was successful, or even if there was no build because everything was up-to-date) to see if they have any waiting parent Nodes, or Nodes waiting on a common side effect, that can be put back on the candidates list.

prepare () → None

Called just before the task is executed.

This is mainly intended to give the target Nodes a chance to unlink underlying files and make all necessary directories before the Action is actually called to build the targets.

trace_message (node, description: str = 'node') → None

Hook to allow the calling interface to display a message.

This hook gets called as part of preparing a task for execution (that is, a Node to be built). As part of figuring out what Node should be built next, the actual target list may be altered, along with a message describing the alteration. The calling interface can subclass Task and provide a concrete implementation of this method to see those messages.

exc_clear () → None

Clears any recorded exception.

This also changes the “exception_raise” attribute to point to the appropriate do-nothing method.

exc_info () → None

Returns info about a recorded exception.

exception_set (exception=None) → None

Records an exception to be raised at the appropriate time.

This also changes the “exception_raise” attribute to point to the method that will, in fact

done: execute

Called to execute the task.

This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in prepare(), executed() or failed().

executed () → None

Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s callback methods.

This may have been a do-nothing operation (to preserve build order), so we must check the node’s state before deciding whether it was “built”, in which case we call the appropriate Node method. In any event, we always call “visited()”, which will handle any post-visit actions that must take place regardless of whether or not the target was an actual built target or a source Node.

executed_with_callbacks () → None

Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s callback methods.

This may have been a do-nothing operation (to preserve build order), so we must check the node’s state before deciding whether it was “built”, in which case we call the appropriate Node method. In any event, we always call
“visited()”, which will handle any post-visit actions that must take place regardless of whether or not the target was an actual built target or a source Node.

executed_without_callbacks () → None
Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s callback methods.

fail_continue () → None
Explicit continue-the-build failure.
This sets failure status on the target nodes and all of their dependent parent nodes.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

fail_stop () → None
Explicit stop-the-build failure.
This sets failure status on the target nodes and all of their dependent parent nodes.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

failed () → None
Default action when a task fails: stop the build.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

get_target ()
Fetch the target being built or updated by this task.

make_ready ()
Marks all targets in a task ready for execution if any target is not current.
This is the default behavior for building only what’s necessary.

make_ready_all () → None
Marks all targets in a task ready for execution.
This is used when the interface needs every target Node to be visited—the canonical example being the “scons -c” option.

make_ready_current ()
Marks all targets in a task ready for execution if any target is not current.
This is the default behavior for building only what’s necessary.

needs_execute ()
Returns True (indicating this Task should be executed) if this Task’s target state indicates it needs executing, which has already been determined by an earlier up-to-date check.

postprocess () → None
Post-processes a task after it’s been executed.
This examines all the targets just built (or not, we don’t care if the build was successful, or even if there was no build because everything was up-to-date) to see if they have any waiting parent Nodes, or Nodes waiting on a common side effect, that can be put back on the candidates list.

prepare () → None
Called just before the task is executed.
This is mainly intended to give the target Nodes a chance to unlink underlying files and make all necessary directories before the Action is actually called to build the targets.

trace_message (node, description: str = ‘node’) → None

class SCons.Taskmaster.Stats
Bases: object
A simple class for holding statistics about the disposition of a Node by the Taskmaster. If we’re collecting statistics, each Node processed by the Taskmaster gets one of these attached, in which case the Taskmaster records its decision each time it processes the Node. (Ideally, that’s just once per Node.)

class SCons.Taskmaster.Task (tm, targets, top, node)
Bases: ABC
SCons build engine abstract task class.
This controls the interaction of the actual building of node and the rest of the engine.
This is expected to handle all of the normally-customizable aspects of controlling a build, so any given application should be able to do what it wants by sub-classing this class and overriding methods as appropriate. If an application
needs to customize something by sub-classing Taskmaster (or some other build engine class), we should first try to
migrate that functionality into this class. Note that it’s generally a good idea for sub-classes to call these methods explicitly to update state, etc., rather than roll their own interaction with Taskmaster from scratch.

LOGGER = None

_exception_raise()

Raises a pending exception that was recorded while getting a Task ready for execution.

_no_exception_to_raise() → None
display(message) → None

Hook to allow the calling interface to display a message.
This hook gets called as part of preparing a task for execution (that is, a Node to be built). As part of figuring out
what Node should be built next, the actual target list may be altered, along with a message describing the
alteration. The calling interface can subclass Task and provide a concrete implementation of this method to see
those messages.

exc_clear() → None

Clears any recorded exception.
This also changes the “exception_raise” attribute to point to the appropriate do-nothing method.

exc_info() → None

Returns info about a recorded exception.

exception_set(exception=None) → None

Records an exception to be raised at the appropriate time.
This also changes the “exception_raise” attribute to point to the method that will, in fact

execute()

Called to execute the task.
This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe
stuff in prepare(), executed() or failed().

executed() → None

Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s
callback methods.

This may have been a do-nothing operation (to preserve build order), so we must check the node’s state before
deciding whether it was “built”, in which case we call the appropriate Node method. In any event, we always call
“visited()”, which will handle any post-visit actions that must take place regardless of whether or not the target was
an actual built target or a source Node.

executed_with_callbacks() → None

Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s
callback methods.

This may have been a do-nothing operation (to preserve build order), so we must check the node’s state before
deciding whether it was “built”, in which case we call the appropriate Node method. In any event, we always call
“visited()”, which will handle any post-visit actions that must take place regardless of whether or not the target was
an actual built target or a source Node.

executed_without_callbacks() → None

Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s
callback methods.

fail_continue() → None

Explicit continue-the-build failure.
This sets failure status on the target nodes and all of their dependent parent nodes.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on
up-to-date nodes when using Configure().

fail_stop() → None

Explicit stop-the-build failure.
This sets failure status on the target nodes and all of their dependent parent nodes.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on
up-to-date nodes when using Configure().

failed() → None

Default action when a task fails: stop the build.
get_target ()
Fetch the target being built or updated by this task.

make_ready ()
Marks all targets in a task ready for execution if any target is not current.
This is the default behavior for building only what’s necessary.

make_ready_all () → None
Marks all targets in a task ready for execution.
This is used when the interface needs every target Node to be visited—the canonical example being the “scons -c” option.

make_ready_current ()
Marks all targets in a task ready for execution if any target is not current.
This is the default behavior for building only what’s necessary.

abstract needs_execute ()

postprocess () → None
Post-processes a task after it’s been executed.
This examines all the targets just built (or not, we don’t care if the build was successful, or even if there was no build because everything was up-to-date) to see if they have any waiting parent Nodes, or Nodes waiting on a common side effect, that can be put back on the candidates list.

prepare () → None
Called just before the task is executed.
This is mainly intended to give the target Nodes a chance to unlink underlying files and make all necessary directories before the Action is actually called to build the targets.

trace_message (node, description: str = 'node') → None

class SCons.Taskmaster.Taskmaster (targets=[], tasker=None, order=None, trace=None)
Bases: object
The Taskmaster for walking the dependency DAG.

_find_next_ready_node ()
Finds the next node that is ready to be built.
This is the main guts of the DAG walk. We loop through the list of candidates, looking for something that has no un-built children (i.e., that is a leaf Node or has dependencies that are all leaf Nodes or up-to-date). Candidate Nodes are re-scanned (both the target Node itself and its sources, which are always scanned in the context of a given target) to discover implicit dependencies. A Node that must wait for some children to be built will be put back on the candidates list after the children have finished building. A Node that has been put back on the candidates list in this way may have itself (or its sources) re-scanned, in order to handle generated header files (e.g.) and the implicit dependencies therein.

Note that this method does not do any signature calculation or up-to-date check itself. All of that is handled by the Task class. This is purely concerned with the dependency graph walk.

_validate_pending_children () → None
Validate the content of the pending_children set. Assert if an internal error is found.
This function is used strictly for debugging the taskmaster by checking that no invariants are violated. It is not used in normal operation.
The pending_children set is used to detect cycles in the dependency graph. We call a “pending child” a child that is found in the “pending” state when checking the dependencies of its parent node.
A pending child can occur when the Taskmaster completes a loop through a cycle. For example, let’s imagine a graph made of three nodes (A, B and C) making a cycle. The evaluation starts at node A. The Taskmaster first considers whether node A’s child B is up-to-date. Then, recursively, node B needs to check whether node C is up-to-date. This leaves us with a dependency graph looking like:

```
           Next candidate
           ^                 |
            |                +-------------------------------------+
```
Now, when the Taskmaster examines the Node C’s child Node A, it finds that Node A is in the “pending” state. Therefore, Node A is a pending child of node C.

Pending children indicate that the Taskmaster has potentially loop back through a cycle. We say potentially because it could also occur when a DAG is evaluated in parallel. For example, consider the following graph:

```
Node A (Pending) --> Node B(Pending) --> Node C (Pending) --> ...
             ^
            |   +----------> Node D (NoState) --------+
               /                               /
Next candidate /
```

The Taskmaster first evaluates the nodes A, B, and C and starts building some children of node C. Assuming, that the maximum parallel level has not been reached, the Taskmaster will examine Node D. It will find that Node C is a pending child of Node D.

In summary, evaluating a graph with a cycle will always involve a pending child at one point. A pending child might indicate either a cycle or a diamond-shaped DAG. Only a fraction of the nodes ends-up being a “pending child” of another node. This keeps the pending_children set small in practice.

We can differentiate between the two cases if we wait until the end of the build. At this point, all the pending children nodes due to a diamond-shaped DAG will have been properly built (or will have failed to build). But, the pending children involved in a cycle will still be in the pending state.

The taskmaster removes nodes from the pending_children set as soon as a pending_children node moves out of the pending state. This also helps to keep the pending_children set small.

```
cleanup ()
    Check for dependency cycles.
configure_trace (trace=None) → None
    This handles the command line option --taskmastertrace= It can be: - : output to stdout <filename> : output to a file False/None : Do not trace
find_next_candidate ()
    Returns the next candidate Node for (potential) evaluation.
    The candidate list (really a stack) initially consists of all of the top-level (command line) targets provided when the Taskmaster was initialized. While we walk the DAG, visiting Nodes, all the children that haven’t finished processing get pushed on to the candidate list. Each child can then be popped and examined in turn for whether their children are all up-to-date, in which case a Task will be created for their actual evaluation and potential building.
    Here is where we also allow candidate Nodes to alter the list of Nodes that should be examined. This is used, for example, when invoking SCons in a source directory. A source directory Node can return its corresponding build directory Node, essentially saying, “Hey, you really need to build this thing over here instead.”
next_task ()
    Returns the next task to be executed.
    This simply asks for the next Node to be evaluated, and then wraps it in the specific Task subclass with which we were initialized.
no_next_candidate ()
    Stops Taskmaster processing by not returning a next candidate.
    Note that we have to clean-up the Taskmaster candidate list because the cycle detection depends on the fact all nodes have been processed somehow.
stop () → None
    Stops the current build completely.
```
SCons Tool package

Module contents

SCons tool selection.

Looks for modules that define a callable object that can modify a construction environment as appropriate for a given tool (or tool chain).

Note that because this subsystem just selects a callable that can modify a construction environment, it’s possible for people to define their own “tool specification” in an arbitrary callable function. No one needs to use or tie in to this subsystem in order to roll their own tool specifications.

SCons.Tool.CreateJarBuilder (env)

The Jar builder expects a list of class files which it can package into a jar file.

The jar tool provides an interface for passing other types of java files such as .java, directories or swig interfaces and will build them to class files in which it can package into the jar.

SCons.Tool.CreateJavaClassDirBuilder (env)
SCons.Tool.CreateJavaClassFileBuilder (env)
SCons.Tool.CreateJavaFileBuilder (env)
SCons.Tool.CreateJavaHBuilder (env)
SCons.Tool.FindAllTools (tools, env)
SCons.Tool.FindTool (tools, env)
SCons.Tool.Initializers (env) → None

class SCons.Tool.Tool (name, toolpath=None, **kwargs)

Bases: object

_tool_module ()

Try to load a tool module.

This will hunt in the toolpath for both a Python file (toolname.py) and a Python module (toolname directory), then try the regular import machinery, then fallback to try a zipfile.

class SCons.Tool.ToolInitializer (env, tools, names)

Bases: object

A class for delayed initialization of Tools modules.

Instances of this class associate a list of Tool modules with a list of Builder method names that will be added by those Tool modules. As part of instantiating this object for a particular construction environment, we also add the appropriate ToolInitializerMethod objects for the various Builder methods that we want to use to delay Tool searches until necessary.

apply_tools (env) → None

Searches the list of associated Tool modules for one that exists, and applies that to the construction environment.

remove_methods (env) → None

Removes the methods that were added by the tool initialization so we no longer copy and re-bind them when the construction environment gets cloned.

class SCons.Tool.ToolInitializerMethod (name, initializer)

Bases: object

This is added to a construction environment in place of a method(s) normally called for a Builder (env.Object, env.StaticObject, etc.). When called, it has its associated ToolInitializer object search the specified list of tools and apply the first one that exists to the construction environment. It then calls whatever builder was (presumably) added to the construction environment in place of this particular instance.

__call__ (env, *args, **kw)

get_builder (env)

Returns the appropriate real Builder for this method name after having the associated ToolInitializer object apply the appropriate Tool module.

SCons.Tool.createCFileBuilders (env)

This is a utility function that creates the CFile/CXXFile Builders in an Environment if they are not there already. If they are there already, we return the existing ones.

This is a separate function because soooo many Tools use this functionality.

The return is a 2-tuple of (CFile, CXXFile)

SCons.Tool.createLoadableModuleBuilder (env, loadable_module_suffix: str = '$LDMODULESUFFIX')

This is a utility function that creates the LoadableModule Builder in an Environment if it is not there already.

111
If it is already there, we return the existing one.

**Parameters:** loadable_module_suffix – The suffix specified for the loadable module builder

SCons.Tool.createObjBuilders (env)

This is a utility function that creates the StaticObject and SharedObject Builders in an Environment if they are not there already.

If they are there already, we return the existing ones.

This is a separate function because soooo many Tools use this functionality.

The return is a 2-tuple of (StaticObject, SharedObject)

SCons.Tool.createProgBuilder (env)

This is a utility function that creates the Program Builder in an Environment if it is not there already.

If it is already there, we return the existing one.

SCons.Tool.createSharedLibBuilder (env, shlib_suffix: str = '$_SHLIBSUFFIX')

This is a utility function that creates the SharedLibrary Builder in an Environment if it is not there already.

If it is already there, we return the existing one.

**Parameters:** shlib_suffix – The suffix specified for the shared library builder

SCons.Tool.createStaticLibBuilder (env)

This is a utility function that creates the StaticLibrary Builder in an Environment if it is not there already.

If it is already there, we return the existing one.

SCons.Tool.find_program_path (env, key_program, default_paths=None, add_path: bool = False) → str | None

Find the location of a tool using various means.

Mainly for windows where tools aren’t all installed in /usr/bin, etc.

**Parameters:**

- env – Current Construction Environment.
- key_program – Tool to locate.
- default_paths – List of additional paths this tool might be found in.
- add_path – If true, add path found if it was from default_paths.

SCons.Tool.tool_list (platform, env)

SCons.Util package

**Submodules**

SCons utility functions

This package contains routines for use by other parts of SCons. Candidates for inclusion here are routines that do not need other parts of SCons (other than Util), and have a reasonable chance of being useful in multiple places, rather then being topical only to one module/package.

**class** SCons.Util.CLVar (initlist=None)

**Bases:** UserList

A container for command-line construction variables.

Forces the use of a list of strings intended as command-line arguments. Like collections.UserList, but the argument passed to the initializer will be processed by the Split() function, which includes special handling for string types: they will be split into a list of words, not coerced directly to a list. The same happens if a string is added to a CLVar, which allows doing the right thing with both Append()/Prepend() methods, as well as with pure Python addition, regardless of whether adding a list or a string to a construction variable.

Side effect: spaces will be stripped from individual string arguments. If you need spaces preserved, pass strings containing spaces inside a list argument.

```python
>>> u = UserList("--some --opts and args")
>>> print(len(u), repr(u))
22 ['--', '--', 's', 'o', 'm', 'e', ' ', '--', 'o', 'p', 't', 's', ' ', 'a', 'n', 'd', ' ', 'a', 'r', 'g', 's']
>>> c = CLVar("--some --opts and args")
```
>>> print(len(c), repr(c))
4 ['--some', '--opts', 'and', 'args']
>>> c += " strips spaces "
>>> print(len(c), repr(c))
6 ['--some', '--opts', 'and', 'args', 'strips', 'spaces']
>>> c += [" does not split or strip "]
7 ['--some', '--opts', 'and', 'args', 'strips', 'spaces', ' does not split or strip ']

_abc_impl = <_abc._abc_data object>
append (item)
    S.append(value) – append value to the end of the sequence
clear () → None -- remove all items from S
copy ()
count (value) → integer -- return number of occurrences of value
extend (other)
    S.extend(iterable) – extend sequence by appending elements from the iterable
index (value[, start[, stop]]) → integer -- return first index of value.
    Raises ValueError if the value is not present.
    Supporting start and stop arguments is optional, but recommended.
insert (i, item)
    S.insert(index, value) – insert value before index
pop ([, index]) → item -- remove and return item at index (default last).
    Raise IndexError if list is empty or index is out of range.
remove (item)
    S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.
reverse ()
    S.reverse() – reverse IN PLACE
sort (*args, **kwds)
class SCons.Util.Delegate (attribute)
    Bases: object
    A Python Descriptor class that delegates attribute fetches to an underlying wrapped subject of a Proxy. Typical use:
    class Foo(Proxy):
        __str__ = Delegate('__str__')

class SCons.Util.DispatchingFormatter (formatters, default_formatter)
    Bases: Formatter
    Logging formatter which dispatches to various formatters.
    converter ()
        localtime([seconds]) -> (tm_year,tm_mon,tm_mday,tm_hour,tm_min,
         tm_sec,tm_wday,tm_yday,tm_isdst)
        Convert seconds since the Epoch to a time tuple expressing local time. When ‘seconds’ is not passed in, convert
        the current time instead.
        default_msec_format = "%s,%03d"
        default_time_format = "%Y-%m-%d %H:%M:%S"
        format (record)
        Format the specified record as text.
        The record’s attribute dictionary is used as the operand to a string formatting operation which yields the returned
        string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the
        record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call
to usesTime()), formatTime() is called to format the event time. If there is exception information, it is formatted using
        formatException() and appended to the message.
        formatException (ei)
        Format and return the specified exception information as a string.
This default implementation just uses traceback.print_exception()

formatMessage (record)
formatStack (stack_info)

This method is provided as an extension point for specialized formatting of stack information.
The input data is a string as returned from a call to traceback.print_stack(), but with the last trailing newline
removed.
The base implementation just returns the value passed in.

formatTime (record, datefmt=None)

Return the creation time of the specified LogRecord as formatted text.
This method should be called from format() by a formatter which wants to make use of a formatted time. This
method can be overridden in formatters to provide for any specific requirement, but the basic behaviour is as
follows: if datefmt (a string) is specified, it is used with time.strftime() to format the creation time of the record.
Otherwise, an ISO8601-like (or RFC 3339-like) format is used. The resulting string is returned. This function uses a
user-configurable function to convert the creation time to a tuple. By default, time.localtime() is used; to change this
for a particular formatter instance, set the ‘converter’ attribute to a function with the same signature as
time.localtime() or time.gmtime(). To change it for all formatters, for example if you want all logging times to be
shown in GMT, set the ‘converter’ attribute in the Formatter class.

usesTime ()
Check if the format uses the creation time of the record.

class SCons.Util.DisplayEngine
Bases: object
A callable class used to display SCons messages.
print_it = True
set_mode (mode) → None

SCons.Util.IDX (n) → bool
Generate index into strings from the tree legends.
These are always a choice between two, so bool works fine.

class SCons.Util.LogicalLines (fileobj)
Bases: object
Wrapper class for the logical_lines() function.
Allows us to read all “logical” lines at once from a given file object.

readlines ()

class SCons.Util.NodeList (initlist=None)
Bases: UserList
A list of Nodes with special attribute retrieval.
Unlike an ordinary list, access to a member’s attribute returns a NodeList containing the same attribute for each
member. Although this can hold any object, it is intended for use when processing Nodes, where fetching an attribute
of each member is very common, for example getting the content signature of each node. The term “attribute” here
includes the string representation.

```python
>>> someList = NodeList(['  foo  ', '  bar  '])
>>> someList.strip()
['foo', 'bar']
```

__getattr__ (name) → NodeList
Returns a NodeList of name from each member.
__getitem__ (index)
Returns one item, forces a NodeList if index is a slice.
_abc_impl = <_abc._abc_data object>
append (item)
S.append(value) – append value to the end of the sequence
clear () → None -- remove all items from S
copy ()
count (value) → integer -- return number of occurrences of value
extend (other)
S.extend(iterable) – extend sequence by appending elements from the iterable
index (value[, start[, stop]]) → integer -- return first index of value.
    Raises ValueError if the value is not present.
    Supporting start and stop arguments is optional, but recommended.
insert (i, item)
    S.insert(index, value) – insert value before index
pop ([, index]) → item -- remove and return item at index (default last).
    Raise IndexError if list is empty or index is out of range.
remove (item)
    S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.
reverse ()
    S.reverse() – reverse IN PLACE
sort (*args, **kwds)
class SCons.Util.Proxy (subject)
    Bases: object
    A simple generic Proxy class, forwarding all calls to subject.
    This means you can take an object, let’s call it ‘obj_a’, and wrap it in this Proxy class, with a statement like this:

proxy_obj = Proxy(obj_a)

Then, if in the future, you do something like this:

x = proxy_obj.var1

since the Proxy class does not have a var1 attribute (but presumably obj_a does), the request actually is equivalent to saying:

x = obj_a.var1

Inherit from this class to create a Proxy.
With Python 3.5+ this does not work transparently for Proxy subclasses that use special dunder method names, because those names are now bound to the class, not the individual instances. You now need to know in advance which special method names you want to pass on to the underlying Proxy object, and specifically delegate their calls like this:

class Foo(Proxy):
    __str__ = Delegate('__str__')

    __getattr__ (name)
    Retrieve an attribute from the wrapped object.
    Raises: AttributeError – if attribute name doesn’t exist.

get ()
    Retrieve the entire wrapped object
SCons.Util.RegError
    alias of _NoError
SCons.Util.RegGetValue (root, key)
SCons.Util.RegOpenKeyEx (root, key)
class SCons.Util.Selector
    Bases: dict
    A callable dict for file suffix lookup.
    Often used to associate actions or emitters with file types.
    Depends on insertion order being preserved so that get_suffix() calls always return the first suffix added.
clear () → None. Remove all items from D.
copy () → a shallow copy of D
fromkeys (value=None, /)
Create a new dictionary with keys from iterable and values set to value.

get(key, default=None) →
Return the value for key if key is in the dictionary, else default.

items() →
A set-like object providing a view on D's items

keys() →
A set-like object providing a view on D's keys

pop(k[, d]) → v, remove specified key and return the corresponding value.
If the key is not found, return the default if given; otherwise, raise a KeyError.

popitem() →
Remove and return a (key, value) pair as a 2-tuple.
Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault(key, default=None) →
Insert key with a value of default if key is not in the dictionary.
Return the value for key if key is in the dictionary, else default.

update([, E], **F) →
Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() →
An object providing a view on D's values

SCons.Util.Split(arg) → list
Returns a list of file names or other objects.
If arg is a string, it will be split on whitespace within the string. If arg is already a list, the list will be returned untouched. If arg is any other type of object, it will be returned in a single-item list.

```python
given data:
>>> print(Split(" this is a string "))
['this', 'is', 'a', 'string']

>>> print(Split("stringlist", " preserving ", " spaces "))
['stringlist', ' preserving ', ' spaces ']
```

class SCons.Util.Unbuffered(file)
Bases: object
A proxy that wraps a file object, flushing after every write.
Delegates everything else to the wrapped object.
write(arg) → None
writelines(arg) → None

class SCons.Util.UniqueList(initlist=None)
Bases: UserList
A list which maintains uniqueness.
Uniquing is lazy: rather than being enforced on list changes, it is fixed up on access by those methods which need to act on a unique list to be correct. That means things like membership tests don’t have to eat the uniquing time.

__make_unique() → None
__abc_impl = <_abc._abc_data object>
append(item) → None
S.append(value) – append value to the end of the sequence

clear() → None -- remove all items from S

copy() → None

count(value) → integer -- return number of occurrences of value

extend(other) → None
S.extend(iterable) – extend sequence by appending elements from the iterable

index(value[, start[, stop]]) → integer -- return first index of value.
Raises ValueError if the value is not present.
Supporting start and stop arguments is optional, but recommended.

insert(i, item) → None
S.insert(index, value) – insert value before index

pop([, index]) → item -- remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

remove(item)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

SCons API Documentation
reverse () → None
S.reverse() – reverse IN PLACE
sort (*args, **kwds)
SCons.Util.WhereIs (file, path=None, pathext=None, reject=None) → str | None
Return the path to an executable that matches file.
Searches the given path for file, considering any filename extensions in pathext (on the Windows platform only), and returns the full path to the matching command of the first match, or None if there are no matches. Will not select any path name or names in the optional reject list.
If path is None (the default), os.environ[PATH] is used. On Windows, If pathext is None (the default), os.environ[PATHEXT] is used.
The construction environment method of the same name wraps a call to this function by filling in path from the execution environment if it is None (and for pathext on Windows, if necessary), so if called from there, this function will not backfill from os.environ.

Note
Finding things in os.environ may answer the question “does file exist on the system”, but not the question “can SCons use that executable”, unless the path element that yields the match is also in the the Execution Environment (e.g. env['ENV']['PATH']). Since this utility function has no environment reference, it cannot make that determination.

exception SCons.Util._NoError
Bases: Exception
add_note ()
Exception.add_note(note) – add a note to the exception
args
with_traceback ()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.
SCons.Util._semi_deepcopy_list (obj) → list
SCons.Util._semi_deepcopy_tuple (obj) → tuple
SCons.Util.adjustixes (fname, pre, suf, ensure_suffix: bool = False) → str
Adjust filename prefixes and suffixes as needed.
Add prefix to fname if specified. Add suffix to fname if specified and if ensure_suffix is True
SCons.Util.case_sensitive_suffixes (s1: str, s2: str) → bool
Returns whether platform distinguishes case in file suffixes.
SCons.Util.cmp (a, b) → bool
A cmp function because one is no longer available in Python3.
SCons.Util.containsAll (s, pat) → bool
Check whether string s contains ALL of the items in pat.
SCons.Util.containsAny (s, pat) → bool
Check whether string s contains ANY of the items in pat.
SCons.Util.containsOnly (s, pat) → bool
Check whether string s contains ONLY items in pat.
SCons.Util.dictify (keys, values, result=None) → dict
SCons.Util.do_flatten (sequence, result, instance=<built-in function isinstance>, StringTypes=(<class 'str'>, <class 'collections.UserString'>), SequenceTypes=(<class 'list'>, <class 'tuple'>, <class 'collections.deque'>, <class 'collections.UserList'>, <class 'collections.abc.MappingView'>)) → None
SCons.Util.flatten (obj, instance=<built-in function isinstance>, StringTypes=(<class 'str'>, <class 'collections.UserString'>), SequenceTypes=(<class 'list'>, <class 'tuple'>, <class 'collections.deque'>, <class 'collections.UserList'>, <class 'collections.abc.MappingView'>), do_flatten=<function do_flatten>) → list
Flatten a sequence to a non-nested list.
Converts either a single scalar or a nested sequence to a non-nested list. Note that flatten() considers strings to be scalars instead of sequences like pure Python would.

SCons.Util.flatten_sequence(sequence, isinstance=<built-in function isinstance>, StringTypes=(<class 'str'>, <class 'collections.UserString'>), SequenceTypes=(<class 'list'>, <class 'tuple'>, <class 'collections.deque'>, <class 'collections.UserList'>, <class 'collections.abc.MappingView'>), do_flatten=<function do_flatten>) → list

Flatten a sequence to a non-nested list.

SCons.Util.get_native_path(path: str) → str

Transform an absolute path into a native path for the system.
In Cygwin, this converts from a Cygwin path to a Windows path, without regard to whether path refers to an existing file system object. For other platforms, path is unchanged.

SCons.Util.logical_lines(physical_lines, joiner=<built-in method join of str object>)

SCons.Util.make_path_relative(path) → str

Converts an absolute path name to a relative pathname.

SCons.Util.print_time()

Hack to return a value from Main if can't import Main.

SCons.Util.print_tree(root, child_func, prune: bool = False, showtags: int = 0, margin: List[bool] = [False], visited: dict | None = None, lastChild: bool = False, singleLineDraw: bool = False) → None

Print a tree of nodes.
This is like func:render_tree, except it prints lines directly instead of creating a string representation in memory, so that huge trees can be handled.

Parameters:
- root – the root node of the tree
- child_func – the function called to get the children of a node
- prune – don’t visit the same node twice
- showtags – print status information to the left of each node line The default is false (value 0). A value of 2 will also print a legend for the margin tags.
- margin – the format of the left margin to use for children of root. Each entry represents a column, where a true value will display a vertical bar and a false one a blank.
- visited – a dictionary of visited nodes in the current branch if prune is false, or in the whole tree if prune is true.
- lastChild – this is the last leaf of a branch
- singleLineDraw – use line-drawing characters rather than ASCII.

SCons.Util.render_tree(root, child_func, prune: bool = False, margin: List[bool] = [False], visited: dict | None = None) → str

Render a tree of nodes into an ASCII tree view.

Parameters:
- root – the root node of the tree
- child_func – the function called to get the children of a node
- prune – don’t visit the same node twice
- margin – the format of the left margin to use for children of root. Each entry represents a column where a true value will display a vertical bar and a false one a blank.
- visited – a dictionary of visited nodes in the current branch if prune is false, or in the whole tree if prune is true.

SCons.Util.rightmost_separator(path, sep)

SCons.Util.sanitize_shell_env(execution_env: dict) → dict
Sanitize all values in execution_env
The execution environment (typically comes from env['ENV']) is propagated to the shell, and may need to be cleaned first.

**Parameters:**
- **execution_env** – The shell environment variables to be propagated
- **shell. (to the spawned)** –

**Returns:** sanitized dictionary of env variables (similar to what you’d get from os.environ)

SCons.Util.semi_deepcopy(obj)
SCons.Util.semi_deepcopy_dict(obj, exclude=None) → dict
SCons.Util.silent_intern(__string: Any) → str
Intern a string without failing.
Perform sys.intern on the passed argument and return the result. If the input is ineligible for interning the original argument is returned and no exception is thrown.

SCons.Util.splitext(path) → tuple
Split path into a (root, ext) pair.
Same as os.path.splitext but faster.

SCons.Util.unique(seq)
Return a list of the elements in seq without duplicates, ignoring order.
For best speed, all sequence elements should be hashable. Then unique() will usually work in linear time.
If not possible, the sequence elements should enjoy a total ordering, and if list(s).sort() doesn’t raise TypeError it is assumed that they do enjoy a total ordering. Then unique() will usually work in O(N*log2(N)) time.
If that’s not possible either, the sequence elements must support equality-testing. Then unique() will usually work in quadratic time.

```python
>>> mylist = unique([1, 2, 3, 1, 2, 3])
>>> print(sorted(mylist))
[1, 2, 3]
>>> mylist = unique("abcabc")
>>> print(sorted(mylist))
['a', 'b', 'c']
>>> mylist = unique(((1, 2), (2, 3), (1, 2)))
>>> print(sorted(mylist))
[(1, 2), (2, 3)]
```

SCons.Util.uniquer_hashables(seq)
SCons.Util.updrive(path) → str
Make the drive letter (if any) upper case.
This is useful because Windows is inconsistent on the case of the drive letter, which can cause inconsistencies when calculating command signatures.

SCons.Util.wait_for_process_to_die(pid) → None
Wait for specified process to die, or alternatively kill it NOTE: This function operates best with psutil pypi package
TODO: Add timeout which raises exception

SCons environment utility functions.

Routines for working with environments and construction variables that don’t need the specifics of the Environment class.

SCons.Util.envs.AddMethod(obj, function: Callable, name: str | None = None) → None
Add a method to an object.
Adds function to obj if obj is a class object. Adds function as a bound method if obj is an instance object. If obj looks like an environment instance, use MethodWrapper to add it. If name is supplied it is used as the name of function.
Although this works for any class object, the intent as a public API is to be used on Environment, to be able to add a method to all construction environments; it is preferred to use env.AddMethod to add to an individual environment.

```python
>>> class A:
...     ...
```
>>> a = A()

>>> def f(self, x, y):
...    self.z = x + y

>>> AddMethod(A, f, "add")
>>> a.add(2, 4)
>>> print(a.z)
6

>>> a.data = ['a', 'b', 'c', 'd', 'e', 'f']
>>> AddMethod(a, lambda self, i: self.data[i], "listIndex")
>>> print(a.listIndex(3))
\d

SCons.Util.envs.AddPathIfNotExists (env_dict, key, path, sep: str = ':') → None
Add a path element to a construction variable.
key is looked up in env_dict, and path is added to it if it is not already present. env_dict[key] is assumed to be in the format of a PATH variable: a list of paths separated by sep tokens.

>>> env = {'PATH': '/bin:/usr/bin:/usr/local/bin'}
>>> AddPathIfNotExists(env, 'PATH', '/opt/bin')
>>> print(env['PATH'])
/opt/bin:/bin:/usr/bin:/usr/local/bin

SCons.Util.envs.AppendPath (oldpath, newpath, sep=':', delete_existing: bool = True, canonicalize: Callable | None = None) → list | str
Append newpath path elements to oldpath.
Will only add any particular path once (leaving the last one it encounters and ignoring the rest, to preserve path order), and will os.path.normpath and os.path.normcase all paths to help assure this. This can also handle the case where oldpath is a list instead of a string, in which case a list will be returned instead of a string. For example:

>>> p = AppendPath("/foo/bar:/foo", "/biz/boom:/foo")
>>> print(p)
/foo/bar:/biz/boom:/foo

If delete_existing is False, then adding a path that exists will not move it to the end; it will stay where it is in the list.

>>> p = AppendPath("/foo/bar:/foo", "/biz/boom:/foo", delete_existing=False)
>>> print(p)
/foo/bar:/foo:/biz/boom

If canonicalize is not None, it is applied to each element of newpath before use.

class SCons.Util.envs.MethodWrapper (obj: Any, method: Callable, name: str | None = None) → None
Bases: object
A generic Wrapper class that associates a method with an object.
As part of creating this MethodWrapper object an attribute with the specified name (by default, the name of the supplied method) is added to the underlying object. When that new “method” is called, our __call__() method adds the object as the first argument, simulating the Python behavior of supplying “self” on method calls.
We hang on to the name by which the method was added to the underlying base class so that we can provide a method to “clone” ourselves onto a new underlying object being copied (without which we wouldn’t need to save that info).

clone (new_object)
Returns an object that re-binds the underlying "method" to the specified new object.

SCons.Util.environ.PrependPath (oldpath, newpath, sep=':', delete_existing: bool = True, canonicalize: Callable | None = None) → list | str

Prepend newpath path elements to oldpath.
Will only add any particular path once (leaving the first one it encounters and ignoring the rest, to preserve path order), and will os.path.normpath and os.path.normcase all paths to help assure this. This can also handle the case where oldpath is a list instead of a string, in which case a list will be returned instead of a string. For example:

```python
>>> p = PrependPath("/foo/bar:/foo", "/biz/boom:/foo")
>>> print(p)
/biz/boom:/foo/bar
```

If delete_existing is False, then adding a path that exists will not move it to the beginning; it will stay where it is in the list.

```python
>>> p = PrependPath("/foo/bar:/foo", "/biz/boom:/foo", delete_existing=False)
>>> print(p)
/biz/boom:/foo/bar:/foo
```

If canonicalize is not None, it is applied to each element of newpath before use.

SCons file locking functions.
Simple-minded filesystem-based locking. Provides a context manager which acquires a lock (or at least, permission) on entry and releases it on exit.

Usage:

```python
from SCons.Util.filelock import FileLock

with FileLock("myfile.txt", writer=True) as lock:
    print(f"Lock on {lock.file} acquired.")
    # work with the file as it is now locked
```

class SCons.Util.filelock.FileLock (file: str, timeout: int | None = None, delay: float | None = 0.05, writer: bool = False)
Bases: object
Lock a file using a lockfile.
Basic locking for when multiple processes may hit an externally shared resource that cannot depend on locking within a single SCons process. SCons does not have a lot of those, but caches come to mind.
Cross-platform safe, does not use any OS-specific features. Provides context manager support, or can be called with acquire_lock() and release_lock().
Lock can be a write lock, which is held until released, or a read lock, which releases immediately upon acquisition - we want to not read a file which somebody else may be writing, but not create the writers starvation problem of the classic readers/writers lock.

TODO: Should default timeout be None (non-blocking), or 0 (block forever),
or some arbitrary number?

Parameters:
- file – name of file to lock. Only used to build the lockfile name.
- timeout – optional time (sec) to give up trying. If None, quit now if we failed to get the lock (non-blocking). If 0, block forever (well, a long time).
- delay – optional delay between tries [default 0.05s]
- writer – if True, obtain the lock for safe writing. If False (default), just wait till the lock is available, give it back right away.
Raises: **SConsLockFailure** – if the operation “timed out”, including the non-blocking mode.

__enter__ () → FileLock

Context manager entry: acquire lock if not holding.

__exit__ (exc_type, exc_value, exc_tb) → None

Context manager exit: release lock if holding.

__repr__ () → str

Nicer display if someone repr’s the lock class.

acquire_lock () → None

Acquire the lock, if possible.

If the lock is in use, check again every delay seconds. Continue until lock acquired or timeout expires.

release_lock () → None

Release the lock by deleting the lockfile.

exception SCons.Util.filelock.SConsLockFailure

Bases: Exception

Lock failure exception.

add_note () → None

Exception.add_note(note) – add a note to the exception

args

with_traceback () → None

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

SCons hash utility routines.

Routines for working with content and signature hashes.

SCons.Util.hashes.MDScollect (signatures)

Deprecated. Use hash_collect() instead.

SCons.Util.hashes.MD5filesignature (fname, chunksize: int = 65536)

Deprecated. Use hash_file_signature() instead.

SCons.Util.hashes.MD5signature (s)

Deprecated. Use hash_signature() instead.

SCons.Util.hashes._attempt_get_hash_function (hash_name, hashlib_used=<module 'hashlib' from '/opt/local/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/hashlib.py'>, sys_used=<module 'sys' (built-in)>)

Wrapper used to try to initialize a hash function given.

If successful, returns the name of the hash function back to the user.

Otherwise returns None.

SCons.Util.hashes._attempt_init_of_python_3_9_hash_object (hash_function_object, sys_used=<module 'sys' (built-in)>)

Initialize hash function with non-security indicator.

In Python 3.9 and onwards, hashlib constructors accept a keyword argument usedforsecurity, which, if set to False, lets us continue to use algorithms that have been deprecated either by FIPS or by Python itself, as the MD5 algorithm SCons prefers is not being used for security purposes as much as a short, 32 char hash that is resistant to accidental collisions.

In prior versions of python, hashlib returns a native function wrapper, which errors out when it’s queried for the optional parameter, so this function wraps that call.

It can still throw a ValueError if the initialization fails due to FIPS compliance issues, but that is assumed to be the responsibility of the caller.

SCons.Util.hashes._get_hash_object (hash_format, hashlib_used=<module 'hashlib' from '/opt/local/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/hashlib.py'>, sys_used=<module 'sys' (built-in)>)

Allocates a hash object using the requested hash format.

**Parameters:** hash_format – Hash format to use.

**Returns:** hashlib object.

SCons.Util.hashes._set_allowed_viable_default_hashes (hashlib_used, sys_used=<module 'sys' (built-in)>) → None

Check if the default hash algorithms can be called.
This utility class is sometimes called prior to setting the user-selected hash algorithm, meaning that on FIPS-compliant systems the library would default-initialize MD5 and throw an exception in set_hash_format. A common case is using the SConf options, which can run prior to main, and thus ignore the options.hash_format variable.

This function checks the DEFAULT_HASH_FORMATS and sets the ALLOWED_HASH_FORMATS to only the ones that can be called. In Python >= 3.9 this will always default to MD5 as in Python 3.9 there is an optional attribute "usedforsecurity" set for the method.

Throws if no allowed hash formats are detected.

SCons.Util.hashes._show_md5_warning (function_name) → None
Shows a deprecation warning for various MD5 functions.

SCons.Util.hashes.get_current_hash_algorithm_used ()
Returns the current hash algorithm name used.
Where the python version >= 3.9, this is expected to return md5. If python’s version is <= 3.8, this returns md5 on non-FIPS-mode platforms, and sha1 or sha256 on FIPS-mode Linux platforms.
This function is primarily useful for testing, where one expects a value to be one of N distinct hashes, and therefore the test needs to know which hash to select.

SCons.Util.hashes.get_hash_format ()
Retrieves the hash format or None if not overridden.
A return value of None does not guarantee that MD5 is being used; instead, it means that the default precedence order documented in SCons.Util.set_hash_format() is respected.

SCons.Util.hashes.hash_collect (signatures, hash_format=None)
Collects a list of signatures into an aggregate signature.

Parameters:
- signatures – a list of signatures
- hash_format – Specify to override default hash format

Returns:
the aggregate signature

SCons.Util.hashes.hash_file_signature (fname, chunksize: int = 65536, hash_format=None)
Generate the md5 signature of a file

Parameters:
- fname – file to hash
- chunksize – chunk size to read
- hash_format – Specify to override default hash format

Returns:
String of Hex digits representing the signature

SCons.Util.hashes.hash_signature (s, hash_format=None)
Generate hash signature of a string

Parameters:
- s – either string or bytes. Normally should be bytes
- hash_format – Specify to override default hash format

Returns:
String of hex digits representing the signature

SCons.Util.hashes.set_hash_format (hash_format, hashlib_used=<module 'hashlib' from '/opt/local/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/hashlib.py'>, sys_used=<module 'sys' (built-in)>)
Sets the default hash format used by SCons.
If hash_format is None or an empty string, the default is determined by this function.
Currently the default behavior is to use the first available format of the following options: MD5, SHA1, SHA256.

Various SCons utility functions

Routines which check types and do type conversions.

class SCons.Util.sctypes.Null (*args, **kwargs)
Bases: object
Null objects always and reliably ‘do nothing’.

class SCons.Util.sctypes.NullSeq (*args, **kwargs)
Bases: Null
A Null object that can also be iterated over.
SCons.Util.sctypes.get_env_bool(env, name: str, default: bool = False) → bool
Convert a construction variable to bool.
If the value of name in dict-like object env is ‘true’, ‘yes’, ‘y’, ‘on’ (case insensitive) or anything convertible to int that yields non-zero, return True; if ‘false’, ‘no’, ‘n’, ‘off’ (case insensitive) or a number that converts to integer zero return False. Otherwise, or if name is not found, return the value of default.

Parameters:
- env – construction environment, or any dict-like object.
- name – name of the variable.
- default – value to return if name not in env or cannot be converted (default: False).

SCons.Util.sctypes.get_environment_var(varstr) → str | None
Return undecorated construction variable string.
Determine if varstr looks like a reference to a single environment variable, like "$FOO" or "${FOO}". If so, return that variable with no decorations, like "FOO". If not, return None.

SCons.Util.sctypes.get_os_env_bool(name: str, default: bool = False) → bool
Convert an external environment variable to boolean.
Like get_env_bool(), but uses os.environ as the lookup dict.

SCons.Util.sctypes.is_Dict(obj, isinstance=<built-in function isinstance>, DictTypes=(<class 'dict'>, <class 'collections.UserDict'>)) → bool
Check if object is a dict.

SCons.Util.sctypes.is_List(obj, isinstance=<built-in function isinstance>, ListTypes=(<class 'list'>, <class 'collections.UserList'>, <class 'collections.deque'>)) → bool
Check if object is a list.

SCons.Util.sctypes.is_Scalar(obj, isinstance=<built-in function isinstance>, StringTypes=(<class 'str'>, <class 'collections.UserString'>), Iterable=<class 'collections.abc.Iterable'>)) → bool
Check if object is a scalar: not a container or iterable.

SCons.Util.sctypes.is_Sequence(obj, isinstance=<built-in function isinstance>, SequenceTypes=(<class 'list'>, <class 'tuple'>, <class 'collections.deque'>, <class 'collections.UserList'>, <class 'collections.abc.MappingView'>)) → bool
Check if object is a sequence.

SCons.Util.sctypes.is_String(obj, isinstance=<built-in function isinstance>, StringTypes=(<class 'str'>, <class 'collections.UserString'>)) → bool
Check if object is a string.

SCons.Util.sctypes.is_Tuple(obj, isinstance=<built-in function isinstance>, tuple=<class 'tuple'>) → bool
Check if object is a tuple.

SCons.Util.sctypes.to_String(data: bytes) → str
Return bytes data converted to text.
Useful for whole-file reads where the data needs some interpretation, particularly for Scanners. Attempts to figure out what the encoding of the text is based upon the BOM bytes, and then decodes the contents so that it’s a valid python string.
SCons API Documentation

SCons.Util.sctypes.to_bytes (s) → bytes
Convert object to bytes.
SCons.Util.sctypes.to_str (s) → str
Convert object to string.

SCons statistics routines.

This package provides a way to gather various statistics during an SCons run and dump that info in several formats.

Additionally, it probably makes sense to do stderr/stdout output of those statistics here as well.

There are basically two types of stats:

1. Timer (start/stop/time) for specific event. These events can be hierarchical. So you can record the children events of some parent. Think program compile could contain the total Program builder time, which could include linking, and stripping the executable.

2. Counter. Counting the number of events and/or objects created. This would likely only be reported at the end of a given SCons run, though it might be useful to query during a run.

    class SCons.Util.stats.CountStats
    Bases: Stats
    _abc_impl = <abc._abc_data object>
    do_append (label)
    do_nothing (*args, **kw)
    do_print ()
    enable (outfp)

class SCons.Util.stats.MemStats
    Bases: Stats
    _abc_impl = <abc._abc_data object>
    do_append (label)
    do_nothing (*args, **kw)
    do_print ()
    enable (outfp)

class SCons.Util.stats.Stats
    Bases: ABC
    _abc_impl = <abc._abc_data object>
    do_append (label)
    do_nothing (*args, **kw)
    do_print ()
    enable (outfp)

class SCons.Util.stats.TimeStats
    Bases: Stats
    _abc_impl = <abc._abc_data object>
    add_command (command, start_time, finish_time)
    do_append (label)
    do_nothing (*args, **kw)
    do_print ()
    enable (outfp)
    total_times (build_time, sconscript_time, scons_exec_time, command_exec_time)

SCons.Util.stats.add_stat_type (name, stat_object)
Add a statistic type to the global collection
SCons.Util.stats.write_scons_stats_file ()
Actually write the JSON file with debug information. Depending which of : count, time, action-timestamps, memory
their information will be written.
SCons API Documentation

SCons.Variables package

Submodules

SCons.Variables.BoolVariable module

Variable type for true/false Variables.

Usage example:

```python
opts = Variables()
opts.Add(BoolVariable('embedded', 'build for an embedded system', False))
...
if env['embedded']:
    ...
```

Return a tuple describing a boolean SCons Variable.  
The input parameters describe a boolean option. Returns a tuple including the correct converter and validator. The help text will have (yes|no) automatically appended to show the valid values. The result is usable as input to Add().

SCons.Variables.BoolVariable._text2bool (val: str) → bool  
Convert boolean-like string to boolean.  
If val looks like it expresses a bool-like value, based on the TRUE_STRINGS and FALSE_STRINGS tuples, return the appropriate value.  
This is usable as a converter function for SCons Variables.

Raises:  
ValueError – if val cannot be converted to boolean.

SCons.Variables.BoolVariable._validator (key, val, env) → None  
Validate that the value of key in env is a boolean.  
Parameter val is not used in the check.  
Usable as a validator function for SCons Variables.

Raises:  
  • KeyError – if key is not set in env  
  • UserError – if the value of key is not True or False.

SCons.Variables.EnumVariable module

Variable type for enumeration Variables.

Enumeration variables allow selection of one from a specified set of values.

Usage example:

```python
opts = Variables()
opts.Add(
    EnumVariable(
        'debug',
        help='debug output and symbols',
        default='no',
        allowed_values=('yes', 'no', 'full'),
        map={},
        ignorecase=2,
    )
)
...
if env['debug'] == 'full':
    ...
```
SCons API Documentation

SCons.Variables.EnumVariable.EnumVariable (key, help, default, allowed_values, map={}, ignorecase: int = 0) → Tuple[str, str, str, Callable, Callable]
Return a tuple describing an enumeration SCons Variable.
The input parameters describe an option with only certain values allowed. Returns A tuple including an appropriate converter and validator. The result is usable as input to Add().
key and default are passed directly on to Add().
help is the descriptive part of the help text, and will have the allowed values automatically appended.
allowed_values is a list of strings, which are the allowed values for this option.
The map-dictionary may be used for converting the input value into canonical values (e.g. for aliases).
The value of ignorecase defines the behaviour of the validator:

- 0: the validator/converter are case-sensitive.
- 1: the validator/converter are case-insensitive.
- 2: the validator/converter is case-insensitive and the converted value will always be lower-case.
The validator tests whether the value is in the list of allowed values. The converter converts input values according to the given map-dictionary (unmapped input values are returned unchanged).

SCons.Variables.ListVariable module
Variable type for list Variables.
A 'list' option may either be 'all', 'none' or a list of names separated by comma. After the option has been processed, the option value holds either the named list elements, all list elements or no list elements at all.

Usage example:

```python
list_of_libs = Split('x11 gl qt ical')

opts = Variables()
opts.Add(
    ListVariable(
        'shared',
        help='libraries to build as shared libraries',
        default='all',
        elems=list_of_libs,
    )
)
...
for lib in list_of_libs:
    if lib in env['shared']:
        env.SharedObject(...)
    else:
        env.Object(...)
```

SCons.Variables.ListVariable.ListVariable (key, help, default, names, map={}) → Tuple[str, str, str, None, Callable]
Return a tuple describing a list SCons Variable.
The input parameters describe a 'list' option. Returns a tuple including the correct converter and validator. The result is usable for input to Add().
help will have text appended indicating the legal values (not including any extra names from map).
map can be used to map alternative names to the ones in names - that is, a form of alias.
A 'list' option may either be 'all', 'none' or a list of names (separated by commas).

SCons.Variables.ListVariable._converter (val, allowedElems, mapdict) → _ListVariable

SCons.Variables.PackageVariable module
Variable type for package Variables.

127
To be used whenever a ‘package’ may be enabled/disabled and the package path may be specified.

Given these options

| x11=no   (disables X11 support) | x11=yes  (will search for the package installation dir) | x11=/usr/local/X11 (will check this path for existence) |

Can be used as a replacement for autoconf’s --with-xxx=yyy

```python
opts = Variables()
opts.Add(  
    PackageVariable(  
        key='x11',  
        help='use X11 installed here (yes = search some places)',  
        default='yes'  
    )  
)
...
if env['x11'] == True:
    dir = ...  # search X11 in some standard places ...
    env['x11'] = dir
if env['x11']:
    ...  # build with x11 ...
```

SCons.Variables.PackageVariable.PkgVariable (key, help, default, searchfunc=) → Tuple[str, str, str, Callable, Callable]

Return a tuple describing a package list SCons Variable.

- The input parameters describe a ‘package list’ option. Returns a tuple including the correct converter and validator appended. The result is usable as input to Add().
- A ‘package list’ option may either be ‘all’, ‘none’ or a pathname string. This information is appended to help.

SCons.Variables.PathVariable.validator (key, val, env, searchfunc) → None

SCons.Variables.PathVariable module

Variable type for path Variables.

To be used whenever a user-specified path override setting should be allowed.

Arguments to PathVariable are:

- **key** - name of this option on the command line (e.g. “prefix”)
- **help** - help string for option
- **default** - default value for this option
- **validator** - [optional] validator for option value. Predefined are:
  - **PathAccept** - accepts any path setting; no validation
  - **PathDir** - path must be an existing directory
  - **PathDirCreate** - path must be a dir; will create
  - **PathFile** - path must be a file
  - **PathExists** - path must exist (any type) [default]
The *validator* is a function that is called and which should return True or False to indicate if the path is valid. The arguments to the validator function are: *(key, val, env)*. *key* is the name of the option, *val* is the path specified for the option, and *env* is the environment to which the Options have been added.

Usage example:

```python
opts = Variables()
opts.Add(
    PathVariable(
        'qtdir',
        help='where the root of Qt is installed',
        default=qtdir,
        validator=PathIsDir,
    )
)
opts.Add(
    PathVariable(
        'qt_includes',
        help='where the Qt includes are installed',
        default='$qtdir/includes',
        validator=PathIsDirCreate,
    )
)
opts.Add(
    PathVariable(
        'qt_libraries',
        help='where the Qt library is installed',
        default='$qtdir/lib',
    )
)
```

**Module contents**

Adds user-friendly customizable variables to an SCons build.

```python
class SCons.Variables.Variables (files=None, args=None, is_global: bool = True)
Bases: object
Holds all the options, updates the environment with the variables, and renders the help text.
If is_global is true, this is a singleton, create only once.
```

**Parameters:**

- **files** *(optional)* – List of option configuration files to load (backward compatibility). If a single string is passed it is automatically placed in a file list (Default value = None)
- **args** *(optional)* – dictionary to override values set from files. (Default value = None)
- **is_global** *(optional)* – global instance? (Default value = True)

```python
Add (key, *args, **kwargs) → None
```

**Parameters:**

- **key** – the name of the variable, or a 5-tuple (or list). If a tuple, and there are no additional arguments, the tuple is unpacked into the four named kwargs from below. If a tuple and there are additional arguments, the first word of the tuple is taken as the key, and the remainder as aliases.
- ***args** – optional positional arguments, corresponding to the four named kwargs below.
SCons API Documentation

**Keyword Arguments:**
- **help** – help text for the options (Default value = "")
- **default** – default value for option (Default value = None)
- **validator** – function called to validate the option’s value (Default value = None)
- **converter** – function to be called to convert the option’s value before putting it in the environment. (Default value = None)
- **kwargs** – arbitrary keyword arguments used by the variable itself.

**AddVariables (**optlist**)** → None

Adds a list of options.
Each list element is a tuple/list of arguments to be passed on to the underlying method for adding options.
Example:

```python
opt.AddVariables(
    ('debug', '', 0),
    ('CC', 'The C compiler'),
    ('VALIDATE', 'An option for testing validation', 'notset', validator, None),
)
```

**FormatVariableHelpText (env, key, help, default, actual, aliases= None)** → str

Generates the help text for the options.

**Parameters:**
- **env** – an environment that is used to get the current values of the options.
- **sort** – Either a comparison function used for sorting (must take two arguments and return -1, 0 or 1) or a boolean to indicate if it should be sorted.

**Save (**filename, env**)** → None

Save the options to a file.
Saves all the options which have non-default settings to the given file as Python expressions. This file can then be used to load the options for a subsequent run. This can be used to create an option cache file.

**Parameters:**
- **filename** – Name of the file to save into
- **env** – the environment get the option values from

**UnknownVariables ()** → dict

Returns unknown variables.

**Update (**env, args= None**)** → None

Updates an environment with the option variables.

**Parameters:**
- **env** – the environment to update.
- **args (optional)** – a dictionary of keys and values to update in env. If omitted, uses the variables from the commandline.

**do_add (key, help: str = '', default= None, validator= None, converter= None, **kwargs)** → None

**aliasfmt** = "in help: %s in default: %s in actual: %s in aliases: %s"

**fmt** = "in help: %s in default: %s in actual: %s"

**instance** = None

**keys ()** → list

Returns the keywords for the options.
SCons compat package

Module contents

SCons compatibility package for old Python versions

This subpackage holds modules that provide backwards-compatible implementations of various things from newer Python versions that we cannot count on because SCons still supported older Pythons.

Other code will not generally reference things in this package through the SCons.compat namespace. The modules included here add things to the builtins namespace or the global module list so that the rest of our code can use the objects and names imported here regardless of Python version. As a result, if this module is used, it should violate the normal convention for imports (standard library imports first, then program-specific imports, each ordered alphabetically) and needs to be listed first.

The rest of the things here will be in individual compatibility modules that are either: 1) suitably modified copies of the future modules that we want to use; or 2) backwards compatible re-implementations of the specific portions of a future module’s API that we want to use.

GENERAL WARNINGS: Implementations of functions in the SCons.compat modules are NOT guaranteed to be fully compliant with these functions in later versions of Python. We are only concerned with adding functionality that we actually use in SCons, so be wary if you lift this code for other uses. (That said, making these more nearly the same as later, official versions is still a desirable goal, we just don’t need to be obsessive about it.)

We name the compatibility modules with an initial ‘_scons_’ (for example, _scons_subprocess.py is our compatibility module for subprocess) so that we can still try to import the real module name and fall back to our compatibility module if we get an ImportError. The import_as() function defined below loads the module as the “real” name (without the ‘_scons_’), after which all of the “import {module}” statements in the rest of our code will find our pre-loaded compatibility module.

class SCons.compat.NoSlotsPyPy (name, bases, dct)
   Bases: type
   Metaclass for PyPy compatibility.
   PyPy does not work well with __slots__ and __class__ assignment.

mro ()
   Return a type’s method resolution order.
SCons.compat.rename_module (new, old) → bool
   Attempt to import the old module and load it under the new name. Used for purely cosmetic name changes in Python 3.x.

Submodules

SCons.Action module

SCons Actions.

Information about executing any sort of action that can build one or more target Nodes (typically files) from one or more source Nodes (also typically files) given a specific Environment.

The base class here is ActionBase. The base class supplies just a few utility methods and some generic methods for displaying information about an Action in response to the various commands that control printing.

A second-level base class is _ActionAction. This extends ActionBase by providing the methods that can be used to show and perform an action. True Action objects will subclass _ActionAction; Action factory class objects will subclass ActionBase.

The heavy lifting is handled by subclasses for the different types of actions we might execute:

   CommandAction CommandGeneratorAction FunctionAction ListAction

The subclasses supply the following public interface methods used by other modules:

   __call__()
THE public interface, “calling” an Action object executes the command or Python function. This also takes care
of printing a pre-substitution command for debugging purposes.

**get_contents()**

Fetches the “contents” of an Action for signature calculation plus the varlist. This is what gets checksummed to
decide if a target needs to be rebuilt because its action changed.

**genstring()**

Returns a string representation of the Action **without** command substitution, but allows a
CommandGeneratorAction to generate the right action based on the specified target, source and env. This is
used by the Signature subsystem (through the Executor) to obtain an (imprecise) representation of the Action
operation for informative purposes.

Subclasses also supply the following methods for internal use within this module:

**__str__()**

Returns a string approximation of the Action; no variable substitution is performed.

**execute()**

The internal method that really, truly, actually handles the execution of a command or Python function. This is
used so that the **__call__()** methods can take care of displaying any pre-substitution representations, and **then**
execute an action without worrying about the specific Actions involved.

**get_presig()**

Fetches the “contents” of a subclass for signature calculation. The varlist is added to this to produce the
Action’s contents. TODO(?): Change this to always return bytes and not str?

**strfunction()**

Returns a substituted string representation of the Action. This is used by the _ActionAction.show() command
to display the command/function that will be executed to generate the target(s).

There is a related independent ActionCaller class that looks like a regular Action, and which serves as a wrapper for
arbitrary functions that we want to let the user specify the arguments to now, but actually execute later (when an
out-of-date check determines that it’s needed to be executed, for example). Objects of this class are returned by an
ActionFactory class that provides a **__call__()** method as a convenient way for wrapping up the functions.

SCons.Action.Action (act, *args, **kw)

A factory for action objects.

**class SCons.Action.ActionBase**

Bases: ABC

Base class for all types of action objects that can be held by other objects (Builders, Executors, etc.) This provides
the common methods for manipulating and combining those actions.

```
abc_impl = <abc._abc_data object>
batch_key (env, target, source)
genstring (target, source, env, executor: Executor | None = None) → str
get_contents (target, source, env)
abstract get_implicit_deps (target, source, env, executor: Executor | None = None)
abstract get_presig (target, source, env, executor: Executor | None = None)
gt_targets (env, executor: Executor | None)
    Returns the type of targets ($TARGETS, $CHANGED_TARGETS) used by this action.
get_varlist (target, source, env, executor: Executor | None = None)
no_batch_key (env, target, source)
presub_lines (env)
```

**class SCons.Action.ActionCaller (parent, args, kw)**

Bases: object

A class for delaying calling an Action function with specific (positional and keyword) arguments until the Action is
actually executed.

This class looks to the rest of the world like a normal Action object, but what it’s really doing is hanging on to the
arguments until we have a target, source and env to use for the expansion.

```
get_contents (target, source, env)
```
SCons API Documentation

```python
def strfunction(target, source, env)
def subst(s, target, source, env)
def subst_args(target, source, env)
def subst_kw(target, source, env)
class SCons.Action.ActionFactory(actfunc, strfunc, convert=<function ActionFactory.<lambda>>)
    Bases: object
    A factory class that will wrap up an arbitrary function as an SCons-executable Action object.
    The real heavy lifting here is done by the ActionCaller class. We just collect the (positional and keyword) arguments
    that we're called with and give them to the ActionCaller object we create, so it can hang onto them until it needs
    them.
class SCons.Action.CommandAction(cmd, **kw)
    Bases: _ActionAction
    Class for command-execution actions.
    _abc_impl = <_abc._abc_data object>
    _get_implicit_deps_heavyweight(target, source, env, executor: Executor | None, icd_int)
        Heavyweight dependency scanning involves scanning more than just the first entry in an action string. The exact
        behavior depends on the value of icd_int. Only files are taken as implicit dependencies; directories are ignored.
        If icd_int is an integer value, it specifies the number of entries to scan for implicit dependencies. Action strings are
        also scanned after a &&. So for example, if icd_int=2 and the action string is "cd <some_dir> && $PYTHON
        $SCRIPT_PATH <another_path>", the implicit dependencies would be the path to the python binary and the path
        to the script.
        If icd_int is None, all entries are scanned for implicit dependencies.
    _get_implicit_deps_lightweight(target, source, env, executor: Executor | None)
        Lightweight dependency scanning involves only scanning the first entry in an action string, even if it contains &&.
    batch_key(env, target, source)
    execute(target, source, env, executor: Executor | None = None)
        Execute a command action.
        This will handle lists of commands as well as individual commands, because construction variable substitution may
        turn a single "command" into a list. This means that this class can actually handle lists of commands, even though
        that's not how we use it externally.
    genstring(target, source, env, executor: Executor | None = None) -> str
    get_contents(target, source, env)
    get_implicit_deps(target, source, env, executor: Executor | None = None)
        Return the implicit dependencies of this action's command line.
    get_presig(target, source, env, executor: Executor | None = None)
        Return the signature contents of this action's command line.
        This strips $(-$) and everything in between the string, since those parts don't affect signatures.
    get_targets(env, executor: Executor | None)
        Returns the type of targets ($TARGETS, $CHANGED_TARGETS) used by this action.
    get_varlist(target, source, env, executor: Executor | None = None)
    no_batch_key(env, target, source)
    presub_lines(env)
    print_cmd_line(s, target, source, env) -> None
        In python 3, and in some of our tests, sys.stdout is a String io object, and it takes unicode strings only. This code
        assumes s is a regular string.
    process(target, source, env, executor=None, overrides: dict | None = None) -> Tuple[List, bool, bool]
    strfunction(target, source, env, executor: Executor | None = None, overrides: dict | None = None) -> str
class SCons.Action.CommandGeneratorAction(generator, kw)
    Bases: ActionBase
    Class for command-generator actions.
    _abc_impl = <_abc._abc_data object>
    _generate(target, source, env, for_signature, executor: Executor | None = None)
    batch_key(env, target, source)
    genstring(target, source, env, executor: Executor | None = None) -> str
```
get_contents (target, source, env)
get_implicit_deps (target, source, env, executor: Executor | None = None)
get_presig (target, source, env, executor: Executor | None = None)

Return the signature contents of this action’s command line.
This strips $(-$) and everything in between the string, since those parts don’t affect signatures.

get_targets (env, executor: Executor | None)

Returns the type of targets ($TARGETS, $CHANGED_TARGETS) used by this action.

get_varlist (target, source, env, executor: Executor | None = None)

no_batch_key (env, target, source)

presub_lines (env)

class SCons.Action.FunctionAction (execfunction, kw)

Bases: _ActionAction

Class for Python function actions.

_abc_impl = <_abc._abc_data object>

batch_key (env, target, source)
execute (target, source, env, executor: Executor | None = None)
function_name ()
genstring (target, source, env, executor: Executor | None = None) → str

get_contents (target, source, env)
get_implicit_deps (target, source, env, executor: Executor | None = None)
get_presig (target, source, env, executor: Executor | None = None)

Return the signature contents of this callable action.

get_targets (env, executor: Executor | None)

Returns the type of targets ($TARGETS, $CHANGED_TARGETS) used by this action.

get_varlist (target, source, env, executor: Executor | None = None)

no_batch_key (env, target, source)
presub_lines (env)

print_cmd_line (s, target, source, env) → None

In python 3, and in some of our tests, sys.stdout is a String io object, and it takes unicode strings only. This code assumes s is a regular string.

strfunction (target, source, env, executor: Executor | None = None)

class SCons.Action.LazyAction (var, kw)

Bases: CommandGeneratorAction, CommandAction

A LazyAction is a kind of hybrid generator and command action for strings of the form "$VAR". These strings normally expand to other strings (think "$CCCOM" to "$CC -c -o $TARGET $SOURCE"), but we also want to be able to replace them with functions in the construction environment. Consequently, we want lazy evaluation and creation of an Action in the case of the function, but that’s overkill in the more normal case of expansion to other strings.

So we do this with a subclass that’s both a generator and a command action. The overridden methods all do a quick check of the construction variable, and if it’s a string we just call the corresponding CommandAction method to do the heavy lifting. If not, then we call the same-named CommandGeneratorAction method. The CommandGeneratorAction methods work by using the overridden _generate() method, that is, our own way of handling “generation” of an action based on what’s in the construction variable.

_abc_impl = <_abc._abc_data object>

_generate (target, source, env, for_signature, executor: Executor | None = None)

_generate_cache (env)

_get_implicit_deps_heavyweight (target, source, env, executor: Executor | None, icd_int)

Heavyweight dependency scanning involves scanning more than just the first entry in an action string. The exact behavior depends on the value of icd_int. Only files are taken as implicit dependencies; directories are ignored.

If icd_int is an integer value, it specifies the number of entries to scan for implicit dependencies. Action strings are also scanned after a &&. So for example, if icd_int=2 and the action string is “cd <some_dir> && $PYTHON $SCRIPT_PATH <another_path>”, the implicit dependencies would be the path to the python binary and the path to the script.

If icd_int is None, all entries are scanned for implicit dependencies.

_get_implicit_deps_lightweight (target, source, env, executor: Executor | None)

Lightweight dependency scanning involves only scanning the first entry in an action string, even if it contains &&.
execute (target, source, env, executor: Executor | None = None)

Execute a command action.

This will handle lists of commands as well as individual commands, because construction variable substitution may turn a single "command" into a list. This means that this class can actually handle lists of commands, even though that’s not how we use it externally.

genstring (target, source, env, executor: Executor | None = None) → str

get_contents (target, source, env)

get_implicit_deps (target, source, env, executor: Executor | None = None)

Return the implicit dependencies of this action’s command line.

generate (target, source, env, executor: Executor | None = None)

get_presig (target, source, env, executor: Executor | None = None)

Return the signature contents of this action’s command line.

get_parent_class (env)

get_targets (env, executor: Executor | None)

Returns the type of targets ($TARGETS, $CHANGED_TARGETS) used by this action.

no_batch_key (env, target, source)

presub_lines (env)

print_cmd_line (s, target, source, env) → None

In python 3, and in some of our tests, sys.stdout is a String io object, and it takes unicode strings only. This code assumes s is a regular string.

process (target, source, env, executor=None, overrides: dict | None = None) → Tuple[List, bool, bool]

strfunction (target, source, env, executor: Executor | None = None, overrides: dict | None = None)

SCons API Documentation

class SCons.Action.ListAction (actionlist)

Bases: ActionBase

Class for lists of other actions.

_abc_impl = <_abc._abc_data object>

batch_key (env, target, source)

genstring (target, source, env, executor: Executor | None = None) → str

get_contents (target, source, env)

get_implicit_deps (target, source, env, executor: Executor | None = None)

get_presig (target, source, env, executor: Executor | None = None)

Return the signature contents of this action list.

get_varlist (target, source, env, executor: Executor | None = None)

no_batch_key (env, target, source)

presub_lines (env)

class SCons.Action._null

Bases: ActionBase

Base class for actions that create output objects.

_abc_impl = <_abc._abc_data object>

batch_key (env, target, source)

genstring (target, source, env, executor: Executor | None = None) → str

get_contents (target, source, env)

get_implicit_deps (target, source, env, executor: Executor | None = None)

get_presig (target, source, env, executor: Executor | None = None)

get_targets (env, executor: Executor | None)

Returns the type of targets ($TARGETS, $CHANGED_TARGETS) used by this action.

get_varlist (target, source, env, executor: Executor | None = None)

no_batch_key (env, target, source)
presub_lines (env)
print_cmd_line (s, target, source, env) → None

In python 3, and in some of our tests, sys.stdout is a String io object, and it takes unicode strings only. This code assumes s is a regular string.

SCons.Action._actionAppend (act1, act2)
Joins two actions together.
Mainly, it handles ListActions by concatenating into a single ListAction.

SCons.Action._callable_contents (obj) → bytecode
Return the signature contents of a callable Python object.

SCons.Action._code_contents (code, docstring=None) → bytecode
Return the signature contents of a code object.
By providing direct access to the code object of the function, Python makes this extremely easy. Hooray!
Unfortunately, older versions of Python include line number indications in the compiled byte code. Boo! So we remove the line number byte codes to prevent recompilations from moving a Python function.

See:

- https://docs.python.org/3/library/inspect.html

For info on what each co_variable provides
The signature is as follows (should be byte/chars):

- co_argcount - Returns the number of positional arguments (including arguments with default values).
- co_varnames - Returns a tuple containing the names of the local variables (starting with the argument names).
- co_cellvars - Returns a tuple containing the names of local variables that are referenced by nested functions.
- co_freevars - Returns a tuple containing the names of free variables.
- co_consts - Returns a tuple containing the literals used by the bytecode.
- co_names - Returns a tuple containing the names used by the bytecode.
- co_code - Returns a string representing the sequence of bytecode instructions.

SCons.Action._do_create_action (act, kw)
The internal implementation for the Action factory method.
This handles the fact that passing lists to Action() itself has different semantics than passing lists as elements of lists.
The former will create a ListAction, the latter will create a CommandAction by converting the inner list elements to strings.

SCons.Action._do_create_keywords (args, kw)
This converts any arguments after the action argument into their equivalent keywords and adds them to the kw argument.

SCons.Action._do_create_list_action (act, kw) → ListAction
A factory for list actions.
Convert the input list act into Actions and then wrap them in a ListAction. If act has only a single member, return that member, not a ListAction. This is intended to allow a contained list to specify a command action without being processed into a list action.

SCons.Action._function_contents (func) → bytecode
Return the signature contents of a function.
The signature is as follows (should be byte/chars):

- func.__code__ - The code object representing the compiled function body.
- func.__defaults__ - A tuple containing default argument values for those arguments that have defaults, or None if no arguments have a default value
- func.__closure__ - None or a tuple of cells that contain bindings for the function’s free variables.

class SCons.Action._null (object)
Bases: object

See also: https://docs.python.org/3/reference/datamodel.html
SCons API Documentation

SCons.Action._object_contents(obj) → bytearray
Return the signature contents of any Python object.
We have to handle the case where object contains a code object since it can be pickled directly.

SCons.Action._object_instance_content(obj)
Returns consistent content for a action class or an instance thereof

Parameters:
- obj Should be either an action class or an instance thereof

Returns: bytearray or bytes representing the obj suitable for generating a signature from.

SCons.Action._resolve_shell_env(env, target, source)
Returns a resolved execution environment.
First get the execution environment. Then if SHELL_ENV_GENERATORS is set and is iterable, call each function to allow it to alter the created execution environment, passing each the returned execution environment from the previous call.
New in version 4.4.

SCons.Action._string_from_cmd_list(cmd_list)
Takes a list of command line arguments and returns a pretty representation for printing.

SCons.Action._subproc(scons_env, cmd, error='ignore', **kw)
Wrapper for subprocess.Popen which pulls from construction env.
Use for calls to subprocess which need to interpolate values from an SCons construction environment into the environment passed to subprocess. Adds an an error-handling argument. Adds ability to specify std{in,out,err} with “devnull” tag.
Depreciated since version 4.6.

SCons.Action.default_exitstatfunc(s)

SCons.Action.get_default_ENV(env)
Returns an execution environment.
If there is one in env, just use it, else return the Default Environment, instantiated if necessary.
A fiddlin' little function that has an import SCons.Environment which cannot be moved to the top level without creating an import loop. Since this import creates a local variable named SCons, it blocks access to the global variable, so we move it here to prevent complaints about local variables being used uninitialized.

SCons.Action.rfile(n)

SCons.Action.scons_subproc_run(scons_env, *args, **kwargs) → CompletedProcess
Run an external command using an SCons execution environment.
SCons normally runs external build commands using subprocess, but does not harvest any output from such commands. This function is a thin wrapper around subprocess.run() allowing running a command in an SCons context (i.e. uses an “execution environment” rather than the user’s existing environment), and provides the ability to return any output in a subprocess.CompletedProcess instance (this must be selected by setting stdout and/or stderr to PIPE, or setting capture_output=True - see Keyword Arguments). Typical use case is to run a tool’s “version” option to find out the installed version.
If supplied, the env keyword argument provides an execution environment to process into appropriate form before it is supplied to subprocess; if omitted, scons_env is used to derive a suitable default. The other keyword arguments are passed through, except that the SCons legacy error keyword is remapped to the subprocess check keyword; if both are omitted check=False will be passed. The caller is responsible for setting up the desired arguments for subprocess.run().
This function retains the legacy behavior of returning something vaguely usable even in the face of complete failure, unless check=True (in which case an error is allowed to be raised): it synthesizes a CompletedProcess instance in this case.
A subset of interesting keyword arguments follows; see the Python documentation of subprocess for the complete list.
Keyword Arguments:

- **stdout** – (and **stderr**, **stdin**) if set to subprocess.PIPE. send input to or collect output from the relevant stream in the subprocess; the default None does no redirection (i.e. output or errors may go to the console or log file, but is not captured); if set to subprocess.DEVNULL they are explicitly thrown away. capture_output=True is a synonym for setting both **stdout** and **stderr** to PIPE.

- **text** – open **stdin**, **stdout**, **stderr** in text mode. Default is binary mode. universal_newlines is a synonym.

- **encoding** – specifies an encoding. Changes to text mode.

- **errors** – specified error handling. Changes to text mode.

- **input** – a byte sequence to be passed to **stdin**, unless text mode is enabled, in which case it must be a string.

- **shell** – if true, the command is executed through the shell.

- **check** – if true and the subprocess exits with a non-zero exit code, raise a subprocess.CalledProcessError exception. Otherwise (the default) in case of an OSError, report the exit code in the CompletedProcess instance.

New in version 4.6.

SCons.Builder module

SCons.Builder

Builder object subsystem.

A Builder object is a callable that encapsulates information about how to execute actions to create a target Node (file) from source Nodes (files), and how to create those dependencies for tracking.

The main entry point here is the Builder() factory method. This provides a procedural interface that creates the right underlying Builder object based on the keyword arguments supplied and the types of the arguments.

The goal is for this external interface to be simple enough that the vast majority of users can create new Builders as necessary to support building new types of files in their configurations, without having to dive any deeper into this subsystem.

The base class here is BuilderBase. This is a concrete base class which does, in fact, represent the Builder objects that we (or users) create.

There is also a proxy that looks like a Builder:

**CompositeBuilder**

This proxies for a Builder with an action that is actually a dictionary that knows how to map file suffixes to a specific action. This is so that we can invoke different actions (compilers, compile options) for different flavors of source files.

Builders and their proxies have the following public interface methods used by other modules:

- **__call__()**
  THE public interface. Calling a Builder object (with the use of internal helper methods) sets up the target and source dependencies, appropriate mapping to a specific action, and the environment manipulation necessary for overridden construction variable. This also takes care of warning about possible mistakes in keyword arguments.

- **add_emitter()**
  Adds an emitter for a specific file suffix, used by some Tool modules to specify that (for example) a yacc invocation on a .y can create a .h and a .c file.

- **add_action()**
Adds an action for a specific file suffix, heavily used by Tool modules to add their specific action(s) for turning a source file into an object file to the global static and shared object file Builders.

There are the following methods for internal use within this module:

- **_execute()**
  The internal method that handles the heavily lifting when a Builder is called. This is used so that the _call_() methods can set up warning about possible mistakes in keyword-argument overrides, and then execute all of the steps necessary so that the warnings only occur once.

- **get_name()**
  Returns the Builder’s name within a specific Environment, primarily used to try to return helpful information in error messages.

- **adjust_suffix()**
- **get_prefix()**
- **get_suffix()**
- **get_src_suffix()**
- **set_src_suffix()**

Miscellaneous stuff for handling the prefix and suffix manipulation we use in turning source file names into target file names.

SCons.Builder.Builder (**kw**)
A factory for builder objects.

```
class SCons.Builder.BuilderBase (action=None, prefix: str = '', suffix: str = '', src_suffix: str = '', target_factory=None, source_factory=None, target_scanner=None, source_scanner=None, emitter=None, multi: int = 0, env=None, single_source: bool = False, name=None, chdir=<class 'SCons.Builder._Null'>, is_explicit: bool = True, src_builder=None, ensure_suffix: bool = False, **overrides)
```

Bases: object

Base class for Builders, objects that create output nodes (files) from input nodes (files).

```
_create_nodes (env, target=None, source=None)
Create and return lists of target and source nodes.

_execute (env, target, source, overwarn={}, executor_kw={})

_get_sdct (env)
    Returns a dictionary mapping all of the source suffixes of all src_builders of this Builder to the underlying Builder that should be called first.
    This dictionary is used for each target specified, so we save a lot of extra computation by memoizing it for each construction environment.

    Note that this is re-computed each time, not cached, because there might be changes to one of our source Builders (or one of their source Builders, and so on, and so on...) that we can’t “see.”
    The underlying methods we call cache their computed values, though, so we hope repeatedly aggregating them into a dictionary like this won’t be too big a hit. We may need to look for a better way to do this if performance data show this has turned into a significant bottleneck.

_get_src_builders_key (env)

_subst_src_suffixes_key (env)
```

```
add_emitter (suffix, emitter) → None
    Add a suffix-emitter mapping to this Builder.
    This assumes that emitter has been initialized with an appropriate dictionary type, and will throw a TypeError if not, so the caller is responsible for knowing that this is an appropriate method to call for the Builder in question.

add_src_builder (builder) → None
    Add a new Builder to the list of src_builders.
    This requires wiping out cached values so that the computed lists of source suffixes get re-calculated.

adjust_suffix (suffix)
get_name (env)
```
Attempts to get the name of the Builder.
Look at the BUILDERS variable of env, expecting it to be a dictionary containing this Builder, and return the key of
the dictionary. If there's no key, then return a directly-configured name (if there is one) or the name of the class (by
default).

get_prefix (env, sources=[])  
get_src_builders (env)  
    Returns the list of source Builders for this Builder.
    This exists mainly to look up Builders referenced as strings in the ‘BUILDER’ variable of the construction
    environment and cache the result.

get_src_suffix (env)  
    Get the first src_suffix in the list of src_suffixes.
get_suffix (env, sources=[])  
set_src_suffix (src_suffix) → None  
set_suffix (suffix) → None

splitext (path, env=None)

src_builder_sources (env, source, overwarn={})

src_suffixes (env)
    Returns the list of source suffixes for all src_builders of this Builder.
    This is essentially a recursive descent of the src_builder “tree.” (This value isn’t cached because there may be
    changes in a src_builder many levels deep that we can’t see.)

subst_src_suffixes (env)
    The suffix list may contain construction variable expansions, so we have to evaluate the individual strings. To avoid
doing this over and over, we memoize the results for each construction environment.

**class** SCons.Builder.CallableSelector

Bases: Selector

A callable dictionary that will, in turn, call the value it finds if it can.

clear () → None. Remove all items from D.

copy () → a shallow copy of D

fromkeys (value=None, /)
    Create a new dictionary with keys from iterable and values set to value.

get (key, default=None, /)
    Return the value for key if key is in the dictionary, else default.

items () → a set-like object providing a view on D’s items

keys () → a set-like object providing a view on D’s keys

pop (k[, d]) → v, remove specified key and return the corresponding value.
    If the key is not found, return the default if given; otherwise, raise a KeyError.

popitem ()
    Remove and return a (key, value) pair as a 2-tuple.
    Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault (key, default= None, /)
    Insert key with a value of default if key is not in the dictionary.
    Return the value for key if key is in the dictionary, else default.

update ([, E], **F) → None. Update D from dict/iterable E and F.
    If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys()
    method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values () → an object providing a view on D’s values

**class** SCons.Builder.CompositeBuilder (builder, cmdgen)

Bases: Proxy

A Builder Proxy whose main purpose is to always have a DictCmdGenerator as its action, and to provide access to
the DictCmdGenerator’s add_action() method.

__getattr__ (name)
    Retrieve an attribute from the wrapped object.

    Raises: AttributeError – if attribute name doesn’t exist.

add_action (suffix, action) → None

get ()
Retrieve the entire wrapped object

**class** SCons.Builder.DictCmdGenerator

(mapping=None, source_ext_match: bool = True)

**Bases:** Selector

This is a callable class that can be used as a command generator function. It holds on to a dictionary mapping file suffixes to Actions. It uses that dictionary to return the proper action based on the file suffix of the source file.

```python
add_action(suffix, action) → None
```

Add a suffix-action pair to the mapping.

clear () → None. Remove all items from D.

copy () → a shallow copy of D.

```python
fromkeys (value=None, /)
```

Create a new dictionary with keys from iterable and values set to value.

get (key, default=None, /)

Return the value for key if key is in the dictionary, else default.

```python
items () → a set-like object providing a view on D's items
```

get (key, default=None, /)

Return the value for key if key is in the dictionary, else default.

```python
keys () → a set-like object providing a view on D's keys
```

pop (k[, d]) → v, remove specified key and return the corresponding value.

If the key is not found, return the default if given; otherwise, raise a KeyError.

```python
popitem ()
```

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

```python
setdefault (key, default=None, /)
```

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

```python
src_suffixes ()
```

update ([, E], **F) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

```python
values () → an object providing a view on D's values
```

**class** SCons.Builder.DictEmitter

**Bases:** Selector

A callable dictionary that maps file suffixes to emitters. When called, it finds the right emitter in its dictionary for the suffix of the first source file, and calls that emitter to get the right lists of targets and sources to return. If there's no emitter for the suffix in its dictionary, the original target and source are returned.

clear () → None. Remove all items from D.

copy () → a shallow copy of D.

```python
fromkeys (value=None, /)
```

Create a new dictionary with keys from iterable and values set to value.

get (key, default=None, /)

Return the value for key if key is in the dictionary, else default.

```python
items () → a set-like object providing a view on D's items
```

keys () → a set-like object providing a view on D's keys

```python
pop (k[, d]) → v, remove specified key and return the corresponding value.
```

If the key is not found, return the default if given; otherwise, raise a KeyError.

```python
popitem ()
```

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

```python
setdefault (key, default=None, /)
```

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

```python
update ([, E], **F) → None. Update D from dict/iterable E and F.
```

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

```python
values () → an object providing a view on D's values
```

**class** SCons.Builder_EMITTERProxy

**Bases:** object

SCons API Documentation
This is a callable class that can act as a Builder emitter. It holds on to a string that is a key into an Environment dictionary, and will look there at actual build time to see if it holds a callable. If so, we will call that as the actual emitter.

```python
class SCons.Builder.ListEmitter (initlist=None)
    Bases: UserList
    A callable list of emitters that calls each in sequence, returning the result.

    _abc_impl = <_abc._abc_data object>
    append (item)
        S.append(value) -- append value to the end of the sequence
    clear ()
        clear () → None -- remove all items from S
    copy ()
    count (value)
        count (value) → integer -- return number of occurrences of value
    extend (other)
        extend (iterable) -- extend sequence by appending elements from the iterable
    index (value[, start[, stop]])
        index (value[, start[, stop]]) → integer -- return first index of value.
        Raises ValueError if the value is not present.
        Supporting start and stop arguments is optional, but recommended.
    insert (i, item)
        insert (index, value) -- insert value before index
    pop ([, index])
        pop ([, index]) → item -- remove and return item at index (default last).
        Raise IndexError if list is empty or index is out of range.
    remove (item)
        remove (value) -- remove first occurrence of value. Raise ValueError if the value is not present.
    reverse ()
        S.reverse() -- reverse IN PLACE
    sort (*args, **kwds)
```

```python
class SCons.Builder.OverrideWarner (mapping)
    Bases: UserDict
    A class for warning about keyword arguments that we use as overrides in a Builder call.
    This class exists to handle the fact that a single Builder call can actually invoke multiple builders. This class only emits the warnings once, no matter how many Builders are invoked.

    _abc_impl = <_abc._abc_data object>
    clear ()
        clear () → None. Remove all items from D.
    copy ()
    classmethod fromkeys (iterable, value=None)
    get (k[, d])
        get (k[, d]) → D[k] if k in D, else d. d defaults to None.
    items ()
        items () → a set-like object providing a view on D's items
    keys ()
        keys () → a set-like object providing a view on D's keys
    pop (k[, d])
        pop (k[, d]) → v, remove specified key and return the corresponding value.
        If key is not found, d is returned if given, otherwise KeyError is raised.
    popitem ()
        popitem () → (k, v), remove and return some (key, value) pair
        as a 2-tuple; but raise KeyError if D is empty.
    setdefault (k[, d])
        setdefault (k[, d]) → D[k]=d if k not in D
    update ([, E], **F)
        update ([, E], **F) → None. Update D from mapping/iterable E and F.
        If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k]=v In either case, this is followed by: for k, v in F.items(): D[k] = v
    values ()
        values () → an object providing a view on D's values
    warn ()
        warn () → None
```

```python
class SCons.Builder._Null
    Bases: object
```

```python
class SCons.Builder._node_errors (builder, env, tlist, slist)
    Validate that the lists of target and source nodes are legal for this builder and environment. Raise errors or issue warnings as appropriate.
```

```python
SCons.Builder._null
    alias of _Null
```

```python
SCons.Builder.is_a_Builder (obj) → bool
```
“Returns True if the specified obj is one of our Builder classes.
The test is complicated a bit by the fact that CompositeBuilder is a proxy, not a subclass of BuilderBase.
SCons.Builder.match_splitext (path, suffixes=[])

SCons.CacheDir module

CacheDir support

```python
class SCons.CacheDir.CacheDir (path):
    Bases: object
    CacheDebug (fmt, target, cachefile) → None
    _readconfig (path)
        Read the cache config.
        If directory or config file do not exist, create. Take advantage of Py3 capability in os.makedirs() and in file open):
            just try the operation and handle failure appropriately.
        Omit the check for old cache format, assume that’s old enough there will be none of those left to worry about.

        Parameters:    path – path to the cache directory
cachepath (node) → tuple
            Return where to cache a file.
            Given a Node, obtain the configured cache directory and the path to the cached file, which is generated from the
            node’s build signature. If caching is not enabled for the None, return a tuple of None.
classmethod copy_from_cache (env, src, dst) → str
            Copy a file from cache.
classmethod copy_to_cache (env, src, dst) → str
            Copy a file to cache.
            Just use the FS copy2 (“with metadata”) method, except do an additional check and if necessary a chmod to
            ensure the cachefile is writeable, to forestall permission problems if the cache entry is later updated.
get_cachedir_csig (node)
```

```python
property hit_ratio: float
is_enabled () → bool
is_readonly () → bool
property misses: int
push (node)
push_if_forced (node)
retrieve (node) → bool
```

Retrieve a node from cache.
Returns True if a successful retrieval resulted.
This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe
stuff in built()

Note that there’s a special trick here with the execute flag (one that’s not normally done for other actions). Basically
if the user requested a no_exec (-n) build, then SCons.Action.execute_actions is set to 0 and when any action is
called, it does its showing but then just returns zero instead of actually calling the action execution operation.
The problem for caching is that if the file does NOT exist in cache then the CacheRetrieveString won’t return anything
to show for the task, but the Action.__call__ won’t call CacheRetrieveFunc; instead it just returns zero, which
makes the code below think that the file was successfully retrieved from the cache, therefore it doesn’t do any
subsequent building. However, the CacheRetrieveString didn’t print anything because it didn’t actually exist in the
cache, and no more build actions will be performed, so the user just sees nothing. The fix is to tell Action.__call__
to always execute the CacheRetrieveFunc and then have the latter explicitly check SCons.Action.execute_actions
itself.

```python
SCons.CacheDir.CachePushFunc (target, source, env)
SCons.CacheDir.CacheRetrieveFunc (target, source, env) → int
SCons.CacheDir.CacheRetrieveString (target, source, env) → None
```

SCons.Conftest module

Autoconf-like configuration support
The purpose of this module is to define how a check is to be performed.

A context class is used that defines functions for carrying out the tests, logging and messages. The following methods and members must be present:

```python
context.Display(msg)
Function called to print messages that are normally displayed for the user. Newlines are explicitly used. The text should also be written to the logfile!
```

```python
context.Log(msg)
Function called to write to a log file.
```

```python
context.BuildProg(text, ext)
Function called to build a program, using “ext” for the file extension. Must return an empty string for success, an error message for failure. For reliable test results building should be done just like an actual program would be build, using the same command and arguments (including configure results so far).
```

```python
context.CompileProg(text, ext)
Function called to compile a program, using “ext” for the file extension. Must return an empty string for success, an error message for failure. For reliable test results compiling should be done just like an actual source file would be compiled, using the same command and arguments (including configure results so far).
```

```python
context.AppendLIBS(lib_name_list)
Append “lib_name_list” to the value of LIBS. “lib_namelist” is a list of strings. Return the value of LIBS before changing it (any type can be used, it is passed to SetLIBS() later.)
```

```python
context.PrependLIBS(lib_name_list)
Prepend “lib_name_list” to the value of LIBS. “lib_namelist” is a list of strings. Return the value of LIBS before changing it (any type can be used, it is passed to SetLIBS() later.)
```

```python
context.SetLIBS(value)
Set LIBS to “value”. The type of “value” is what AppendLIBS() returned. Return the value of LIBS before changing it (any type can be used, it is passed to SetLIBS() later.)
```

```python
context.headerfilename
Name of file to append configure results to, usually “confdefs.h”. The file must not exist or be empty when starting. Empty or None to skip this (some tests will not work!).
```

```python
context.config_h (may be missing).
If present, must be a string, which will be filled with the contents of a config_h file.
```

```python
context.vardict
Dictionary holding variables used for the tests and stores results from the tests, used for the build commands. Normally contains “CC”, “LIBS”, “CPPFLAGS”, etc.
```

```python
context.havedict
Dictionary holding results from the tests that are to be used inside a program. Names often start with “HAVE_”. These are zero (feature not present) or one (feature present). Other variables may have any value, e.g., “PERLVERSION” can be a number and “SYSTEMNAME” a string.
```

```python
SCons.Conftest.CheckBuilder (context, text=None, language=None)
Configure check to see if the compiler works. Note that this uses the current value of compiler and linker flags, make sure $CFLAGS, $CPPFLAGS and $LIBS are set correctly. “language” should be “C” or “C++” and is used to select the compiler. Default is “C”. “text” may be used to specify the code to be build. Returns an empty string for success, an error message for failure.
```

```python
SCons.Conftest.CheckCC (context)
Configure check for a working C compiler. This checks whether the C compiler, as defined in the $CC construction variable, can compile a C source file. It uses the current $CCCOM value too, so that it can test against non working flags.
```

```python
SCons.Conftest.CheckCXX (context)
Configure check for a working CXX compiler.
```
This checks whether the CXX compiler, as defined in the $CXX construction variable, can compile a CXX source file. It uses the current $CXXCOM value too, so that it can test against non working flags.

SCons.Conftest.CheckDeclaration (context, symbol, includes=None, language=None)
Checks whether symbol is declared.
Use the same test as autoconf, that is test whether the symbol is defined as a macro or can be used as an r-value.

Parameters:
- **symbol** – str the symbol to check
- **includes** – str Optional “header” can be defined to include a header file.
- **language** – str only C and C++ supported.

Returns: bool True if the check failed, False if succeeded.

Return type: status

SCons.Conftest.CheckFunc (context, function_name, header=None, language=None, funcargs=None)
Configure check for a function “function_name”. “language” should be “C” or “C++” and is used to select the compiler. Default is “C”. Optional “header” can be defined to define a function prototype, include a header file or anything else that comes before main(). Optional “funcargs” can be defined to define an argument list for the generated function invocation. Sets HAVE_function_name in context.havedict according to the result. Note that this uses the current value of compiler and linker flags, make sure $CFLAGS, $CPPFLAGS and $LIBS are set correctly. Returns an empty string for success, an error message for failure.

Changed in version 4.7.0: The funcargs parameter was added.

SCons.Conftest.CheckHeader (context, header_name, header=None, language=None, include_quotes=None)
Configure check for a C or C++ header file “header_name”. Optional “header” can be defined to do something before including the header file (unusual, supported for consistency). “language” should be “C” or “C++” and is used to select the compiler. Default is “C”. Sets HAVE_header_name in context.havedict according to the result. Note that this uses the current value of compiler and linker flags, make sure $CFLAGS and $CPPFLAGS are set correctly. Returns an empty string for success, an error message for failure.

SCons.Conftest.CheckLib (context, libs, func_name=None, header=None, extra_libs=None, call=None, language=None, autoadd: int = 1, append: bool = True, unique: bool = False)
Configure check for a C or C++ libraries “libs”. Searches through the list of libraries, until one is found where the test succeeds. Tests if “func_name” or “call” exists in the library. Note: if it exists in another library the test succeeds anyway! Optional “header” can be defined to include a header file. If not given a default prototype for “func_name” is added. Optional “extra_libs” is a list of library names to be added after “lib_name” in the build command. To be used for libraries that “lib_name” depends on. Optional “call” replaces the call to “func_name” in the test code. It must consist of complete C statements, including a trailing “;”. Both “func_name” and “call” arguments are optional, and in that case, just linking against the lib is tested. “language” should be “C” or “C++” and is used to select the compiler. Default is “C”. Note that this uses the current value of compiler and linker flags, make sure $CFLAGS and $CPPFLAGS and $LIBS are set correctly. Returns an empty string for success, an error message for failure.

SCons.Conftest.CheckMember (context, aggregate_member, header=None, language=None)
Configure check for a C or C++ member “aggregate_member”. Optional “header” can be defined to include a header file. “language” should be “C” or “C++” and is used to select the compiler. Default is “C”. Note that this uses the current value of compiler and linker flags, make sure $CFLAGS, $CPPFLAGS and $LIBS are set correctly.

Parameters:
- **aggregate_member** – str the member to check. For example, ‘struct tm.tm_gmtoff’.
- **includes** – str Optional “header” can be defined to include a header file.
- **language** – str only C and C++ supported.

Returns the status (0 or False = Passed, True/non-zero = Failed).

SCons.Conftest.CheckProg (context, prog_name)
Configure check for a specific program.
Check whether program prog_name exists in path. If it is found, returns the path for it, otherwise returns None.

SCons.Conftest.CheckSHCC (context)
Configure check for a working shared C compiler.
This checks whether the C compiler, as defined in the $SHCC construction variable, can compile a C source file. It uses the current $SHCCCOM value too, so that it can test against non working flags.
SCons API Documentation

SCons.Conftest.CheckSHCXX (context)
Configure check for a working shared CXX compiler.
This checks whether the CXX compiler, as defined in the $SHCXX construction variable, can compile a CXX source
file. It uses the current $SHCXXCOM value too, so that it can test against non working flags.

SCons.Conftest.CheckType (context, type_name, fallback=None, header=None, language=None)
Configure check for a C or C++ type "type_name". Optional "header" can be defined to include a header file.
"language" should be "C" or "C++" and is used to select the compiler. Default is "C". Sets HAVE_type_name in
context.havedict according to the result. Note that this uses the current value of compiler and linker flags, make sure
$CFLAGS, $CPPFLAGS and $LIBS are set correctly. Returns an empty string for success, an error message for failure.

SCons.Conftest.CheckTypeSize (context, type_name, header=None, language=None, expect=None)
This check can be used to get the size of a given type, or to check whether the type is of expected size.

Parameters:
- `type` (str) – str the type to check
- `includes` (sequence) – sequence list of headers to include in the test code before testing the type
- `language` (str) – str ‘C’ or ‘C++’
- `expect` (int) – int if given, will test wether the type has the given number of bytes. If not
given, will automatically find the size.
- `Returns` – status int 0 if the check failed, or the found size of the type if the check succeeded.

SCons.Conftest._Have (context, key, have, comment=None)
Store result of a test in context.havedict and context.headerfilename.

Parameters:
- `key` is a “HAVE_abc” name. It is turned into all CAPITALS and non-alphanumerics are
  replaced by an underscore.
- `have` - value as it should appear in the header file, include quotes when desired and
  escape special characters!
- `comment` is the C comment to add above the line defining the symbol (the comment is
  automatically put inside a /* */). If None, no comment is added.

The value of “have” can be:
- 1 - Feature is defined, add “#define key”.
- 0 - Feature is not defined, add “/* #undef key */”. Adding “undef” is what autoconf does. Not useful for the
  compiler, but it shows that the test was done.
- number - Feature is defined to this number “#define key have”. Doesn’t work for 0 or 1, use a string then.
- string - Feature is defined to this stringa “#define key have”.

SCons.Conftest._LogFailed (context, text, msg) → None
Write to the log about a failed program. Add line numbers, so that error messages can be understood.

SCons.Conftest._YesNoResult (context, ret, key, text, comment=None) → None
Handle the result of a test with a “yes” or “no” result.

Parameters:
- `ret` is the return value: empty if OK, error message when not.
- `key` is the name of the symbol to be defined (HAVE_foo).
- `text` is the source code of the program used for testing.
- `comment` is the C comment to add above the line defining the symbol (the comment is
  automatically put inside a /* */). If None, no comment is added.

SCons.Conftest._check_empty_program (context, comp, text, language, use_shared: bool = False)
Return 0 on success, 1 otherwise.

SCons.Conftest._lang2suffix (lang)
Convert a language name to a suffix. When “lang” is empty or None C is assumed. Returns a tuple (lang, suffix, None) when it works. For an unrecognized language returns (None, None, msg).

**Where:**
- lang = the unified language name
- suffix = the suffix, including the leading dot
- msg = an error message

**SCons.Debug module**

Code for debugging SCons internal things.

Shouldn’t be needed by most users. Quick shortcuts:

```python
from SCons.Debug import caller_trace
caller_trace()
```

```python
SCons.Debug.Trace(msg, tracefile=None, mode='w', tstamp=False) → None
```

Write a trace message.

Write messages when debugging which do not interfere with stdout. Useful in tests, which monitor stdout and would break with unexpected output. Trace messages can go to the console (which is opened as a file), or to a disk file; the tracefile argument persists across calls unless overridden.

**Parameters:**
- `tracefile` – file to write trace message to. If omitted, write to the previous trace file (default: console).
- `mode` – file open mode (default: 'w')
- `tstamp` – write relative timestamps with trace. Outputs time since scons was started, and time since last trace (default: False)

```python
SCons.Debug._dump_one_caller(key, file, level=0) → None
```

```python
SCons.Debug.caller_stack() → return caller’s stack
```

```python
SCons.Debug.caller_trace(back=0) → None
```

Trace caller stack and save info into global dicts, which are printed automatically at the end of SCons execution.

```python
SCons.Debug.countLoggedInstances(classes, file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>) → None
```

```python
SCons.Debug.dumpLoggedInstances(classes, file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>) → None
```

```python
SCons.Debug.dump_caller_counts(file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>) → None
```

```python
SCons.Debug.fetchLoggedInstances(classes='*') → None
```

```python
SCons.Debug.func_shorten(func_tuple) → None
```

```python
SCons.Debug.listLoggedInstances(classes, file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>) → None
```

```python
SCons.Debug.logInstanceCreation(instance, name=None) → None
```

```python
SCons.Debug.memory() → int
```

```python
SCons.Debug.string_to_classes(s) → None
```

**SCons.Defaults module**

Builders and other things for the local site.

Here’s where we’ll duplicate the functionality of autoconf until we move it into the installation procedure or use something like qmconf.

The code that reads the registry to find MSVC components was borrowed from distutils.msvccompiler.

```python
SCons.Defaults.DefaultEnvironment(*args, **kwargs) → None
```

Construct the global (“default”) construction environment.

The environment is provisioned with the values from `kwargs`.
After the environment is created, this function is replaced with a reference to `_fetch_DefaultEnvironment()` which efficiently returns the initialized default construction environment without checking for its existence. Historically, some parts of the code held references to this function. Thus it still has the existence check for `_default_env` rather than just blindly creating the environment and overwriting itself.

class SCons.Defaults.NullCmdGenerator (cmd)
Bases: object
Callable class for use as a no-effect command generator.
The `__call__` method for this class simply returns the thing you instantiated it with. Example usage:

```python
env['DO NOTHING'] = NullCmdGenerator
env['LINKCOM'] = '${DO NOTHING($LINK $SOURCES $TARGET)}'
```

SCons.Defaults.SharedFlagChecker (source, target, env)
SCons.Defaults.SharedObjectEmitter (target, source, env)
SCons.Defaults.StaticObjectEmitter (target, source, env)
class SCons.Defaults.Variable_Method_Caller (variable, method)
Bases: object
A class for finding a construction variable on the stack and calling one of its methods.
Used to support "construction variables" appearing in string `eval`s that actually stand in for methods--specifically, the use of "RDirs" in a call to :func:`_concat` that should actually execute the `TARGET.RDirs` method.
Historical note: This was formerly supported by creating a little "build dictionary" that mapped RDirs to the method, but this got in the way of Memoizing construction environments, because we had to create new environment objects to hold the variables.

SCons.Defaults._lib_either_version_flag (env, version_var1, version_var2, flags_var)
if $version_var1 or $version_var2 is not empty, returns env[flags_var], otherwise returns None :param env: :param version_var1: :param version_var2: :param flags_var: :return:
SCons.Defaults._libversionflags (env, version_var, flags_var)
if version_var is not empty, returns env[flags_var], otherwise returns None :param env: :param version_var: :param flags_var: :return:
SCons.Defaults._concat (prefix, items_iter, suffix, env, f=<function <lambda>>, target=None, source=None, affect_signature: bool = True)
Creates a new list from 'items_iter' by first interpolating each element in the list using the 'env' dictionary and then calling f on the list, and finally calling _concat_ixes to concatenate 'prefix' and 'suffix' onto each element of the list.
SCons.Defaults._concat_ixes (prefix, items_iter, suffix, env)
Creates a new list from 'items_iter' by concatenating the 'prefix' and 'suffix' arguments onto each element of the list.
A trailing space on 'prefix' or leading space on 'suffix' will cause them to be put into separate list elements rather than being concatenated.
SCons.Defaults._defines (prefix,defs, suffix, env, target=None, source=None, c=<function _concat_ixes>)
A wrapper around _concat_ixes() that turns a list or string into a list of C preprocessor command-line definitions.
SCons.Defaults._fetch_DefaultEnvironment (*args, **kwargs)
Returns the already-created default construction environment.
SCons.Defaults._stripixes (prefix: str, items, suffix: str, stripsuffixes: List[str], stripprefixes: List[str], env, literal_prefix: str = '', c: Callable[[list], list] = None) → list
Returns a list with text added to items after first stripping them.
A companion to _concat_ixes(), used by tools (like the GNU linker) that need to turn something like `libfoo.a` into `-lfoo`. stripprefixes and stripsuffixes are stripped from items. Calls function c to postprocess the result.
Parameters:
- **prefix** – string to prepend to elements
- **items** – string or iterable to transform
- **suffix** – string to append to elements
- **stripprefixes** – prefix string(s) to strip from elements
- **stripsuffixes** – suffix string(s) to strip from elements
- **env** – construction environment for variable interpolation
- **c** – optional function to perform a transformation on the list. The default is `None`, which will select `_concat_ixes()`.

SCons.Defaults.chmod_func (dest, mode) → None
Implementation of the Chmod action function.

mode can be either an integer (normally expressed in octal mode, as in 0o755) or a string following the syntax of the POSIX chmod command (for example “ugo+w”). The latter must be converted, since the underlying Python only takes the numeric form.

SCons.Defaults.chmod_strfunc (dest, mode) → str
strfunction for the Chmod action function.

SCons.Defaults.copy_func (dest, src, symlinks: bool = True) → int
Implementation of the Copy action function.

Copies src to dest. If src is a list, dest must be a directory, or not exist (will be created). Since Python shiutil methods, which know nothing about SCons Nodes, will be called to perform the actual copying, args are converted to strings first.

If symlinks evaluates true, then a symbolic link will be shallow copied and recreated as a symbolic link; otherwise, copying a symbolic link will be equivalent to copying the symbolic link’s final target regardless of symbolic link depth.

SCons.Defaults.copy_strfunc (dest, src, symlinks: bool = True) → str
strfunction for the Copy action function.

SCons.Defaults.delete_func (dest, must_exist: bool = False) → None
Implementation of the Delete action function.

Lets the Python os.unlink() raise an error if dest does not exist, unless must_exist evaluates false (the default).

SCons.Defaults.delete_strfunc (dest, must_exist: bool = False) → str
strfunction for the Delete action function.

SCons.Defaults.get_paths_str (dest) → str
Generates a string from dest for use in a strfunction.

If dest is a list, manually converts each elem to a string.

SCons.Defaults.mkdir_func (dest) → None
Implementation of the Mkdir action function.

SCons.Defaults.move_func (dest, src) → None
Implementation of the Move action function.

SCons.Defaults.processDefines (defs) → List[str]
Return list of strings for preprocessor defines from defs.

Resolves the different forms CPPDEFINES can be assembled in: if the Append/Prepend routines are used beyond a initial setting it will be a deque, but if written to only once (Environment initializer, or direct write) it can be a multitude of types.

Any prefix/suffix is handled elsewhere (usually _concat_ixes()).

Changed in version 4.5.0: Bare tuples are now treated the same as tuple-in-sequence, assumed to describe a valued macro. Bare strings are now split on space. A dictionary is no longer sorted before handling.

SCons.Defaults.touch_func (dest) → None
Implementation of the Touch action function.

SCons.Environment module

Base class for construction Environments.

These are the primary objects used to communicate dependency and construction information to the build engine.
Keyword arguments supplied when the construction Environment is created are construction variables used to initialize the Environment.

```python
class SCons.Environment.Base (platform=None, tools=None, toolpath=None, variables=None, parse_flags=None, **kw):
    Bases: SubstitutionEnvironment
    Base class for “real” construction Environments.
    These are the primary objects used to communicate dependency and construction information to the build engine.
    Keyword arguments supplied when the construction Environment is created are construction variables used to initialize the Environment.
    Action(*args, **kw)
    AddMethod (function, name=None) → None
        Adds the specified function as a method of this construction environment with the specified name. If the name is omitted, the default name is the name of the function itself.
    AddPostAction (files, action)
    AddPreAction (files, action)
    Alias (target, source=[], action=None, **kw)
    AlwaysBuild (*targets)
    Append (**kw) → None
        Append values to construction variables in an Environment.
        The variable is created if it is not already present.
    AppendENVPath (name, newpath, envname: str = 'ENV', sep=':', delete_existing: bool = False) → None
        Append path elements to the path name in the envname dictionary for this environment. Will only add any particular path once, and will normpath and normcase all paths to help assure this. This can also handle the case where the env variable is a list instead of a string.
        If delete_existing is False, a newpath element already in the path will not be moved to the end (it will be left where it is).
    AppendUnique (delete_existing: bool = False, **kw) → None
        Append values to existing construction variables in an Environment, if they’re not already there. If delete_existing is True, removes existing values first, so values move to end.
    Builder (**kw)
    CacheDir (path, custom_class=None) → None
    Clean (targets, files) → None
    Clone (tools=[], toolpath=None, parse_flags=None, **kw)
        Return a copy of a construction Environment.
        The copy is like a Python “deep copy”—that is, independent copies are made recursively of each objects—except that a reference is copied when an object is not deep-copyable (like a function). There are no references to any mutable objects in the original Environment.
    Command (*args, **kw)
    Decider (function)
    Depends (target, dependency)
        Explicitly specify that ‘target’s depend on ‘dependency’.
    Detect (progs)
        Return the first available program from one or more possibilities.

        Parameters: progs (str or list) – one or more command names to check for
        Dictionary (*args)
        Return construction variables from an environment.

        Parameters: *args (optional) – variable names to look up
        Returns: If args omitted, the dictionary of all construction variables. If one arg, the corresponding value is returned. If more than one arg, a list of values is returned.
        Raises: KeyError – if any of args is not in the construction environment.
```
SCons API Documentation

Dir (name, *args, **kw)
Dump (key=None, format: str = 'pretty')
  Return construction variables serialized to a string.

  Parameters:
  - key (optional) – if None, format the whole dict of variables. Else format the value of key
    (Default value = None)
  - format (str, optional) – specify the format to serialize to. “pretty” generates a
    pretty-printed string, “json” a JSON-formatted string. (Default value = “pretty”)

Entry (name, *args, **kw)
Environment (**kw)
Execute (action, *args, **kw)
  Directly execute an action through an Environment
File (name, *args, **kw)
FindFile (file, dirs)
FindInstalledFiles ()
  returns the list of all targets of the Install and InstallAs Builder.
FindIxes (paths, prefix, suffix)
  Search a list of paths for something that matches the prefix and suffix.

  Parameters:
  - paths – the list of paths or nodes.
  - prefix – construction variable for the prefix.
  - suffix – construction variable for the suffix.

  Returns: the matched path or None
FindSourceFiles (node: str = '.') → list
  Return a list of all source files.
Flatten (sequence)
GetBuildPath (files)
Glob (pattern, ondisk: bool = True, source: bool = False, strings: bool = False, exclude=None)
Ignore (target, dependency)
  Ignore a dependency.
Literal (string)
Local (*targets)
MergeFlags (args, unique: bool = True) → None
  Merge flags into construction variables.
  Merges the flags from args into this construction environent. If args is not a dict, it is first converted to one with
  flags distributed into appropriate construction variables. See ParseFlags().
  As a side effect, if unique is true, a new object is created for each modified construction variable by the loop at the end. This is silently expected by the Override() parse_flags functionality, which does not want to share the list (or
  whatever) with the environment being overridden.

  Parameters:
  - args – flags to merge
  - unique – merge flags rather than appending (default: True). When merging, path
    variables are retained from the front, other construction variables from the end.

NoCache (*targets)
  Tags a target so that it will not be cached
NoClean (*targets)
  Tags a target so that it will not be cleaned by -c
Override (overrides)
  Produce a modified environment whose variables are overridden by the overrides dictionaries. “overrides” is a
  dictionary that will override the variables of this environment.
  This function is much more efficient than Clone() or creating a new Environment because it doesn’t copy the
  construction environment dictionary, it just wraps the underlying construction environment, and doesn’t even create
  a wrapper object if there are no overrides.
ParseConfig (command, function=None, unique: bool = True)
Parse the result of running a command to update construction vars.
Use function to parse the output of running command in order to modify the current environment.

Parameters:
- command – a string or a list of strings representing a command and its arguments.
- function – called to process the result of command, which will be passed as args. If function is omitted or None, MergeFlags() is used. Takes 3 args (env, args, unique)
- unique – whether no duplicate values are allowed (default true)

ParseDepends (filename, must_exist=None, only_one: bool = False)
Parse a mkdep-style file for explicit dependencies. This is completely abusable, and should be unnecessary in the “normal” case of proper SCons configuration, but it may help make the transition from a Make hierarchy easier for some people to swallow. It can also be genuinely useful when using a tool that can write a .d file, but for which writing a scanner would be too complicated.

ParseFlags (**flags) → dict
Return a dict of parsed flags.
Parse flags and return a dict with the flags distributed into the appropriate construction variable names. The flags are treated as a typical set of command-line flags for a GNU-style toolchain, such as might have been generated by one of the {foo}-config scripts, and used to populate the entries based on knowledge embedded in this method - the choices are not expected to be portable to other toolchains.
If one of the flags strings begins with a bang (exclamation mark), it is assumed to be a command and the rest of the string is executed; the result of that evaluation is then added to the dict.

Platform (platform)
Precious (**targets)
Mark targets as precious: do not delete before building.
Prepend (**kw) → None
Prepend values to construction variables in an Environment.
The variable is created if it is not already present.
PrependENVPath (name, newpath, envname: str = 'ENV', sep=':', delete_existing: bool = True) → None
Prepend path elements to the path name in the envname dictionary for this environment. Will only add any particular path once, and will normpath and normcase all paths to help assure this. This can also handle the case where the env variable is a list instead of a string.
If delete_existing is False, a newpath component already in the path will not be moved to the front (it will be left where it is).
PrependUnique (delete_existing: bool = False, **kw) → None
Prepend values to existing construction variables in an Environment, if they’re not already there. If delete_existing is True, removes existing values first, so values move to front.

Pseudo (**targets)
Mark targets as pseudo: must not exist.

PyPackageDir (modulename)
RemoveMethod (function) → None
Removes the specified function’s MethodWrapper from the added_methods list, so we don’t re-bind it when making a clone.
Replace (**kw) → None
Replace existing construction variables in an Environment with new construction variables and/or values.
ReplaceIxes (path, old_prefix, old_suffix, new_prefix, new_suffix)
Replace old_prefix with new_prefix and old_suffix with new_suffix.
env - Environment used to interpolate variables. path - the path that will be modified. old_prefix - construction variable for the old prefix. old_suffix - construction variable for the old suffix. new_prefix - construction variable for the new prefix. new_suffix - construction variable for the new suffix.
Repository (**dirs, **kw) → None
Specify Repository directories to search.
Requires (target, prerequisite)
Specify that prerequisite must be built before target.
SCons API Documentation

Creates an order-only relationship, not a full dependency. **prerequisite** must exist before **target** can be built, but a change to **prerequisite** does not trigger a rebuild of **target**.

*SConsignFile* (*name*='.sconsign', *dbm_module*=None) → None

*SConsignFile* is used to store a sconsign file. This file is used to store the meta data of a build. It is not directly accessed by the user.

*Scanner* (*args*, **kw*)

A scanner is a function that is given a list of args. It can do anything it wants, but it must return something.

*SetDefault* (**kw*)

*SConsignFile* sets the default values for the arguments passed to *SConsignFile*.

*SideEffect* (*side_effect*, **target*)

Tell scons that side_effects are built as side effects of building targets.

*Split* (*arg*)

This function converts a string or list into a list of strings or Nodes. This makes things easier for users by allowing files to be specified as a white-space separated list to be split.

The input rules are:

- A single string containing names separated by spaces. These will be split apart at the spaces.
- A single Node instance
- A list containing either strings or Node instances. Any strings in the list are not split at spaces.

In all cases, the function returns a list of Nodes and strings.

*Tool* (*tool*, *toolpath*=None, **kwargs*) → Tool

Find and run tool module *tool*.

*Value* (*value*, *built_value*=None, **name*)

Return a Value (Python expression) node.

*VariantDir* (*variant_dir*, *src_dir*, *duplicate*: *int* = 1) → None

*VariantDir* is used to specify the source directory and variant directory.

*Whereis* (*prog*, *path*=None, *pathext*=None, *reject*=None)

*Whereis* is used to find a program in a path.

The following are some auxiliary methods:

- **_canonicalize** (*path*)
- **_init_special** ()
- **_changed_build** (*dependency*, *target*, *prev_ni*, *repo_node*=None) → bool
- **_changed_content** (*dependency*, *target*, *prev_ni*, *repo_node*=None) → bool
- **_changed_timestamp_match** (*dependency*, *target*, *prev_ni*, *repo_node*=None) → bool
- **_changed_timestamp_newer** (*dependency*, *target*, *prev_ni*, *repo_node*=None) → bool
- **_changed_timestamp_then_content** (*dependency*, *target*, *prev_ni*, *repo_node*=None) → bool
- **_find_toolpath_dir** (*tp*)
- **_gsm** ()
- **_update** (*other*) → None
- **_update_onlynew** (*other*) → None

*arg2nodes* (*args*, *node_factory*=<class 'SCons.Environment._Null'>, *lookup_list*=<class 'SCons.Environment._Null'>, **kw*)

Converts *args* to a list of nodes.
Parameters:

- **just** *(args - filename strings or nodes to convert; nodes are)* – added to the list without further processing.
- **not** *(node_factory - optional factory to create the nodes; if)* – specified, will use this environment’s `fs.File` method.
- **to** *(lookup_list - optional list of lookup functions to call)* – attempt to find the file referenced by each args.
- **add.**(kw - keyword arguments that represent additional nodes to)* –

backtick *(command) → str*

Emulate command substitution.

Provides behavior conceptually like POSIX Shell notation for running a command in backquotes (backticks) by running `command` and returning the resulting output string.

This is not really a public API any longer, it is provided for the use of ParseFlags() (which supports it using a syntax of !command) and ParseConfig().

**Raises:** **OSError** – if the external command returned non-zero exit status.

get *(key, default=None)*

Emulates the get() method of dictionaries.

get_CacheDir ()

get_builder (name)

Fetch the builder with the specified name from the environment.

get_factory *(factory, default: str = 'File')*

Return a factory function for creating Nodes for this construction environment.

g_get_scanner *(skey)*

Find the appropriate scanner given a key (usually a file suffix).

gvars ()

Emulates the items() method of dictionaries.

items ()

Emulates the keys() method of dictionaries.

kvars ()

Emulates the setdefault() method of dictionaries.

scanner_map_delete *(kw=None) → None*

Delete the cached scanner map (if we need to).

setdefault *(key, default=None)*

Emulates the setdefault() method of dictionaries.

subst *(string, raw: int = 0, target=None, source=None, conv=None, executor: Executor | None = None, overrides: dict | None = None)*

Recursively interpolates construction variables from the Environment into the specified string, returning the expanded result. Construction variables are specified by a $ prefix in the string and begin with an initial underscore or alphabetic character followed by any number of underscores or alphanumeric characters. The construction variable names may be surrounded by curly braces to separate the name from trailing characters.

**Note:**

Recursively interpolates construction variables from the Environment into the specified string, returning the expanded result. Construction variables are specified by a $ prefix in the string and begin with an initial underscore or alphabetic character followed by any number of underscores or alphanumeric characters. The construction variable names may be surrounded by curly braces to separate the name from trailing characters.

**Note:**

Recursively interpolates construction variables from the Environment into the specified string, returning the expanded result. Construction variables are specified by a $ prefix in the string and begin with an initial underscore or alphabetic character followed by any number of underscores or alphanumeric characters. The construction variable names may be surrounded by curly braces to separate the name from trailing characters.
Validate the passed custom CacheDir class, or if no args are passed, validate the custom CacheDir class from the environment.

values()  
Emulates the values() method of dictionaries.

```python
class SCons.Environment.BuilderDict(mapping, env)
```

Bases: UserDict

This is a dictionary-like class used by an Environment to hold the Builders. We need to do this because every time someone changes the Builders in the Environment’s BUILDERS dictionary, we must update the Environment’s attributes.

```python
class SCons.Environment.BuilderWrapper(obj: Any, method: Callable, name: str | None = None)
```

Bases: MethodWrapper

A MethodWrapper subclass that that associates an environment with a Builder.

This mainly exists to wrap the __call__() function so that all calls to Builders can have their argument lists massaged in the same way (treat a lone argument as the source, treat two arguments as target then source, make sure both target and source are lists) without having to have cut-and-paste code to do it.

As a bit of obsessive backwards compatibility, we also intercept attempts to get or set the “env” or “builder” attributes, which were the names we used before we put the common functionality into the MethodWrapper base class. We’ll keep this around for a while in case people shipped Tool modules that reached into the wrapper (like the Tool/qt.py module does, or did). There shouldn’t be a lot attribute fetching or setting on these, so a little extra work shouldn’t hurt.

```python
clone(new_object)
```

Returns an object that re-binds the underlying “method” to the specified new object.

```python
SCons.Environment.NoSubstitutionProxy(subject)
```

An entry point for returning a proxy subclass instance that overrides the subst*() methods so they don’t actually perform construction variable substitution. This is specifically intended to be the shim layer in between global function calls (which don’t want construction variable substitution) and the DefaultEnvironment() (which would substitute variables if left to its own devices).

We have to wrap this in a function that allows us to delay definition of the class until it’s necessary, so that when it subclasses Environment it will pick up whatever Environment subclass the wrapper interface might have assigned to SCons.Environment.Environment.

```python
class SCons.Environment.OverrideEnvironment(subject, overrides=None)
```

Bases: Base

A proxy that overrides variables in a wrapped construction environment by returning values from an overrides dictionary in preference to values from the underlying subject environment.

This is a lightweight (I hope) proxy that passes through most use of attributes to the underlying Environment.Base class, but has just enough additional methods defined to act like a real construction environment with overridden values. It can wrap either a Base construction environment, or another OverrideEnvironment, which can in turn nest arbitrary OverrideEnvironments...
Note that we do not call the underlying base class (SubstitutionEnvironment) initialization, because we get most of those from proxying the attributes of the subject construction environment. But because we subclass SubstitutionEnvironment, this class also has inherited arg2nodes() and subst*() methods; those methods can’t be proxied because they need this object’s methods to fetch the values from the overrides dictionary.

**Action** (*args, **kw*)

**AddMethod** (function, name=None) → None

Adds the specified function as a method of this construction environment with the specified name. If the name is omitted, the default name is the name of the function itself.

**AddPostAction** (files, action)

**AddPreAction** (files, action)

**Alias** (target, source=[], action=None, **kw)**

**AlwaysBuild** (*targets*)

**Append** (**kw**) → None

Append values to construction variables in an Environment.

The variable is created if it is not already present.

**AppendENVPath** (name, newpath, envname: str = ‘ENV’, sep=‘:’, delete_existing: bool = False) → None

Append path elements to the path name in the envname dictionary for this environment. Will only add any particular path once, and will normpath and normcase all paths to help assure this. This can also handle the case where the env variable is a list instead of a string.

If delete_existing is False, a newpath element already in the path will not be moved to the end (it will be left where it is).

**AppendUnique** (delete_existing: bool = False, **kw**) → None

Append values to existing construction variables in an Environment, if they’re not already there. If delete_existing is True, removes existing values first, so values move to end.

**Builder** (**kw**) → None

Returns a copy of a construction Environment.

The copy is like a Python “deep copy”–that is, independent copies are made recursively of each objects–except that a reference is copied when an object is not deep-copyable (like a function). There are no references to any mutable objects in the original Environment.

**Configure** (*args, **kw*)

Returns the first available program from one or more possibilities.

**Parameters:** progs (*str or list*) – one or more command names to check for

**Dictionary** (*args*)

Returns construction variables from an environment.

**Parameters:** *args (optional) – variable names to look up

**Returns:** If args omitted, the dictionary of all construction variables. If one arg, the corresponding value is returned. If more than one arg, a list of values is returned.

**Raises:** KeyError – if any of args is not in the construction environment.

**Dir** (name, *args, **kw*)

**Dump** (key=None, format: str = ‘pretty’) → None

Return construction variables serialized to a string.
Parameters:

- **key** *(optional)* – if None, format the whole dict of variables. Else format the value of key
  (Default value = None)

- **format** *(str, optional)* – specify the format to serialize to. “pretty” generates a
  pretty-printed string, “json” a JSON-formatted string. (Default value = “pretty”)

Entry *(name, *args, **kw)*
Environment *(**kw)*
Execute *(action, *args, **kw)*
  Directly execute an action through an Environment
File *(name, *args, **kw)*
FindFile *(file, dirs)*
FindInstalledFiles *
  returns the list of all targets of the Install and InstallAs Builder.
FindIxes *(paths, prefix, suffix)*
  Search a list of paths for something that matches the prefix and suffix.

Parameters:

- **paths** – the list of paths or nodes.
- **prefix** – construction variable for the prefix.
- **suffix** – construction variable for the suffix.

Returns: the matched path or None

FindSourceFiles *(node: str = '.') → list*
  Return a list of all source files.
Flatten *(sequence)*
GetBuildPath *(files)*
Glob *(pattern, ondisk: bool = True, source: bool = False, strings: bool = False, exclude=None)*
Ignore *(target, dependency)*
  Ignore a dependency.
Literal *(string)*
Local *(*targets)*
MergeFlags *(args, unique: bool = True) → None*
  Merge flags into construction variables.
  Merges the flags from *args* into this construction environment. If *args* is not a dict, it is first converted to one with
  flags distributed into appropriate construction variables. See ParseFlags().
  As a side effect, if *unique* is true, a new object is created for each modified construction variable by the loop at the
  end. This is silently expected by the Override() *parse_flags* functionality, which does not want to share the list (or
  whatever) with the environment being overridden.

Parameters:

- **args** – flags to merge
- **unique** – merge flags rather than appending (default: True). When merging, path
  variables are retained from the front, other construction variables from the end.

NoCache *(*targets)*
  Tags a target so that it will not be cached
NoClean *(*targets)*
  Tags a target so that it will not be cleaned by -c
Override *(overrides)*
  Produce a modified environment whose variables are overridden by the overrides dictionaries. “overrides” is a
  dictionary that will override the variables of this environment.
  This function is much more efficient than Clone() or creating a new Environment because it doesn’t copy the
  construction environment dictionary, it just wraps the underlying construction environment, and doesn’t even create
  a wrapper object if there are no overrides.
ParseConfig *(command, function=None, unique: bool = True)*
  Parse the result of running a command to update construction vars.
  Use function to parse the output of running command in order to modify the current environment.
Parameters:

- **command** – a string or a list of strings representing a command and its arguments.
- **function** – called to process the result of command, which will be passed as args. If function is omitted or None, MergeFlags() is used. Takes 3 args (env, args, unique)
- **unique** – whether no duplicate values are allowed (default true)

ParseDepends (filename, must_exist=None, only_one: bool = False)
Parse a mkdep-style file for explicit dependencies. This is completely abusable, and should be unnecessary in the “normal” case of proper SCons configuration, but it may help make the transition from a Make hierarchy easier for some people to swallow. It can also be genuinely useful when using a tool that can write a .d file, but for which writing a scanner would be too complicated.

ParseFlags (*flags) → dict
Return a dict of parsed flags.
Parse flags and return a dict with the flags distributed into the appropriate construction variable names. The flags are treated as a typical set of command-line flags for a GNU-style toolchain, such as might have been generated by one of the (foo)-config scripts, and used to populate the entries based on knowledge embedded in this method - the choices are not expected to be portable to other toolchains.
If one of the flags strings begins with a bang (exclamation mark), it is assumed to be a command and the rest of the string is executed; the result of that evaluation is then added to the dict.

Platform (platform)
Precious (*targets)
Mark targets as precious: do not delete before building.
Prepend (**kw) → None
Prepend values to construction variables in an Environment.
The variable is created if it is not already present.
PrependENVPath (name, newpath, envname: str = 'ENV', sep=':', delete_existing: bool = True) → None
Prepend path elements to the path name in the envname dictionary for this environment. Will only add any particular path once, and will normpath and normcase all paths to help assure this. This can also handle the case where the env variable is a list instead of a string.
If delete_existing is False, a newpath component already in the path will not be moved to the front (it will be left where it is).
PrependUnique (delete_existing: bool = False, **kw) → None
Prepend values to existing construction variables in an Environment, if they’re not already there. If delete_existing is True, removes existing values first, so values move to front.

Pseudo (*targets)
Mark targets as pseudo: must not exist.

PyPackageDir (modulename)
RemoveMethod (function) → None
Removes the specified function’s MethodWrapper from the added_methods list, so we don’t re-bind it when making a clone.

Replace (**kw) → None
Replace existing construction variables in an Environment with new construction variables and/or values.
ReplaceIxes (path, old_prefix, old_suffix, new_prefix, new_suffix)
Replace old_prefix with new_prefix and old_suffix with new_suffix.
env - Environment used to interpolate variables. path - the path that will be modified. old_prefix - construction variable for the old prefix. old_suffix - construction variable for the old suffix. new_prefix - construction variable for the new prefix. new_suffix - construction variable for the new suffix.
Repository (*dirs, **kw) → None
Specify Repository directories to search.

Requires (target, prerequisite)
Specify that prerequisite must be built before target.
Creates an order-only relationship, not a full dependency. prerequisite must exist before target can be built, but a change to prerequisite does not trigger a rebuild of target.
SConsignFile (name='.sconsign', dbm_module=None) → None
Scanner(*args, **kw)
SetDefault(**kw) → None
SideEffect(side_effect, target)
    Tell scons that side_effects are built as side effects of building targets.
Split(arg)
    This function converts a string or list into a list of strings or Nodes. This makes things easier for users by allowing files to be specified as a white-space separated list to be split.

The input rules are:

- A single string containing names separated by spaces. These will be split apart at the spaces.
- A single Node instance
- A list containing either strings or Node instances. Any strings in the list are not split at spaces.

In all cases, the function returns a list of Nodes and strings.

Tool(tool, toolpath=None, **kwargs) → Tool
    Find and run tool module tool.
    Changed in version 4.2: returns the tool module rather than None.

Value(value, built_value=None, name=None)
    Return a Value (Python expression) node.
    Changed in version 4.0: the name parameter was added.

VariantDir(variant_dir, src_dir, duplicate: int = 1) → None

WhereIs(prog, path=None, pathext=None, reject=None)
    Find prog in the path.

canonicalize(path)
    Allow Dirs and strings beginning with # for top-relative.
    Note this uses the current env's fs (in self).

_changed_build(dependency, target, prev_ni, repo_node=None) → bool
_changed_content(dependency, target, prev_ni, repo_node=None) → bool
_changed_timestamp_match(dependency, target, prev_ni, repo_node=None) → bool
_changed_timestamp_newer(dependency, target, prev_ni, repo_node=None) → bool
_changed_timestamp_then_content(dependency, target, prev_ni, repo_node=None) → bool
_find_toolpath_dir(tp)
_gsm()
_init_special() → None
    Initial the dispatch tables for special handling of special construction variables.
_update(other) → None
    Private method to update an environment's consvar dict directly.
    Bypasses the normal checks that occur when users try to set items.
_update_onlynew(other) → None
    Update a dict with new keys.
    Unlike the .update method, if the key is already present, it is not replaced.
arg2nodes(args, node_factory=<class 'SCons.Environment._Null'>, lookup_list=<class 'SCons.Environment._Null'>, **kw)
    Converts args to a list of nodes.

Parameters:

- **just** *(args - filename strings or nodes to convert; nodes are) – added to the list without further processing.*
- **not** *(node_factory - optional factory to create the nodes; if) – specified, will use this environment's `fs.File method.*
- **to** *(lookup_list - optional list of lookup functions to call) – attempt to find the file referenced by each args.*
- **add.** *(kw - keyword arguments that represent additional nodes to) –

backtick(command) → str
    Emulate command substitution.
Provides behavior conceptually like POSIX Shell notation for running a command in backquotes (backticks) by running `command` and returning the resulting output string.

This is not really a public API any longer, it is provided for the use of ParseFlags() (which supports it using a syntax of `!command`) and ParseConfig().

**Raises:**  
`OSError` – if the external command returned non-zero exit status.

- `get(key, default=None)`  
  Emulates the `get()` method of dictionaries.

- `get_CacheDir()`  
  Fetch the builder with the specified name from the environment.

- `get_factory(factory, default: str = 'File')`  
  Return a factory function for creating Nodes for this construction environment.

- `get_scanner(skey)`  
  Find the appropriate scanner given a key (usually a file suffix).

- `gvars()`  
  Emulates the `items()` method of dictionaries.

- `getkeys()`  
  Emulates the `keys()` method of dictionaries.

- `lvars()`  
  Emulates the `values()` method of dictionaries.

- `scanner_map_delete(kw=None) → None`  
  Delete the cached scanner map (if we need to).

- `setdefault(key, default=None)`  
  Emulates the `setdefault()` method of dictionaries.

- `subst(string, raw: int = 0, target=None, source=None, conv=None, executor: Executor | None = None, overrides: dict | None = None)`  
  Recursively interpolates construction variables from the Environment into the specified string, returning the expanded result. Construction variables are specified by a $ prefix in the string and begin with an initial underscore or alphabetic character followed by any number of underscores or alphanumeric characters. The construction variable names may be surrounded by curly braces to separate the name from trailing characters.

- `validate_CacheDir_class(custom_class=None)`  
  Validate the passed custom CacheDir class, or if no args are passed, validate the custom CacheDir class from the environment.

- `values()`  
  Emulates the `values()` method of dictionaries.

**class SCons.Environment.SubstitutionEnvironment(**kw)**

**Bases:** object

*Base class for different flavors of construction environments.*

This class contains a minimal set of methods that handle construction variable expansion and conversion of strings to Nodes, which may or may not be actually useful as a stand-alone class. Which methods ended up in this class is pretty arbitrary right now. They're basically the ones which we've empirically determined are common to the different
construction environment subclasses, and most of the others that use or touch the underlying dictionary of
construction variables.
Eventually, this class should contain all the methods that we determine are necessary for a “minimal” interface to the
build engine. A full “native Python” SCons environment has gotten pretty heavyweight with all of the methods and
Tools and construction variables we’ve jammed in there, so it would be nice to have a lighter weight alternative for
interfaces that don’t need all of the bells and whistles. (At some point, we’ll also probably rename this class “Base,”
since that more reflects what we want this class to become, but because we’ve released comments that tell people to
subclass Environment.Base to create their own flavors of construction environment, we’ll save that for a future
refactoring when this class actually becomes useful.)

AddMethod (function, name=None) → None
Adds the specified function as a method of this construction environment with the specified name. If the name is
omitted, the default name is the name of the function itself.

MergeFlags (args, unique: bool = True) → None
Merge flags into construction variables.
Merges the flags from args into this construction environment. If args is not a dict, it is first converted to one with
flags distributed into appropriate construction variables. See ParseFlags().
As a side effect, if unique is true, a new object is created for each modified construction variable by the loop at the
end. This is silently expected by the Override() parse_flags functionality, which does not want to share the list (or
whatever) with the environment being overridden.

Parameters:
- args – flags to merge
- unique – merge flags rather than appending (default: True). When merging, path
  variables are retained from the front, other construction variables from the end.

Override (overrides)
Produce a modified environment whose variables are overridden by the overrides dictionaries. “overrides” is a
dictionary that will override the variables of this environment.
This function is much more efficient than Clone() or creating a new Environment because it doesn’t copy the
construction environment dictionary, it just wraps the underlying construction environment, and doesn’t even create
a wrapper object if there are no overrides.

ParseFlags (*flags) → dict
Return a dict of parsed flags.
Parse flags and return a dict with the flags distributed into the appropriate construction variable names. The flags
are treated as a typical set of command-line flags for a GNU-style toolchain, such as might have been generated
by one of the (foo)-config scripts, and used to populate the entries based on knowledge embedded in this method -
the choices are not expected to be portable to other toolchains.
If one of the flags strings begins with a bang (exclamation mark), it is assumed to be a command and the rest of
the string is executed; the result of that evaluation is then added to the dict.

RemoveMethod (function) → None
Removes the specified function’s MethodWrapper from the added_methods list, so we don’t re-bind it when
making a clone.

_init_special () → None
Initial the dispatch tables for special handling of special construction variables.

arg2nodes (args, node_factory=<class 'SCons.Environment._Null'>, lookup_list=<class
'SCons.Environment._Null'>, **kw)
Converts args to a list of nodes.

Parameters:
- just (args - filename strings or nodes to convert; nodes are) – added to the list without
  further processing.
- not (node_factory - optional factory to create the nodes; if) – specified, will use this
  environment’s ”fs.File method.
- to (lookup_list - optional list of lookup functions to call) – attempt to find the file
  referenced by each args.
- add (kw - keyword arguments that represent additional nodes to) –

backtick (command) → str
Emulate command substitution.
Provides behavior conceptually like POSIX Shell notation for running a command in backquotes (backticks) by running `command` and returning the resulting output string.
This is not really a public API any longer, it is provided for the use of ParseFlags() (which supports it using a syntax of `!command`) and ParseConfig().

**Raises:** OSError – if the external command returned non-zero exit status.

```python
get (key, default=None)
Emulates the get() method of dictionaries.
gvars ()
Emulates the items() method of dictionaries.
keys ()
Emulates the keys() method of dictionaries.
lvars ()
```

```python
setdefault (key, default=None)
Emulates the setdefault() method of dictionaries.
```

```python
subst (string, raw: int = 0, target=None, source=None, conv=None, executor: Executor | None = None, overrides: dict | None = None)
Recursively interpolates construction variables from the Environment into the specified string, returning the expanded result. Construction variables are specified by a $ prefix in the string and begin with an initial underscore or alphabetic character followed by any number of underscores or alphanumeric characters. The construction variable names may be surrounded by curly braces to separate the name from trailing characters.
```

```python
subst_kw (kw, raw: int = 0, target=None, source=None)
```

```python
subst_list (string, raw: int = 0, target=None, source=None, conv=None, executor: Executor | None = None, overrides: dict | None = None)
Calls through to SCons.Subst.scons_subst_list().
See the documentation for that function.
```

```python
subst_path (path, target=None, source=None)
Substitute a path list.
```

```python
subst_target_source (string, raw: int = 0, target=None, source=None, conv=None, executor: Executor | None = None, overrides: dict | None = None)
Recursively interpolates construction variables from the Environment into the specified string, returning the expanded result. Construction variables are specified by a $ prefix in the string and begin with an initial underscore or alphabetic character followed by any number of underscores or alphanumeric characters. The construction variable names may be surrounded by curly braces to separate the name from trailing characters.
```

```python
values ()
Emulates the values() method of dictionaries.
```

```python
class SCons.Environment._Null
Bases: object
SCons.Environment._add_cppdefines (env_dict: dict, val, prepend: bool = False, unique: bool = False, delete_existing: bool = False) \rightarrow None
Adds to CPPDEFINES, using the rules for C preprocessor macros.
This is split out from regular construction variable addition because these entries can express either a macro with a replacement value or one without. A macro with replacement value can be supplied as `val` in three ways: as a combined string "name=value"; as a tuple (name, value), or as an entry in a dictionary ("name": value). A list argument with multiple macros can also be given.
Additions can be unconditional (duplicates allowed) or uniquing (no dupes).
Note if a replacement value is supplied, `unique` requires a full match to decide uniqueness - both the macro name and the replacement. The inner `_is_in()` is used to figure that out.
```
**Parameters:**

- `env_dict` – the dictionary containing the `CPPDEFINES` to be modified.
- `val` – the value to add, can be string, sequence or dict
- `prepend` – whether to put `val` in front or back.
- `unique` – whether to add `val` if it already exists.
- `delete_existing` – if `unique` is true, add `val` after removing previous.

New in version 4.5.0.

SCons.Environment._del_SCANNERS (env, key) → None
SCons.Environment._delete_duplicates (l, keep_last)
Delete duplicates from a sequence, keeping the first or last.
SCons.Environment._null
  alias of _Null
SCons.Environment._set_BUILDERS (env, key, value)
SCons.Environment._set_SCANNERS (env, key, value) → None
SCons.Environment._set_future_reserved (env, key, value) → None
SCons.Environment._set_reserved (env, key, value) → None
SCons.Environment.alias_builder (env, target, source) → None
SCons.Environment.apply_tools (env, tools, toolpath) → None
SCons.Environment.copy_non_reserved_keywords (dict)
SCons.Environment.default_copy_from_cache (env, src, dst)
SCons.Environment.default_copy_to_cache (env, src, dst)
SCons.Environment.default_decide_source (dependency, target, prev_ni, repo_node=None)
SCons.Environment.default_decide_target (dependency, target, prev_ni, repo_node=None)
SCons.Environment.is_valid_construction_var (varstr) → bool
Return True if `varstr` is a legitimate construction variable.

SCons.Errors module
SCons exception classes.

**exception** SCons.Errors.BuildError (node=None, errstr: str = 'Unknown error', status: int = 2, exitstatus: int = 2, filename=None, executor: SCons.Executor.Executor | None = None, action=None, command=None, exc_info=(None, None, None))
Bases: Exception
SCons Errors that can occur while building.
A BuildError exception contains information both about the error itself, and what caused the error.
Variables:

- **node** – *(cause)* the error occurred while building this target node(s)
- **errstr** – *(info)* a description of the error message
- **status** – *(info)* the return code of the action that caused the build error. Must be set to a non-zero value even if the build error is not due to an action returning a non-zero returned code.
- **exitstatus** – *(info)* SCons exit status due to this build error. Must be nonzero unless due to an explicit Exit() call. Not always the same as status, since actions return a status code that should be respected, but SCons typically exits with 2 irrespective of the return value of the failed action.
- **filename** – *(info)* The name of the file or directory that caused the build error. Set to None if no files are associated with this error. This might be different from the target being built. For example, failure to create the directory in which the target file will appear. It can be None if the error is not due to a particular filename.
- **executor** – *(cause)* the executor that caused the build to fail (might be None if the build failures is not due to the executor failing)
- **action** – *(cause)* the action that caused the build to fail (might be None if the build failures is not due to the an action failure)
- **command** – *(cause)* the command line for the action that caused the build to fail (might be None if the build failures is not due to the an action failure)
- **exc_info** – *(info)* Info about exception that caused the build error. Set to (None, None, None) if this build error is not due to an exception.

### Exception Classes

- **SCons.Errors.ExplicitExit** *(node=*, status=*, *args)*
  - Bases: Exception
- **SCons.Errors.InternalError**
  - Bases: Exception
- **SCons.Errors.MSVCError**
  - Bases: OSError
- **SCons.Errors.SConsEnvironmentError**
  - Bases: Exception
- **SCons.Errors.StopError**
  - Bases: Exception
- **SCons.Errors.UserError**
  - Bases: Exception

### SCons.Errors.convert_to_BuildError(status, exc_info=None)

Convert a return code to a BuildError Exception.

The `buildError.status` we set here will normally be used as the exit status of the “scons” process.

**Parameters:**

- **status** – can either be a return code or an Exception.
- **exc_info** *(tuple, optional)* – explicit exception information.

### SCons.Executor module

Execute actions with specific lists of target and source Nodes.

**SCons.Executor.AddBatchExecutor** *(key: str, executor: Executor) → None*

**class** SCons.Executor.Batch *(targets=[], sources=[])*

- Bases: object
- Remembers exact association between targets and sources of executor.

**class** SCons.Executor.Executor *(action, env=None, overridelist=[], targets=[], sources=[], builder_kw=())
Bases: object
A class for controlling instances of executing an action.
This largely exists to hold a single association of an action, environment, list of environment override dictionaries, targets and sources for later processing as needed.

_changed_sources_list
_changed_targets_list
_do_execute
_execute_str
_get_changed_sources (*args, **kw)
_get_changed_targets (*args, **kw)
_get_changes () → None
_get_source (*args, **kw)
_get_sources (*args, **kw)
_get_target (*args, **kw)
_get_targets (*args, **kw)
_get_unchanged_sources (*args, **kw)
_get_unchanged_targets (*args, **kw)
_get_unignored_sources_key (node, ignore=())
_memo
_unchanged_sources_list
_unchanged_targets_list
action_list
add_batch (targets, sources) → None
Add pair of associated target and source to this Executor’s list. This is necessary for “batch” Builders that can be called repeatedly to build up a list of matching target and source files that will be used in order to update multiple target files at once from multiple corresponding source files, for tools like MSVC that support it.
add_post_action (action) → None
add_pre_action (action) → None
add_sources (sources) → None
Add source files to this Executor’s list. This is necessary for “multi” Builders that can be called repeatedly to build up a source file list for a given target.
batches
builder_kw
cleanup () → None
env
get_action_list ()
get_action_side_effects ()
Returns all side effects for all batches of this Executor used by the underlying Action.
get_action_targets ()
get_all_children ()
Returns all unique children (dependencies) for all batches of this Executor.
The Taskmaster can recognize when it’s already evaluated a Node, so we don’t have to make this list unique for its intended canonical use case, but we expect there to be a lot of redundancy (long lists of batched .cc files #including the same .h files over and over), so removing the duplicates once up front should save the Taskmaster a lot of work.
get_all_prerequisites ()
Returns all unique (order-only) prerequisites for all batches of this Executor.
get_all_sources ()
Returns all sources for all batches of this Executor.
get_all_targets ()
Returns all targets for all batches of this Executor.
get_build_env ()
Fetch or create the appropriate build Environment for this Executor.
get_build_scanner_path (scanner)
Fetch the scanner path for this executor’s targets and sources.
get_contents ()
Fetch the signature contents. This is the main reason this class exists, so we can compute this once and cache it regardless of how many target or source Nodes there are.

Returns bytes

generate implicit_deps()

Return the executor’s implicit dependencies, i.e. the nodes of the commands to be executed.

generate kw(kw={})
generate lvars()
generate sources()
generate_timestamp() \rightarrow int

Fetch a time stamp for this Executor. We don’t have one, of course (only files do), but this is the interface used by the timestamp module.

generate_unignored_sources(node, ignore={})

lvars

nullify() \rightarrow None

overrides

post_actions

pre_actions

prepare()

Preparatory checks for whether this Executor can go ahead and (try to) build its targets.

scan(scanner, node_list) \rightarrow None

Scan a list of this Executor’s files (targets or sources) for implicit dependencies and update all of the targets with them. This essentially short-circuits an N*M scan of the sources for each individual target, which is a hell of a lot more efficient.

scan_sources(scanner) \rightarrow None

scan_targets(scanner) \rightarrow None

set_action_list(action)

SCons.Executor.GetBatchExecutor(key: str) \rightarrow Executor

class SCons.Executor.Null(*args, **kw)

Bases: object

A null Executor, with a null build Environment, that does nothing when the rest of the methods call it.

This might be able to disappear when we refactor things to disassociate Builders from Nodes entirely, so we’re not going to worry about unit tests for this—at least for now.

_changed_sources_list

_changed_targets_list

_do_execute

_execute_str

_memo

_morph() \rightarrow None

Morph this Null executor to a real Executor object.

_changed_sources_list

_changed_targets_list

action_list

add_post_action(action) \rightarrow None

add_pre_action(action) \rightarrow None

batches

builder_kw

cleanup() \rightarrow None

eenv

generate_action_list()
generate_action_side_effects()
generate_action_targets()
generate_all_children()
generate_all_prerequisites()
generate_all_sources()
generate_all_targets()
generate_build_env()
SCons API Documentation

```python
get_build_scanner_path ()
get_contents () \rightarrow \text{str}
get_unignored_sources (*\text{args}, **\text{kw})
lvars
overrides
post_actions
pre_actions
prepare () \rightarrow \text{None}
set_action_list (\text{action}) \rightarrow \text{None}
class SCons.Executor.NullEnvironment (*\text{args}, **\text{kwargs})
    Bases: Null
    SCons = <module 'SCons' from '/Users/bdbaddog/devel/scons/git/as_scons/SCons/__init__.py'>
    _CacheDir = <SCons.CacheDir.CacheDir object>
    _CacheDir_path = None
    get_CacheDir ()
class SCons.Executor.TSList (func)
    Bases: UserList
    A class that implements $TARGETS or $SOURCES expansions by wrapping an executor Method. This class is used
    in the Executor.lvars() to delay creation of NodeList objects until they're needed.
    Note that we subclass collections.UserList purely so that the is_Sequence() function will identify an object of this
    class as a list during variable expansion. We're not really using any collections.UserList methods in practice.
    _abc_impl = <abc._abc_data object>
    append (item)
        S.append(value) \rightarrow \text{append value to the end of the sequence}
    clear () \rightarrow \text{None -- remove all items from S}
    copy ()
    count (value) \rightarrow \text{integer -- return number of occurrences of value}
    extend (other)
        S.extend(iterable) \rightarrow \text{extend sequence by appending elements from the iterable}
    index (value[, start[, stop]]) \rightarrow \text{integer -- return first index of value.}
        Raises ValueError if the value is not present.
        Supporting start and stop arguments is optional, but recommended.
    insert (i, item)
        S.insert(index, value) \rightarrow \text{insert value before index}
    pop ([, index]) \rightarrow \text{item -- remove and return item at index (default last).}
        Raise IndexError if list is empty or index is out of range.
    remove (item)
        S.remove(value) \rightarrow \text{remove first occurrence of value. Raise ValueError if the value is not present.}
    reverse ()
    S.reverse() \rightarrow \text{reverse IN PLACE}
    sort (*args, **kwargs)
class SCons.Executor.TSObject (func)
    Bases: object
    A class that implements $TARGET or $SOURCE expansions by wrapping an Executor method.
    SCons.Executor.execute_action_list (obj, target, kw)
        Actually execute the action list.
    SCons.Executor.execute_actions_str (obj)
    SCons.Executor.execute_nothing (obj, target, kw) \rightarrow \text{int}
    SCons.Executor.execute_null_str (obj) \rightarrow \text{str}
    SCons.Executor.get_NullEnvironment ()
        Use singleton pattern for Null Environments.
    SCons.Executor.rfile (node)
        A function to return the results of a Node's rfile() method, if it exists, and the Node itself otherwise (if it's a Value
        Node, e.g.).
```
SCons.Memoize module

Decorator-based memoizer to count caching stats.

A decorator-based implementation to count hits and misses of the computed values that various methods cache in memory.

Use of this modules assumes that wrapped methods be coded to cache their values in a consistent way. In particular, it requires that the class uses a dictionary named "._memo" to store the cached values.

Here is an example of wrapping a method that returns a computed value, with no input parameters:

```python
@SCons.Memoize.CountMethodCall
def foo(self):
    try:
        return self._memo['foo']
    except KeyError:
        pass
    result = self.compute_foo_value()
    self._memo['foo'] = result
    return result
```

Here is an example of wrapping a method that will return different values based on one or more input arguments:

```python
def _bar_key(self, argument):
    return argument

@SCons.Memoize.CountDictCall(_bar_key)
def bar(self, argument):
    memo_key = argument
    try:
        memo_dict = self._memo['bar']
    except KeyError:
        memo_dict = {}
    self._memo['dict'] = memo_dict
    else:
        try:
            return memo_dict[memo_key]
        except KeyError:
            pass
    result = self.compute_bar_value(argument)
    memo_dict[memo_key] = result
    return result
```

Deciding what to cache is tricky, because different configurations can have radically different performance tradeoffs, and because the tradeoffs involved are often so non-obvious. Consequently, deciding whether or not to cache a given method will likely be more of an art than a science, but should still be based on available data from this module. Here are some VERY GENERAL guidelines about deciding whether or not to cache return values from a method that’s being called a lot:
– The first question to ask is, “Can we change the calling code” so this method isn’t called so often?” Sometimes this can be done by changing the algorithm. Sometimes the caller should be memoized, not the method you’re looking at.

The memoized function should be timed with multiple configurations to make sure it doesn’t inadvertently slow down some other configuration.

– When memoizing values based on a dictionary key composed of input arguments, you don’t need to use all of the arguments if some of them don’t affect the return values.

class SCons.Memoize.CountDict(cls_name, method_name, keymaker)
Bases: Counter
A counter class for memoized values stored in a dictionary, with keys based on the method’s input arguments.
A CountDict object is instantiated in a decorator for each of the class’s methods that memoizes its return value in a dictionary, indexed by some key that can be computed from one or more of its input arguments.
count (*args, **kw) → None
Counts whether the computed key value is already present in the memoization dictionary (a hit) or not (a miss).
display () → None
key ()

SCons.Memoize.CountDictCall(keyfunc)
Decorator for counting memoizer hits/misses while accessing dictionary values with a key-generating function. Like CountMethodCall above, it wraps the given method fn and uses a CountDict object to keep track of the caching statistics. The dict-key function keyfunc has to get passed in the decorator call and gets stored in the CountDict instance. Wrapping gets enabled by calling EnableMemoization().

SCons.Memoize.CountMethodCall(fn)
Decorator for counting memoizer hits/misses while retrieving a simple value in a class method. It wraps the given method fn and uses a CountValue object to keep track of the caching statistics. Wrapping gets enabled by calling EnableMemoization().

class SCons.Memoize.CountValue(cls_name, method_name)
Bases: Counter
A counter class for simple, atomic memoized values.
A CountValue object should be instantiated in a decorator for each of the class’s methods that memoizes its return value by simply storing the return value in its _memo dictionary.
count (*args, **kw) → None
Counts whether the memoized value has already been set (a hit) or not (a miss).
display () → None
key ()

class SCons.Memoize.Counter(cls_name, method_name)
Bases: object
Base class for counting memoization hits and misses.
We expect that the initialization in a matching decorator will fill in the correct class name and method name that represents the name of the function being counted.
display () → None
key ()

SCons.Memoize.Dump(title=None) → None
Dump the hit/miss count for all the counters collected so far.
SCons.Memoize.EnableMemoization() → None

SCons.PathList module
Handle lists of directory paths.

These are the path lists that get set as CPPPATH, LIBPATH, etc.) with as much caching of data and efficiency as we can, while still keeping the evaluation delayed so that we Do the Right Thing (almost) regardless of how the variable is specified.

SCons.PathList.PathList(pathlist, split=True)
Returns the cached_pathList object for the specified pathlist, creating and caching a new object as necessary.
class SCons.PathList._PathList(pathlist, split=True)
Bases: object

An actual PathList object.

```
subst_path(env, target, source)
```

Performs construction variable substitution on a pre-digested PathList for a specific target and source.

```
SCons.PathList.node_conv(obj)
```

This is the "string conversion" routine that we have our substitutions use to return Nodes, not strings. This relies on the fact that an EntryProxy object has a get() method that returns the underlying Node that it wraps, which is a bit of architectural dependence that we might need to break or modify in the future in response to additional requirements.

SCons.SConf module

Autoconf-like configuration support.

In other words, SConf allows to run tests on the build machine to detect capabilities of system and do some things based on result: generate config files, header files for C/C++, update variables in environment.

Tests on the build system can detect if compiler sees header files, if libraries are installed, if some command line options are supported etc.

```
SCons.SConf.CheckCC (context) → bool
SCons.SConf.CheckCHeader (context, header, include_quotes: str = '')
A test for a C header file.
SCons.SConf.CheckCXX (context) → bool
SCons.SConf.CheckCXXHeader (context, header, include_quotes: str = '')
A test for a C++ header file.
```

```
class SCons.SConf.CheckContext (sconf)
Bases: object

Provides a context for configure tests. Defines how a test writes to the screen and log file.

A typical test is just a callable with an instance of CheckContext as first argument:

```
def CheckCustom(context, …):
    context.Message('Checking my weird test … ') ret = myWeirdTestFunction(…) context.Result(ret)
```

Often, myWeirdTestFunction will be one of context.TryCompile/context.TryLink/context.TryRun. The results of those are cached, for they are only rebuild, if the dependencies have changed.

```
AppendLIBS (lib_name_list, unique: bool = False)
BuildProg (text, ext) → bool
CompileProg (text, ext) → bool
CompileSharedObject (text, ext) → bool
Display (msg) → None
Log (msg) → None
Message (text) → None
    Inform about what we are doing right now, e.g. ‘Checking for SOMETHING … ‘
PrependLIBS (lib_name_list, unique: bool = False)
Result (res) → None
    Inform about the result of the test. If res is not a string, displays ‘yes’ or ‘no’ depending on whether res is evaluated as true or false. The result is only displayed when self.did_show_result is not set.
RunProg (text, ext)
SetLIBS (val)
    TryAction (*args, **kw)
    TryBuild (*args, **kw)
    TryCompile (*args, **kw)
    TryLink (*args, **kw)
    TryRun (*args, **kw)
```

```
SCons.SConf.CheckDeclaration (context, declaration, includes: str = '', language=None) → bool
SCons.SConf.CheckFunc (context, function_name, header=None, language=None, funcargs=None) → bool
SCons.SConf.CheckHeader (context, header, include_quotes: str = '<>', language=None) → bool
A test for a C or C++ header file.
SCons.SConf.CheckLib (context, library=None, symbol: str = 'main', header=None, language=None, autoadd: bool = True, append: bool = True, unique: bool = False) → bool
```
A test for a library. See also CheckLibWithHeader. Note that library may also be None to test whether the given symbol compiles without flags.

\[\text{SCons.SConf.CheckLibWithHeader (context, libs, header, language, call=None, autoadd: bool = True, append: bool = True, unique: bool = False) → bool}\]

Another (more sophisticated) test for a library. Checks, if library and header is available for language (may be ‘C’ or ‘CXX’). Call maybe be a valid expression _with_ a trailing ‘;’. As in CheckLib, we support library=None, to test if the call compiles without extra link flags.

\[\text{SCons.SConf.CheckMember (context, aggregate_member, header=None, language=None) → bool}\]

Returns the status (False : failed, True : ok).

\[\text{SCons.SConf.CheckProg (context, prog_name) → bool}\]

Simple check if a program exists in the path. Returns the path for the application, or None if not found.

\[\text{SCons.SConf.CheckSHCC (context) → bool}\]

\[\text{SCons.SConf.CheckSHCXX (context) → bool}\]

\[\text{SCons.SConf.CheckType (context, type_name, includes: str = '', language=None) → bool}\]

\[\text{SCons.SConf.CheckTypeSize (context, type_name, includes: str = '', language=None, expect=None) → bool}\]

exception \[\text{SCons.SConf.ConfigureCacheError (target)}\]

Bases: SConfError

Raised when a use explicitly requested the cache feature, but the test is run the first time.

add_note ()

Exception.add_note(note) – add a note to the exception

args

with_traceback ()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception \[\text{SCons.SConf.ConfigureDryRunError (target)}\]

Bases: SConfError

Raised when a file or directory needs to be updated during a Configure process, but the user requested a dry-run.

add_note ()

Exception.add_note(note) – add a note to the exception

args

with_traceback ()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

\[\text{SCons.SConf.CreateConfigHBuilder (env) → None}\]

Called if necessary just before the building targets phase begins.

\[\text{SCons.SConf.NeedConfigHBuilder () → bool}\]

\[\text{SCons.SConf.SConf (**args, **kw)}\]

class \[\text{SCons.SConf.SConfBase (env, custom_tests={}, conf_dir: str = '$CONFIGUREDIR', log_file: str = '$CONFIGURELOG', config_h=None, _depth: int = 0)}\]

Bases: object

This is simply a class to represent a configure context. After creating a SConf object, you can call any tests. After finished with your tests, be sure to call the Finish() method, which returns the modified environment. Some words about caching: In most cases, it is not necessary to cache Test results explicitly. Instead, we use the scons dependency checking mechanism. For example, if one wants to compile a test program (SConf.TryLink), the compiler is only called, if the program dependencies have changed. However, if the program could not be compiled in a former SConf run, we need to explicitly cache this error.

\[\text{AddTest (test_name, test_instance) → None}\]

Adds test_class to this SConf instance. It can be called with self.test_name(…)

\[\text{AddTests (tests) → None}\]

Adds all the tests given in the tests dictionary to this SConf instance

\[\text{BuildNodes (nodes)}\]

Tries to build the given nodes immediately. Returns 1 on success, 0 on error.

\[\text{Define (name, value=None, comment=None) → None}\]

Define a pre processor symbol name, with the optional given value in the current config header.

If value is None (default), then #define name is written. If value is not none, then #define name value is written.

comment is a string which will be put as a C comment in the header, to explain the meaning of the value (appropriate C comments will be added automatically).

Finish ()
Call this method after finished with your tests: `env = sconf.Finish()`

class TestWrapper (test, sconf)
    Bases: object
    A wrapper around Tests (to ensure sanity)

TryAction (action, text=None, extension: str = '')
    Tries to execute the given action with optional source file contents <text> and optional source file extension <extension>, Returns the status (0 : failed, 1 : ok) and the contents of the output file.

TryBuild (builder, text=None, extension: str = '')
    Low level TryBuild implementation. Normally you don’t need to call that - you can use TryCompile / TryLink / TryRun instead

TryCompile (text, extension)
    Compiles the program given in text to an env.Object, using extension as file extension (e.g. `.c`). Returns 1, if compilation was successful, 0 otherwise. The target is saved in self.lastTarget (for further processing).

TryLink (text, extension)
    Compiles the program given in text to an executable env.Program, using extension as file extension (e.g. `.c`). Returns 1, if compilation was successful, 0 otherwise. The target is saved in self.lastTarget (for further processing).

TryRun (text, extension)
    Compiles and runs the program given in text, using extension as file extension (e.g. `.c`). Returns (1, outputStr) on success, (0, '') otherwise. The target (a file containing the program’s stdout) is saved in self.lastTarget (for further processing).

_createDir (node)
_shutdown ()
    Private method. Reset to non-piped spawn
_startup () → None
    Private method. Set up logstream, and set the environment variables necessary for a piped build

pspawn_wrapper (sh, escape, cmd, args, env)
    Wrapper function for handling piped spawns.
    This looks to the calling interface (in Action.py) like a “normal” spawn, but associates the call with the PSPAWN variable from the construction environment and with the streams to which we want the output logged. This gets slid into the construction environment as the SPAWN variable so Action.py doesn’t have to know or care whether it’s spawning a piped command or not.

class SCons.SConf.SConfBuildInfo
    Bases: FileBuildInfo
    Special build info for targets of configure tests. Additional members are result (did the builder succeed last time?) and string, which contains messages of the original build phase.

_getstate__ ()
    Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a ‘__dict__’ slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all instances of a class.

_setstate__ (state) → None
    Restore the attributes from a pickled state.

bact
bactsig
bdepends
bdependsigs
bimplicit
bimplicit sigs
bsources
bsourcesigs
convert_from_sconsign (dir, name) → None
    Converts a newly-read FileBuildInfo object for in-SCons use
    For normal up-to-date checking, we don’t have any conversion to perform— but we’re leaving this method here to make that clear.

convert_to_sconsign () → None
    Converts this FileBuildInfo object for writing to a .sconsign file
This replaces each Node in our various dependency lists with its usual string representation: relative to the top-level SConstruct directory, or an absolute path if it’s outside.

current_version_id = 2

dependency_map

format (names: int = 0)

merge (other) → None

Merge the fields of another object into this object. Already existing information is overwritten by the other instance’s data. WARNING: If a ‘__dict__’ slot is added, it should be updated instead of replaced.

prepare_dependencies () → None

Prepares a FileBuildInfo object for explaining what changed The bsources, bdepends and bimplicit lists have all been stored on disk as paths relative to the top-level SConstruct directory. Convert the strings to actual Nodes (for use by the –debug=explain code and –implicit-cache).

result

set_build_result (result, string) → None

string

class SCons.SConf.SConfBuildTask (tm, targets, top, node)

Bases: AlwaysTask

This is almost the same as SCons.Script.BuildTask. Handles SConfErrors correctly and knows about the current cache_mode.

LOGGER = None

_abc_impl = "<abc._abc_data object>

_exception_raise ()

Raises a pending exception that was recorded while getting a Task ready for execution.

_no_exception_to_raise () → None

collect_node_states () → Tuple [bool, bool, bool]

display (message) → None

Hook to allow the calling interface to display a message.

This hook gets called as part of preparing a task for execution (that is, a Node to be built). As part of figuring out what Node should be built next, the actual target list may be altered, along with a message describing the alteration. The calling interface can subclass Task and provide a concrete implementation of this method to see those messages.

display_cached_string (bi) → None

Logs the original builder messages, given the SConfBuildInfo instance bi.

exc_clear () → None

Clears any recorded exception.

This also changes the “exception_raise” attribute to point to the appropriate do-nothing method.

exc_info ()

Returns info about a recorded exception.

exception_set (exception=None) → None

Records an exception to be raised at the appropriate time.

This also changes the “exception_raise” attribute to point to the method that will, in fact

execute ()

Called to execute the task.

This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in prepare(), executed() or failed().

executed () → None

Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s callback methods.

This may have been a do-nothing operation (to preserve build order), so we must check the node’s state before deciding whether it was “built”, in which case we call the appropriate Node method. In any event, we always call “visited()”, which will handle any post-visit actions that must take place regardless of whether or not the target was an actual built target or a source Node.

executed_with_callbacks () → None

Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s callback methods.
This may have been a do-nothing operation (to preserve build order), so we must check the node’s state before deciding whether it was “built”, in which case we call the appropriate Node method. In any event, we always call “visited()”, which will handle any post-visit actions that must take place regardless of whether or not the target was an actual built target or a source Node.

executed_without_callbacks () → None
Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s callback methods.

fail_continue () → None
Explicit continue-the-build failure.
This sets failure status on the target nodes and all of their dependent parent nodes.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

fail_stop () → None
Explicit stop-the-build failure.
This sets failure status on the target nodes and all of their dependent parent nodes.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

failed ()
Default action when a task fails: stop the build.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

get_target ()
Fetch the target being built or updated by this task.

make_ready () → None
Marks all targets in a task ready for execution if any target is not current.
This is the default behavior for building only what’s necessary.

make_ready_all () → None
Marks all targets in a task ready for execution.
This is used when the interface needs every target Node to be visited— the canonical example being the “scons -c” option.

make_ready_current () → None
Marks all targets in a task ready for execution if any target is not current.
This is the default behavior for building only what’s necessary.

needs_execute () → bool
Always returns True (indicating this Task should always be executed).
Subclasses that need this behavior (as opposed to the default of only executing Nodes that are out of date w.r.t. their dependencies) can use this as follows:

class MyTaskSubclass(SCons.Taskmaster.Task):
    needs_execute = SCons.Taskmaster.AlwaysTask.needs_execute

non_sconf_nodes = {}

postprocess () → None
Post-processes a task after it’s been executed.
This examines all the targets just built (or not, we don’t care if the build was successful, or even if there was no build because everything was up-to-date) to see if they have any waiting parent Nodes, or Nodes waiting on a common side effect, that can be put back on the candidates list.

prepare () → None
Called just before the task is executed.
This is mainly intended to give the target Nodes a chance to unlink underlying files and make all necessary directories before the Action is actually called to build the targets.

trace_message (node, description: str = 'node') → None

exception SCons.SConf.SConfError(msg)
Bases: UserError

add_note ()
Exception.add_note(note) – add a note to the exception
args
with_traceback ()
    
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception  SCons.SConf.SConfWarning

    Bases: SConsWarning

    add_note ()
        
        Exception.add_note(note) – add a note to the exception

    args

    with_traceback ()
        
        Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

SCons.SConf.SetBuildType (buildtype) \rightarrow None

SCons.SConf.SetCacheMode (mode)
    
    Set the Configure cache mode. mode must be one of “auto”, “force”, or “cache”.

SCons.SConf.SetProgressDisplay (display) \rightarrow None

    Set the progress display to use (called from SCons.Script)

class  SCons.SConfStreamer (orig)

    Bases: object

    ‘Sniffer’ for a file-like writable object. Similar to the unix tool tee.

    flush () \rightarrow None

   .getvalue ()
        
        Return everything written to orig since the Streamer was created.

    write (str) \rightarrow None

    writelines (lines) \rightarrow None

SCons.SConf._createConfigH (target, source, env) \rightarrow None

SCons.SConf._createSource (target, source, env) \rightarrow None

SCons.SConf._set_confsetup_node (node) \rightarrow None

SCons.SConf._stringConfigH (target, source, env)

SCons.SConf._stringSource (target, source, env)

SCons.SConf.createIncludesFromHeaders (headers, leaveLast, include_quotes: str = ‘”’)

SCons.SConsign module

Operations on signature database files (.sconsign).

class  SCons.SConsign.Base

    Bases: object

    This is the controlling class for the signatures for the collection of entries associated with a specific directory. The
    actual directory association will be maintained by a subclass that is specific to the underlying storage method. This
    class provides a common set of methods for fetching and storing the individual bits of information that make up
    signature entry.

    do_not_set_entry (filename, obj) \rightarrow None

    do_not_store_info (filename, node) \rightarrow None

    get_entry (filename)
        
        Fetch the specified entry attribute.

    merge () \rightarrow None

    set_entry (filename, obj) \rightarrow None

    Set the entry.

    store_info (filename, node) \rightarrow None

class  SCons.SConsign.DB (dir)

    Bases: Base

    A Base subclass that reads and writes signature information from a global .sconsign.db* file—the actual file suffix is
determined by the database module.

    do_not_set_entry (filename, obj) \rightarrow None

    do_not_store_info (filename, node) \rightarrow None

    get_entry (filename)
        
        Fetch the specified entry attribute.

    merge () \rightarrow None

    set_entry (filename, obj) \rightarrow None
Set the entry.
store_info (filename, node) → None
write (sync: int = 1) → None
class SCons.SConsign.Dir (fp=None, dir=None)
   Bases: Base
   do_not_set_entry (filename, obj) → None
do_not_store_info (filename, node) → None
get_entry (filename)
      Fetch the specified entry attribute.
      merge () → None
set_entry (filename, obj) → None
      Set the entry.
store_info (filename, node) → None
class SCons.SConsign.DirFile (dir)
   Bases: Dir
   Encapsulates reading and writing a per-directory .sconsign file.
do_not_set_entry (filename, obj) → None
do_not_store_info (filename, node) → None
get_entry (filename)
      Fetch the specified entry attribute.
      merge () → None
set_entry (filename, obj) → None
      Set the entry.
store_info (filename, node) → None
write (sync: int = 1) → None
         Write the .sconsign file to disk.
      Try to write to a temporary file first, and rename it if we succeed. If we can’t write to the temporary file, it’s probably
because the directory isn’t writable (and if so, how did we build anything in this directory, anyway?), so try to write
directly to the .sconsign file as a backup. If we can’t rename, try to copy the temporary contents back to the
.sconsign file. Either way, always try to remove the temporary file at the end.
SCons.SConsign.File (name, dbm_module=None) → None
   Arrange for all signatures to be stored in a global .sconsign.db* file.
SCons.SConsign.ForDirectory
   alias of DB
SCons.SConsign.Get_DataBase (dir)
SCons.SConsign.Reset () → None
   Reset global state. Used by unit tests that end up using SConsign multiple times to get a clean slate for each test.
class SCons.SConsign.SConsignEntry
   Bases: object
   Wrapper class for the generic entry in a .sconsign file. The Node subclass populates it with attributes as it pleases.
   XXX As coded below, we do expect a `.binfo` attribute to be added, but we’ll probably generalize this in the next
   refactorings.
binfo
class SCons.SConsign.corrupt_dblite_warning (filename) → None
SCons.SConsign.current_sconsign_filename ()
SCons.SConsign.write () → None
SCons.Subst module
SCons string substitution.
class SCons.Subst.CmdStringHolder (cmd, literal=None)
   Bases: UserString
This is a special class used to hold strings generated by `scons_subst()` and `scons_subst_list()`. It defines a special method `escape()`. When passed a function with an escape algorithm for a particular platform, it will return the contained string with the proper escape sequences inserted.

```
_abc_impl = <_abc._abc_data object>
capitalize ()
casefold ()
center (width, *args)
count (value) → integer -- return number of occurrences of value
code (encoding='utf-8', errors='strict')
endwith (suffix, start=0, end=9223372036854775807)
escape (escape_func, quote_func=<function quote_spaces>)
    Escape the string with the supplied function. The function is expected to take an arbitrary string, then return it with all special characters escaped and ready for passing to the command interpreter.
    After calling this function, the next call to str() will return the escaped string.
expandtabs (tabsize=8)
find (sub, start=0, end=9223372036854775807)
format (*args, **kwds)
format_map (mapping)
index (value[, start[, stop]]) → integer -- return first index of value.
    Raises ValueError if the value is not present.
    Supporting start and stop arguments is optional, but recommended.
is_literal () → bool
isalnum ()
isalpha ()
isascii ()
isdecimal ()
isdigit ()
isidentifier ()
islower ()
isnumeric ()
isprintable ()
ispace ()
istitle ()
isupper ()
join (seq)
ljust (width, *args)
lower ()
lstrip (chars=None)
partition (sep)
removeprefix (prefix, /)
removesuffix (suffix, /)
replace (old, new, maxsplit=-1)
find (sub, start=0, end=9223372036854775807)
index (sub, start=0, end=9223372036854775807)
rjust (width, *args)
rpartition (sep)
rsplit (sep=None, maxsplit=-1)
rstrip (chars=None)
split (sep=None, maxsplit=-1)
```
SCons API Documentation

```
splitlines (keepends=False)
startswith (prefix, start=0, end=9223372036854775807)
strip (chars=None)
swapcase ()
title ()
translate (*args)
upper ()
zfill (width)
class SCons.Subst.ListSubber (env, mode, conv, gvars)
    Bases: UserList
    A class to construct the results of a scons_subst_list() call.
    Like StringSubber, this class binds a specific construction environment, mode, target and source with two methods
    (substitute() and expand()) that handle the expansion.
    In addition, however, this class is used to track the state of the result(s) we’re gathering so we can do the appropriate
    thing whenever we have to append another word to the result—start a new line, start a new word, append to the
    current word, etc. We do this by setting the “append” attribute to the right method so that our wrapper methods only
    need ever call ListSubber.append(), and the rest of the object takes care of doing the right thing internally.
    _abc_impl = <_abc._abc_data object>
    add_new_word (x) → None
    add_to_current_word (x) → None
        Append the string x to the end of the current last word in the result. If that is not possible, then just add it as a new
        word. Make sure the entire concatenated string inherits the object attributes of x (in particular, the escape function)
        by wrapping it as CmdStringHolder.
    append (item)
        S.append(value) – append value to the end of the sequence
    clear () → None -- remove all items from S
    close_strip (x) → None
        Handle the “close strip” $) token.
    copy ()
    count (value) → integer -- return number of occurrences of value
    expand (s, lvars, within_list)
        Expand a single “token” as necessary, appending the expansion to the current result.
        This handles expanding different types of things (strings, lists, callables) appropriately. It calls the wrapper
        substitute() method to re-expand things as necessary, so that the results of expansions of side-by-side strings still
        get re-evaluated separately, not smushed together.
    expanded (s) → bool
        Determines if the string s requires further expansion.
        Due to the implementation of ListSubber expand will call itself 2 additional times for an already expanded string.
        This method is used to determine if a string is already fully expanded and if so exit the loop early to prevent these
        recursive calls.
    extend (other)
        S.extend(iterable) – extend sequence by appending elements from the iterable
    index (value[, start[, stop]]) → integer -- return first index of value.
        Raises ValueError if the value is not present.
        Supporting start and stop arguments is optional, but recommended.
    insert (i, item)
        S.insert(index, value) – insert value before index
    literal (x)
    next_line () → None
        Arrange for the next word to start a new line. This is like starting a new word, except that we have to append
        another line to the result.
    next_word () → None
        Arrange for the next word to start a new word.
    open_strip (x) → None
        Handle the “open strip” $( token.
    pop ([, index]) → item -- remove and return item at index (default last).
```
Raise IndexError if list is empty or index is out of range.

remove (item)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.
reverse ()
S.reverse() – reverse IN PLACE
sort (*args, **kwds)
substitute (args, lvars, within_list) → None
Substitute expansions in an argument or list of arguments.
This serves as a wrapper for splitting up a string into separate tokens.
this_word () → None
Arrange for the next word to append to the end of the current last word in the result.
class SCons.Subst.Literal (lstr)
    Bases: object
A wrapper for a string. If you use this object wrapped around a string, then it will be interpreted as literal. When
passed to the command interpreter, all special characters will be escaped.
    escape (escape_func)
    for_signature ()
    is_literal () → bool
class SCons.Subst.NLWrapper (list, func)
    Bases: object
A wrapper class that delays turning a list of sources or targets into a NodeList until it’s needed. The specified function
supplied when the object is initialized is responsible for turning raw nodes into proxies that implement the special
attributes like .abspath, .source, etc. This way, we avoid creating those proxies just “in case” someone is going to use
$TARGET or the like, and only go through the trouble if we really have to.
    _create_nodelist ()
    _gen_nodelist ()
    _return_nodelist ()
class SCons.Subst.NullNodeList (*args, **kwargs)
    Bases: NullSeq
    _instance
SCons.Subst.SetAllowableExceptions (*excepts) → None
class SCons.Subst.SpecialAttrWrapper (lstr, for_signature=None)
    Bases: object
This is a wrapper for what we call a ‘Node special attribute.’ This is any of the attributes of a Node that we can
reference from Environment variable substitution, such as $TARGET.abspath or $SOURCES[1].filebase. We
implement the same methods as Literal so we can handle special characters, plus a for_signature method, such that
we can return some canonical string during signature calculation to avoid unnecessary rebuilds.
    escape (escape_func)
    for_signature ()
    is_literal () → bool
class SCons.Subst.StringSubber (env, mode, conv, gvars)
    Bases: object
A class to construct the results of a scons_subst() call.
    This binds a specific construction environment, mode, target and source with two methods (substitute() and
expand()) that handle the expansion.
    expand (s, lvars)
Expand a single “token” as necessary, returning an appropriate string containing the expansion.
This handles expanding different types of things (strings, lists, callables) appropriately. It calls the wrapper
substitute() method to re-expand things as necessary, so that the results of expansions of side-by-side strings still
get re-evaluated separately, not smushed together.
    substitute (args, lvars)
Substitute expansions in an argument or list of arguments.
This serves as a wrapper for splitting up a string into separate tokens.
class SCons.Subst.Target_or_Source (nl)
    Bases: object
A class that implements $TARGET or $SOURCE expansions by in turn wrapping a NLWrapper. This class handles
the different methods used to access an individual proxy Node, calling the NLWrapper to create a proxy on demand.

class SCons.Subst.Targets_or_Sources (nl)

Bases: UserList

A class that implements $TARGETS or $SOURCES expansions by in turn wrapping a NLWrapper. This class
handles the different methods used to access the list, calling the NLWrapper to create proxies on demand.

Note that we subclass collections.UserList purely so that the is_Sequence() function will identify an object of this
class as a list during variable expansion. We’re not really using any collections.UserList methods in practice.

_abc_impl = _abc._abc_data object

append (item)

S.append(value) – append value to the end of the sequence
clear () → None -- remove all items from S
copy ()
count (value) → integer -- return number of occurrences of value
extend (other)

S.extend(iterable) – extend sequence by appending elements from the iterable
index (value[, start[, stop]]) → integer -- return first index of value.

Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.
insert (i, item)

S.insert(index, value) – insert value before index
pop ([, index]) → item -- remove and return item at index (default last).

Raise IndexError if list is empty or index is out of range.
remove (item)

S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.
reverse ()

S.reverse() – reverse IN PLACE
sort (*args, **kwds)

SCons.Subst._remove_list (list)

SCons.Subst._rm_list (list)

SCons.Subst.escape_list (mylist, escape_func)

Escape a list of arguments by running the specified escape_func on every object in the list that has an escape() method.

SCons.Subst.quote_spaces (arg)

Generic function for putting double quotes around any string that has white space in it.

SCons.Subst.raise_exception (exception, target, s)

SCons.Subst.scons_subst (strSubst, env, mode=1, target=None, source=None, gvars={}, lvars={},

cnv=None, overrides: dict | None = None)

Expand a string or list containing construction variable substitutions.

This is the work-horse function for substitutions in file names and the like. The companion scons_subst_list() function
(below) handles separating command lines into lists of arguments, so see that function if that’s what you’re looking
for.

SCons.Subst.scons_subst_list (strSubst, env, mode=1, target=None, source=None, gvars={}, lvars={},

cnv=None, overrides: dict | None = None)

Substitute construction variables in a string (or list or other object) and separate the arguments into a command list.
The companion scons_subst() function (above) handles basic substitutions within strings, so see that function
instead if that’s what you’re looking for.

SCons.Subst.scons_subst_once (strSubst, env, key)

Perform single (non-recursive) substitution of a single construction variable keyword.

This is used when setting a variable when copying or overriding values in an Environment. We want to capture
(expand) the old value before we override it, so people can do things like:

env2 = env.Clone(CCFLAGS = '$CCFLAGS -g')

We do this with some straightforward, brute-force code here...

SCons.Subst.subst_dict (target, source)

Create a dictionary for substitution of special construction variables.

This translates the following special arguments:
**SCons.Warnings module**

The SCons Warnings framework.

Enables issuing warnings in situations where it is useful to alert the user of a condition that does not warrant raising an exception that could terminate the program.

A new warning class should inherit (perhaps indirectly) from one of two base classes: SConsWarning or WarningOnByDefault, which are the same except warnings derived from the latter will start out in an enabled state. Enabled warnings cause a message to be printed when called, disabled warnings are silent.

There is also a hierarchy for indicating deprecations and future changes: for these, derive from DeprecatedWarning, MandatoryDeprecatedWarning, FutureDeprecatedWarning or FutureReservedVariableWarning.

Whether or not to display warnings, beyond those that are on by default, is controlled through the command line (`--warn`) or through `SetOption('warn')`. The names used there use a different naming style than the warning class names. `process_warn_strings()` converts the names before enabling/disabling.

The behavior of issuing only a message (for “enabled” warnings) can be toggled to raising an exception instead by calling the `warningAsException()` function.

For new/removed warnings, the manpage needs to be kept in sync. Any warning class defined here is accepted, but we don’t want to make people have to dig around to find the names. Warnings do not have to be defined in this file, though it is preferred: those defined elsewhere cannot use the enable/disable functionality unless they monkeypatch the warning into this module’s namespace.

You issue a warning, either in SCons code or in a build project’s SConscripts, by calling the `warn()` function defined in this module. Raising directly with an instance of a warning class bypasses the framework and it will behave like an ordinary exception.

```python
exception SCons.Warnings.CacheCleanupErrorWarning
    Bases: SConsWarning
    Problems removing retrieved target prior to rebuilding.
exception SCons.Warnings.CacheVersionWarning
    Bases: WarningOnByDefault
    The derived-file cache directory has an out of date config.
exception SCons.Warnings.CacheWriteErrorWarning
    Bases: SConsWarning
    Problems writing a derived file to the cache.
exception SCons.Warnings.CorruptSConsignWarning
    Bases: WarningOnByDefault
    Problems decoding the contents of the sconsign database.
exception SCons.Warnings.DependencyWarning
    Bases: SConsWarning
    A scanner identified a dependency but did not add it.
exception SCons.Warnings.DeprecatedDebugOptionsWarning
    Bases: MandatoryDeprecatedWarning
    Option-arguments to --debug that are deprecated.
exception SCons.Warnings.DeprecatedOptionsWarning
    Bases: MandatoryDeprecatedWarning
    Options that are deprecated.
exception SCons.Warnings.DeprecatedWarning
    Bases: SConsWarning
    Base class for deprecated features, will be removed in future.
exception SCons.Warnings.DevelopmentVersionWarning
```

---

**target** - the target (object or array of objects),
used to generate the TARGET and TARGETS construction variables

**source** - the source (object or array of objects),
used to generate the SOURCES and SOURCE construction variables
SCons API Documentation

- Bases: WarningOnByDefault
  Use of a deprecated feature.

```
exception SCons.Warnings.DuplicateEnvironmentWarning
    Bases: WarningOnByDefault
    A target appears in more than one consenv with identical actions.
    A duplicate target with different rules cannot be built; with the same rule it can, but this could indicate a problem in the build configuration.
```

```
exception SCons.Warnings.FortranCxxMixWarning
    Bases: LinkWarning
    Fortran and C++ objects appear together in a link line.
    Some compilers support this, others do not.
```

```
exception SCons.Warnings.FutureDeprecatedWarning
    Bases: SConsWarning
    Base class for features that will become deprecated in a future release.
```

```
exception SCons.Warnings.FutureReservedVariableWarning
    Bases: WarningOnByDefault
    Setting a variable marked to become reserved in a future release.
```

```
exception SCons.Warnings.LinkWarning
    Bases: WarningOnByDefault
    Base class for linker warnings.
```

```
exception SCons.Warnings.MandatoryDeprecatedWarning
    Bases: DeprecatedWarning
    Base class for deprecated features where warning cannot be disabled.
```

```
exception SCons.Warnings.MisleadingKeywordsWarning
    Bases: WarningOnByDefault
    Use of possibly misspelled kwargs in Builder calls.
```

```
exception SCons.Warnings.MissingSConscriptWarning
    Bases: WarningOnByDefault
    The script specified in an SConscript() call was not found.
    TODO: this is now an error, so no need for a warning. Left in for a while in case anyone is using, remove eventually.
    Manpage entry removed in 4.6.0.
```

```
exception SCons.Warnings.NoObjectCountWarning
    Bases: WarningOnByDefault
    Object counting (debug mode) could not be enabled.
```

```
exception SCons.Warnings.NoParallelSupportWarning
    Bases: WarningOnByDefault
    Fell back to single-threaded build, as no thread support found.
```

```
exception SCons.Warnings.PythonVersionWarning
    Bases: DeprecatedWarning
    SCons was run with a deprecated Python version.
```

```
exception SCons.Warnings.ReservedVariableWarning
    Bases: WarningOnByDefault
    Attempt to set reserved construction variable names.
```

```
exception SCons.Warnings.SConsWarning
    Bases: UserError
    Base class for all SCons warnings.
```

```
SCons.Warnings.SConsWarningOnByDefault
    alias of WarningOnByDefault
```

```
exception SCons.Warnings.StackSizeWarning
    Bases: WarningOnByDefault
    Requested thread stack size could not be set.
```

```
exception SCons.Warnings.TargetNotBuiltWarning
    Bases: SConsWarning
    A target build indicated success but the file is not found.
```

```
exception SCons.Warnings.ToolQtDeprecatedWarning
    Bases: DeprecatedWarning
```

182
Exception SCons.Warnings.VisualCMissingWarning
   Bases: WarningOnByDefault
   Requested MSVC version not found and policy is to not fail.
Exception SCons.Warnings.VisualStudioMissingWarning
   Bases: SConsWarning
Exception SCons.Warnings.VisualVersionMismatch
   Bases: WarningOnByDefault
   MSVC_VERSION and MSVS_VERSION do not match.
   Note MSVS_VERSION is deprecated, use MSVC_VERSION.
Exception SCons.Warnings.WarningOnByDefault
   Bases: SConsWarning
   Base class for SCons warnings that are enabled by default.
SCons.Warnings.enableWarningClass (clazz) → None
   Enables all warnings of type clazz or derived from clazz.
SCons.Warnings.process_warn_strings (arguments: Sequence[str]) → None
   Process requests to enable/disable warnings.
   The requests come from the option-argument string passed to the --warn command line option or as the value
   passed to the SetOption function with a first argument of warn;
   The arguments are expected to be as documented in the SCons manual page for the --warn option, in the style
   some-type, which is converted here to a camel-case name like SomeTypeWarning, to try to match the warning
   classes defined here, which are then passed to enableWarningClass() or suppressWarningClass().
   For example, a string "deprecated" enables the DeprecatedWarning class, while a string "no-dependency" disables
   the DependencyWarning class.
   As a special case, the string "all" disables all warnings and a the string "no-all" disables all warnings.
SCons.Warnings.suppressWarningClass (clazz) → None
   Suppresses all warnings of type clazz or derived from clazz.
SCons.Warnings.warn (clazz, *args) → None
   Issue a warning, accounting for SCons rules.
   Check if warnings for this class are enabled. If warnings are treated as exceptions, raise exception. Use the global
   warning emitter _warningOut, which allows selecting different ways of presenting a traceback (see Script/Main.py).
SCons.Warnings.warningAsException (flag: bool = True) → bool
   Sets global _warningAsExeption flag.
   If true, any enabled warning will cause an exception to be raised.
   Returns: The previous value.

SCons.cpp module
SCons C Pre-Processor module
SCons.cpp.CPP_to_Python (s)→ None
   Converts a C pre-processor expression into an equivalent Python expression that can be evaluated.
SCons.cpp.CPP_to_Python_Ops_Sub (m)→ None
SCons.cpp.Cleanup_CPP_Expressions (ts)→ None
Class SCons.cpp.DumbPreProcessor (*args, **kw)
   Bases: PreProcessor
   A preprocessor that ignores all #if/#elif/#else/#endif directives and just reports back all of the #include files (like the
classic SCons scanner did).
   This is functionally equivalent to using a regular expression to find all of the #include lines, only slower. It exists
mainly as an example of how the main PreProcessor class can be sub-classed to tailor its behavior.
   __call__ (file)
   Pre-processes a file.
   This is the main public entry point.
   __do_if_else_condition (condition) → None
   Common logic for evaluating the conditions on #if, #ifdef and #ifndef lines.
   __match_tuples (tuples)


_SCons API Documentation_

```python
_parse_tuples(contents)
_process_tuples(tuples, file=None)
all_include (t) → None
do_define (t) → None
    Default handling of a #define line.
do_elif (t) → None
    Default handling of a #elif line.
do_else (t) → None
    Default handling of a #else line.
do_endif (t) → None
    Default handling of a #endif line.
do_if (t) → None
    Default handling of a #if line.
do ifdef (t) → None
    Default handling of a #ifdef line.
do ifndef (t) → None
    Default handling of a #ifndef line.
do_import (t) → None
    Default handling of a #import line.
do_include (t) → None
    Default handling of a #include line.
do_include_next (t) → None
    Default handling of a #include line.
do_nothing (t) → None
    Null method for when we explicitly want the action for a specific preprocessor directive to do nothing.
do_undef (t) → None
    Default handling of a #undef line.
eval_expression (t)
    Evaluates a C preprocessor expression.
    This is done by converting it to a Python equivalent and eval()ing it in the C preprocessor namespace we use to
    track #define values.
finalize_result (fname)
find_include_file (t)
    Finds the #include file for a given preprocessor tuple.
initialize_result (fname) → None
process_contents (contents)
    Pre-processes a file contents.
    Is used by tests
process_file (file)
    Pre-processes a file.
    This is the main internal entry point.
read_file (file) → str
resolve_include (t)
    Resolve a tuple-ized #include line.
    This handles recursive expansion of values without "" or <> surrounding the name until an initial " or < is found, to
    handle #include FILE where FILE is a #define somewhere else.
restore () → None
    Pops the previous dispatch table off the stack and makes it the current one.
save () → None
    Pushes the current dispatch table on the stack and re-initializes the current dispatch table to the default.
scons_current_file (t) → None
start_handling_includes (t=None) → None
    Causes the PreProcessor object to start processing #import, #include and #include_next lines.
    This method will be called when a #if, #ifdef, #ifndef or #elif evaluates True, or when we reach the #else in a #if,
    #ifdef, #ifndef or #elif block where a condition already evaluated False.
stop_handling_includes (t=None) → None
```
Causes the PreProcessor object to stop processing #import, #include and #include_next lines. This method will be called when a #if, #ifdef, #ifndef or #elif evaluates False, or when we reach the #else in a #if, #ifdef, #ifndef or #elif block where a condition already evaluated True.

tupleize (contents)
Turns the contents of a file into a list of easily-processed tuples describing the CPP lines in the file. The first element of each tuple is the line’s preprocessor directive (#if, #include, #define, etc., minus the initial ‘#’). The remaining elements are specific to the type of directive, as pulled apart by the regular expression.

class SCons.cpp.FunctionEvaluator (name, args, expansion)
Bases: object
Handles delayed evaluation of a #define function call.
__call__ (*values)
Evaluates the expansion of a #define macro function called with the specified values.

class SCons.cpp.PreProcessor (current=‘.’, cpppath=(), dict={}, all: int = 0, depth=-1)
Bases: object
The main workhorse class for handling C pre-processing.
__call__ (file)
Pre-processes a file. This is the main public entry point.
_do_if_else_condition (condition) → None
Common logic for evaluating the conditions on #if, #ifdef and #ifndef lines.
_match_tuples (tuples)
_parse_tuples (contents)
_process_tuples (tuples, file=None)
_all_include (t) → None
do_define (t) → None
Default handling of a #define line.
do_elif (t) → None
Default handling of a #elif line.
do_else (t) → None
Default handling of a #else line.
do endif (t) → None
Default handling of a #endif line.
do_if (t) → None
Default handling of a #if line.
do_ifdef (t) → None
Default handling of a #ifdef line.
do ifndef (t) → None
Default handling of a #ifndef line.
do import (t) → None
Default handling of a #import line.
do include (t) → None
Default handling of a #include line.
do include_next (t) → None
Default handling of a #include line.
do nothing (t) → None
Null method for when we explicitly want the action for a specific preprocessor directive to do nothing.
do undef (t) → None
Default handling of a #undef line.
eval_expression (t)
Evaluates a C preprocessor expression. This is done by converting it to a Python equivalent and eval()ing it in the C preprocessor namespace we use to track #define values.
finalize_result (fname)
find_include_file (t)
Finds the #include file for a given preprocessor tuple.
initialize_result (fname) → None
process_contents (contents)
Pre-processes a file contents.
Is used by tests
process_file (file)
Pre-processes a file.
This is the main internal entry point.
read_file (file) → str
resolve_include (t)
Resolve a tuple-ized #include line.
This handles recursive expansion of values without "" or <> surrounding the name until an initial "" or < is found, to
handle #include FILE where FILE is a #define somewhere else.
restore () → None
Pops the previous dispatch table off the stack and makes it the current one.
save () → None
Pushes the current dispatch table on the stack and re-initializes the current dispatch table to the default.
start_handling_includes (t=None) → None
Causes the PreProcessor object to start processing #import, #include and #include_next lines.
This method will be called when a #if, ifndef, ifndef or #elif evaluates True, or when we reach the #else in a #if,
ifndef, ifndef or #elif block where a condition already evaluated False.
stop_handling_includes (t=None) → None
Causes the PreProcessor object to stop processing #import, #include and #include_next lines.
This method will be called when a #if, ifndef, ifndef or #elif evaluates False, or when we reach the #else in a #if,
ifndef, ifndef or #elif block where a condition already evaluated True.
tupleize (contents)
Turns the contents of a file into a list of easily-processed tuples describing the CPP lines in the file.
The first element of each tuple is the line’s preprocessor directive (#if, #include, #define, etc., minus the initial ‘#’).
The remaining elements are specific to the type of directive, as pulled apart by the regular expression.

SCons.dblite module
dblite.py module contributed by Ralf W. Grosse-Kunstleve. Extended for Unicode by Steven Knight.
This is a very simple-minded “database” used for saved signature information, with an interface modeled on the Python
$dbm$ database interface module.
class SCons.dblite._Dblite (file_base_name, flag='r', mode=438)
Bases: object
Lightweight signature database class.
Behaves like a dict when in memory, loads from a pickled disk file on open and writes back out to it on close.
Open the database file using a path derived from file_base_name. The optional flag argument can be:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'r'</td>
<td>Open existing database for reading only (default)</td>
</tr>
<tr>
<td>'w'</td>
<td>Open existing database for reading and writing</td>
</tr>
<tr>
<td>'c'</td>
<td>Open database for reading and writing, creating it if it doesn’t exist</td>
</tr>
<tr>
<td>'n'</td>
<td>Always create a new, empty database, open for reading and writing</td>
</tr>
</tbody>
</table>

The optional mode argument is the POSIX mode of the file, used only when the database has to be created. It
defaults to octal 0o666.
_check_writable ()
static _open (file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)
Open file and return a stream. Raise OSError upon failure.
file is either a text or byte string giving the name (and the path if the file isn’t in the current working directory) of the
file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when
the returned I/O object is closed, unless closefd is set to False.)
mode is an optional string that specifies the mode in which the file is opened. It defaults to ‘r’ which means open for reading in text mode. Other common values are ‘w’ for writing (truncating the file if it already exists), ‘x’ for creating and writing to a new file, and ‘a’ for appending (which on some Unix systems, means that all writes append to the end of the file regardless of the current seek position). In text mode, if encoding is not specified the encoding used is platform dependent: locale.getencoding() is called to get the current locale encoding. (For reading and writing raw bytes use binary mode and leave encoding unspecified.) The available modes are:

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘r’</td>
<td>open for reading (default)</td>
</tr>
<tr>
<td>‘w’</td>
<td>open for writing, truncating the file first</td>
</tr>
<tr>
<td>‘x’</td>
<td>create a new file and open it for writing</td>
</tr>
<tr>
<td>‘a’</td>
<td>open for writing, appending to the end of the file if it exists</td>
</tr>
<tr>
<td>‘b’</td>
<td>binary mode</td>
</tr>
<tr>
<td>‘t’</td>
<td>text mode (default)</td>
</tr>
<tr>
<td>‘+’</td>
<td>open a disk file for updating (reading and writing)</td>
</tr>
</tbody>
</table>

The default mode is ‘rt’ (open for reading text). For binary random access, the mode ‘w+b’ opens and truncates the file to 0 bytes, while ‘r+b’ opens the file without truncation. The ‘x’ mode implies ‘w’ and raises an `FileExistsError` if the file already exists.

Python distinguishes between files opened in binary and text modes, even when the underlying operating system doesn’t. Files opened in binary mode (appending ‘b’ to the mode argument) return contents as bytes objects without any decoding. In text mode (the default, or when ‘t’ is appended to the mode argument), the contents of the file are returned as strings, the bytes having been first decoded using a platform-dependent encoding or using the specified encoding if given.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size of a fixed-size chunk buffer. When no buffering argument is given, the default buffering policy works as follows:

- Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device’s “block size” and falling back on `io.DEFAULT_BUFFER_SIZE`. On many systems, the buffer will typically be 4096 or 8192 bytes long.
- ”Interactive” text files (files for which isatty() returns True) use line buffering. Other text files use the policy described above for binary files.

encoding is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent, but any encoding supported by Python can be passed. See the codecs module for the list of supported encodings.

eol is an optional string that specifies how encoding errors are to be handled—this argument should not be used in binary mode. Pass ‘strict’ to raise a `ValueError` exception if there is an encoding error (the default of None has the same effect), or pass ‘ignore’ to ignore errors. (Note that ignoring encoding errors can lead to data loss.) See the documentation for codecs.register or run ‘help(codecs.Codec)’ for a list of the permitted encoding error strings.

eol controls how universal newlines works (it only applies to text mode). It can be None, ‘’, ‘n’, ‘y’, and ‘m’. It works as follows:

- On input, if newline is None, universal newlines mode is enabled. Lines in the input can end in ‘n’, ‘r’, or ‘rn’, and these are translated into ‘n’ before being returned to the caller. If it is ‘’, universal newline mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- On output, if newline is None, any ‘n’ characters written are translated to the system default line separator, os.linesep. If newline is ‘ ‘ or ‘n’, no translation takes place. If newline is any of the other legal values, any ‘n’ characters written are translated to the given string.

If closedfd is False, the underlying file descriptor will be kept open when the file is closed. This does not work when a file name is given and must be True in that case.
A custom opener can be used by passing a callable as `opener`. The underlying file descriptor for the file object is then obtained by calling `opener` with `(file, flags)`. `opener` must return an open file descriptor (passing `os.open` as `opener` results in functionality similar to passing `None`).

`open()` returns a file object whose type depends on the mode, and through which the standard file operations such as reading and writing are performed. When `open()` is used to open a file in a text mode (`'w', 'r', 'wt', 'rt', etc.`), it returns a `TextIOWrapper`. When used to open a file in a binary mode, the returned class varies: in read binary mode, it returns a `BufferedReader`; in write binary and append binary modes, it returns a `BufferedWriter`, and in read/write mode, it returns a `BufferedRandom`.

It is also possible to use a string or `bytearray` as a file for both reading and writing. For strings, `StringIO` can be used like a file opened in a text mode, and for bytes `BytesIO` can be used like a file opened in a binary mode.

```python
static _os_chmod (path, mode, *, dir_fd=None, follow_symlinks=True)
Change the access permissions of a file.

path
Path to be modified. May always be specified as a str, bytes, or a path-like object. On some platforms, path may also be specified as an open file descriptor. If this functionality is unavailable, using it raises an exception.

mode
Operating-system mode bitfield.

dir_fd
If not None, it should be a file descriptor open to a directory, and path should be relative; path will then be relative to that directory.

follow_symlinks
If False, and the last element of the path is a symbolic link, chmod will modify the symbolic link itself instead of the file the link points to.

It is an error to use `dir_fd` or `follow_symlinks` when specifying `path` as an open file descriptor.

dir_fd and follow_symlinks may not be implemented on your platform.
If they are unavailable, using them will raise a `NotImplementedError`.

```python
static _os_chown (path, uid, gid, *, dir_fd=None, follow_symlinks=True)
Change the owner and group id of path to the numeric uid and gid.

path
Path to be examined; can be string, bytes, a path-like object, or open-file-descriptor int.

dir_fd
If not None, it should be a file descriptor open to a directory, and path should be relative; path will then be relative to that directory.

follow_symlinks
If False, and the last element of the path is a symbolic link, chown will examine the symbolic link itself instead of the file the link points to.

path may always be specified as a string. On some platforms, path may also be specified as an open file descriptor.

If this functionality is unavailable, using it raises an exception.

If `dir_fd` is not None, it should be a file descriptor open to a directory, and path should be relative; path will then be relative to that directory.

If `follow_symlinks` is False, and the last element of the path is a symbolic link, chown will modify the symbolic link itself instead of the file the link points to.

It is an error to use `dir_fd` or `follow_symlinks` when specifying `path` as an open file descriptor.

dir_fd and follow_symlinks may not be implemented on your platform.
If they are unavailable, using them will raise a `NotImplementedError`.

```python
static _os_replace (src, dst, *, src_dir_fd=None, dst_dir_fd=None)
```

Rename a file or directory, overwriting the destination.

**If either `src_dir_fd` or `dst_dir_fd` is not None, it should be a file**

descriptor open to a directory, and the respective path string (src or dst) should be relative; the path will then be relative to that directory.

**`src_dir_fd` and `dst_dir_fd`, may not be implemented on your platform.**

If they are unavailable, using them will raise a `NotImplementedError`.

```python
static _pickle_dump (obj, file, protocol=None, *, fix_imports=True, buffer_callback=None)
```

Write a pickled representation of `obj` to the open file object `file`.

This is equivalent to `Pickler(file, protocol).dump(obj)`, but may be more efficient.

The optional `protocol` argument tells the pickler to use the given protocol; supported protocols are 0, 1, 2, 3, 4 and 5. The default protocol is 4. It was introduced in Python 3.4, and is incompatible with previous versions.

Specifying a negative protocol version selects the highest protocol version supported. The higher the protocol used, the more recent the version of Python needed to read the pickle produced.

The `file` argument must have a `write()` method that accepts a single bytes argument. It can thus be a file object opened for binary writing, an `io.BytesIO` instance, or any other custom object that meets this interface.

If `fix_imports` is True and protocol is less than 3, pickle will try to map the new Python 3 names to the old module names used in Python 2, so that the pickle data stream is readable with Python 2.

If `buffer_callback` is None (the default), buffer views are serialized into `file` as part of the pickle stream. It is an error if `buffer_callback` is not None and `protocol` is None or smaller than 5.

```
_pickled_protocol = 4
```

```python
static _shutil_copyfile (src, dst, *, follow_symlinks=True)
```

Copy data from `src` to `dst` in the most efficient way possible.

If `follow_symlinks` is not set and `src` is a symbolic link, a new symlink will be created instead of copying the file it points to.

```python
static _time_time ()
```

Return the current time in seconds since the Epoch. Fractions of a second may be present if the system clock provides them.

```
close () → None
items ()
keys ()
```

```python
opener (path, flags)
```

Database open helper when creation may be needed.

The high-level Python open() function cannot specify a file mode for creation. Using this as the opener with the saved mode lets us do that.

```
sync () → None
```

Flush the database to disk.

This routine must succeed, since the in-memory and on-disk copies are out of sync as soon as we do anything that changes the in-memory version. Thus, to be cautious, flush to a temporary file and then move it over with some error handling.

```
values ()
```

SCons.dblite._exercise ()

SCons.dblite.open (file, flag='r', mode: int = 438)

SCons.exitfuncs module

Register functions which are executed when SCons exits for any reason.

```
SCons.exitfuncs._run_exitfuncs () → None
```

run any registered exit functions

`_exithandlers` is traversed in reverse order so functions are executed last in, first out.

```
SCons.exitfuncs.register (func, *targs, **kargs) → None
```

register a function to be executed upon normal program termination
SCons.compat package

func - function to be called at exit
targs - optional arguments to pass to func
kargs - optional keyword arguments to pass to func

SCons.compat package

Module contents

SCons compatibility package for old Python versions

This subpackage holds modules that provide backwards-compatible implementations of various things from newer Python versions that we cannot count on because SCons still supported older Pythons.

Other code will not generally reference things in this package through the SCons.compat namespace. The modules included here add things to the builtins namespace or the global module list so that the rest of our code can use the objects and names imported here regardless of Python version. As a result, if this module is used, it should violate the normal convention for imports (standard library imports first, then program-specific imports, each ordered alphabetically) and needs to be listed first.

The rest of the things here will be in individual compatibility modules that are either: 1) suitably modified copies of the future modules that we want to use; or 2) backwards compatible re-implementations of the specific portions of a future module's API that we want to use.

GENERAL WARNINGS: Implementations of functions in the SCons.compat modules are NOT guaranteed to be fully compliant with these functions in later versions of Python. We are only concerned with adding functionality that we actually use in SCons, so be wary if you lift this code for other uses. (That said, making these more nearly the same as later, official versions is still a desirable goal, we just don't need to be obsessive about it.)

We name the compatibility modules with an initial '_scons_' (for example, _scons_subprocess.py is our compatibility module for subprocess) so that we can still try to import the real module name and fall back to our compatibility module if we get an ImportError. The import_as() function defined below loads the module as the "real" name (without the '_scons'), after which all of the "import {module}" statements in the rest of our code will find our pre-loaded compatibility module.

class SCons.compat.NoSlotsPyPy (name, bases, dct)
   Bases: type
   Metaclass for PyPy compatibility.
   PyPy does not work well with __slots__ and __class__ assignment.
   mro ()
      Return a type's method resolution order.
   SCons.compat.rename_module (new, old) → bool
      Attempt to import the old module and load it under the new name. Used for purely cosmetic name changes in Python 3.x.

SCons.Node package

Submodules

SCons.Node.Alias module

Alias nodes.

This creates a hash of global Aliases (dummy targets).

class SCons.Node.Alias.Alias (name)
   Bases: Node
class Attrs
      Bases: object
      shared
      BuildInfo
      alias of AliasBuildInfo
Decider (function) → None
GetTag (key)
    Return a user-defined tag.
NodeInfo
    alias of AliasNodeInfo
Tag (key, value) → None
    Add a user-defined tag.
__add_child (collection, set, child) → None
    Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.
__children_get ()
__children_reset () → None
__func_exists
__func_get_contents
__func_is_derived
__func_rexists
__func_target_from_source
__get_scanner (env, initial_scanner, root_node_scanner, kw)
__memo
__specific_sources
__tags
add_dependency (depend)
    Adds dependencies.
add_ignore (depend)
    Adds dependencies to ignore.
add_prerequisite (prerequisite) → None
    Adds prerequisites
add_source (source)
    Adds sources.
add_to_implicit (deps) → None
add_to_waiting_parents (node) → int
    Returns the number of nodes added to our waiting parents list: 1 if we add a unique waiting parent, 0 if not. (Note that the returned values are intended to be used to increment a reference count, so don’t think you can “clean up” this function by using True and False instead…)
add_to_waiting_s_e (node) → None
add_wkid (wkid) → None
    Add a node to the list of kids waiting to be evaluated
all_children (scan: int = 1)
    Return a list of all the node’s direct children.
alter_targets ()
    Return a list of alternate targets for this Node.
always_build
attributes
binfo
build () → None
    A “builder” for aliases.
binfo
builder
builder_set (builder) → None
built () → None
    Called just after this node is successfully built.
cached
changed (node=None, allowcache: bool = False)
    Returns if the node is up-to-date with respect to the BuildInfo stored last time it was built. The default behavior is to compare it against our own previously stored BuildInfo, but the stored BuildInfo from another Node (typically one in a Repository) can be used instead.
Note that we now always check every dependency. We used to short-circuit the check by returning as soon as we detected any difference, but we now rely on checking every dependency to make sure that any necessary Node information (for example, the content signature of an #included .h file) is updated.

The allowcache option was added for supporting the early release of the executor/builder structures, right after a File target was built. When set to true, the return value of this changed method gets cached for File nodes. Like this, the executor isn’t needed any longer for subsequent calls to changed().

@see: FS.File.changed(), FS.File.release_target_info()

changed_since_last_build
check_attributes (name)
  Simple API to check if the node.attributes for name has been set
children (scan: int = 1)
  Return a list of the node’s direct children, minus those that are ignored by this node.
children_are_up_to_date () → bool
  Alternate check for whether the Node is current: If all of our children were up-to-date, then this Node was up-to-date, too.
clear () → None
  Completely clear a Node of all its cached state (so that it can be re-evaluated by interfaces that do continuous integration builds).
clear_memoized_values () → None
clear_memoized_values () → None
convert () → None
del_binfo () → None
  Delete the build info from this node.
depends
depends_set
disambiguate (must_exist=None)
  env
  env_set (env, safe: bool = False) → None
  executor
  executor_cleanup () → None
  Let the executor clean up any cached information.
exists () → bool
  Reports whether node exists.
explain ()
for_signature ()
  Return a string representation of the Node that will always be the same for this particular Node, no matter what.
  This is by contrast to the __str__() method, which might, for instance, return a relative path for a file Node. The purpose of this method is to generate a value to be used in signature calculation for the command line used to build a target, and we use this method instead of str() to avoid unnecessary rebuilds. This method does not need to return something that would actually work in a command line; it can return any kind of nonsense, so long as it does not change.
get_abspath ()
  Return an absolute path to the Node. This will return simply str(Node) by default, but for Node types that have a concept of relative path, this might return something different.
get_binfo ()
  Fetch a node’s build information.
  node - the node whose sources will be collected cache - alternate node to use for the signature cache returns - the build signature
  This no longer handles the recursive descent of the node’s children’s signatures. We expect that they’re already built and updated by someone else, if that’s what’s wanted.
get_build_env ()
  Fetch the appropriate Environment to build this node.
get_build_scanner_path (scanner)
  Fetch the appropriate scanner path for this node.
get_builder (default_builder=None)
  Return the set builder, or a specified default value
get_cachedir_csig()
get_contents()
The contents of an alias is the concatenation of the content signatures of all its sources.
get_csig()
Generate a node's content signature, the digested signature of its content.
node - the node cache - alternate node to use for the signature cache returns - the content signature
get_env()
get_env_scanner (env, kw={})
get_executor (create: int = 1) → Executor
Fetch the action executor for this node. Create one if there isn’t already one, and requested to do so.
get_found_includes (env, scanner, path)
Return the scanned include lines (implicit dependencies) found in this node.
The default is no implicit dependencies. We expect this method to be overridden by any subclass that can be scanned for implicit dependencies.
get_implicit_deps (env, initial_scanner, path_func, kw={})
Return a list of implicit dependencies for this node.
This method exists to handle recursive invocation of the scanner on the implicit dependencies returned by the scanner, if the scanner's recursive flag says that we should.
get_ninfo()
get_source_scanner (node)
Fetch the source scanner for the specified node
NOTE: “self” is the target being built, “node” is the source file for which we want to fetch the scanner.
Implies self.has_builder() is true; again, expect to only be called from locations where this is already verified.
This function may be called very often; it attempts to cache the scanner found to improve performance.
get_state()
get_stored_implicit ()
Fetch the stored implicit dependencies
get_stored_info ()
get_string (for_signature)
This is a convenience function designed primarily to be used in command generators (i.e., CommandGeneratorActions or Environment variables that are callable), which are called with a for_signature argument that is nonzero if the command generator is being called to generate a signature for the command line, which determines if we should rebuild or not.
Such command generators should use this method in preference to str(Node) when converting a Node to a string, passing in the for_signature parameter, such that we will call Node.for_signature() or str(Node) properly, depending on whether we are calculating a signature or actually constructing a command line.
get_subst_proxy ()
This method is expected to return an object that will function exactly like this Node, except that it implements any additional special features that we would like to be in effect for Environment variable substitution. The principle use is that some Nodes would like to implement a __getattr__() method, but putting that in the Node type itself has a tendency to kill performance. We instead put it in a proxy and return it from this method. It is legal for this method to return self if no new functionality is needed for Environment substitution.
get_suffix () → str
get_target_scanner ()
has_builder () → bool
Return whether this Node has a builder or not.
In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly (“if node.builder: …”). When the builder attribute is examined directly, it ends up calling __getattr__ for both the __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.
has_explicit_builder () → bool
Return whether this Node has an explicit builder.
This allows an internal Builder created by SCons to be marked non-explicit, so that it can be overridden by an explicit builder that the user supplies (the canonical example being directories).
ignore
ignore_set
implicit
implicit_set
includes
is_conftest () \rightarrow bool
    Returns true if this node is an conftest node
is-derived () \rightarrow bool
    Returns true if this node is derived (i.e. built).
    This should return true only for nodes whose path should be in the variant directory when duplicate=0 and should
    contribute their build signatures when they are used as source files to other derived files. For example: source with
    source builders are not derived in this sense, and hence should not return true.

is_explicit
is_literal () \rightarrow bool
    Always pass the string representation of a Node to the command interpreter literally.

is_sconscript () \rightarrow bool
    Returns true if this node is an sconscript

is_under (dir) \rightarrow bool

is_up_to_date () \rightarrow bool
    Alternate check for whether the Node is current: If all of our children were up-to-date, then this Node was
    up-to-date, too.

linked
make_ready () \rightarrow None
    Get a Node ready for evaluation.
    This is called before the Taskmaster decides if the Node is up-to-date or not. Overriding this method allows for a
    Node subclass to be disambiguated if necessary, or for an implicit source builder to be attached.

missing () \rightarrow bool
multiple_side_effect_has_builder () \rightarrow bool
    Return whether this Node has a builder or not.
    In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly ("if
    node.builder: …"). When the builder attribute is examined directly, it ends up calling __getattr__ for both the
    __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and
    slowing things down immensely.

new_binfo ()
nnew_ninfo ()
ninfo
nocache
noclean
postprocess () \rightarrow None
    Clean up anything we don’t need to hang onto after we’ve been built.

precious
prepare ()
    Prepare for this Node to be built.
    This is called after the Taskmaster has decided that the Node is out-of-date and must be rebuilt, but before actually
    calling the method to build the Node.
    This default implementation checks that explicit or implicit dependencies either exist or are derived, and initializes
    the BuildInfo structure that will hold the information about how this node is, uh, built.
    (The existence of source files is checked separately by the Executor, which aggregates checks for all of the targets
    built by a specific action.)
    Overriding this method allows for for a Node subclass to remove the underlying file from the file system. Note that
    subclass methods should call this base class method to get the child check and the BuildInfo structure.

prerequisites
pseudo
push_to_cache () \rightarrow None
    Try to push a node into a cache
really_build (**kw)
    Actually build the node.
This is called by the Taskmaster after it's decided that the Node is out-of-date and must be rebuilt, and after the prepare() method has gotten everything, uh, prepared.
This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().

ref_count
release_target_info () \rightarrow None
Called just after this node has been marked up-to-date or was built completely.
This is where we try to release as many target node infos as possible for clean builds and update runs, in order to minimize the overall memory consumption.
By purging attributes that aren't needed any longer after a Node (=File) got built, we don't have to care that much how many KBytes a Node actually requires...as long as we free the memory shortly afterwards.
@see: built() and File.release_target_info()
remove ()
Remove this Node: no-op by default.
render_include_tree ()
Return a text representation, suitable for displaying to the user, of the include tree for the sources of this node.
reset_executor () \rightarrow None
Remove cached executor; forces recompute when needed.
retrieve_from_cache () \rightarrow bool
Try to retrieve the node's content from a cache
This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().
Returns true if the node was successfully retrieved.
rexists ()
Does this node exist locally or in a repository?
scan () \rightarrow None
Scan this node's dependents for implicit dependencies.
scanner_key ()
sconsign () \rightarrow None
An Alias is not recorded in .sconsign files
select_scanner (scanner)
Selects a scanner for this Node.
This is a separate method so it can be overridden by Node subclasses (specifically, Node.FS.Dir) that must use their own Scanner and don't select one the Scanner.Selector that's configured for the target.
set_always_build (always_build: int = 1) \rightarrow None
Set the Node's always_build value.
set_executor (executor: Executor) \rightarrow None
Set the action executor for this node.
set_explicit (is_explicit) \rightarrow None
set_nocache (nocache: int = 1) \rightarrow None
Set the Node's nocache value.
set_noclean (noclean: int = 1) \rightarrow None
Set the Node's noclean value.
set_precious (precious: int = 1) \rightarrow None
Set the Node's precious value.
set_pseudo (pseudo: bool = True) \rightarrow None
Set the Node's pseudo value.
set_specific_source (source) \rightarrow None
set_state (state) \rightarrow None
side_effect
side_effects
sources
sources_set
state
store_info
str_for_display ()
target_peers
visited () → None
   Called just after this node has been visited (with or without a build).
waiting_parents
waiting_s_e
wkids

class SCons.Node.Alias.AliasBuildInfo
Bases: BuildInfoBase
__getstate__() → None
   Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a
   '__dict__' slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all
   instances of a class.
__setstate__(state) → None
   Restore the attributes from a pickled state.
bact
bactsig
bdepends
bdependsigs
bimplicit
bimplicitsigs
bsources
bsourcesigs
current_version_id = 2
merge (other) → None
   Merge the fields of another object into this object. Already existing information is overwritten by the other instance's
data. WARNING: If a '__dict__' slot is added, it should be updated instead of replaced.
class SCons.Node.Alias.AliasNameSpace (dict=None, /, **kwargs)
Bases: UserDict
Alias (name, **kw)
   _abc_impl = <_abc._abc_data object>
clear () → None. Remove all items from D.
copy ()
   classmethod fromkeys (iterable, value=None)
   get (k[, d]) → D[k] if k in D, else d. d defaults to None.
   items () → a set-like object providing a view on D's items
   keys () → a set-like object providing a view on D's keys
   lookup (name, **kw)
      pop (k[, d]) → v, remove specified key and return the corresponding value.
      If key is not found, d is returned if given, otherwise KeyError is raised.
      popitem () → (k, v), remove and return some (key, value) pair
      as a 2-tuple; but raise KeyError if D is empty.
      setdefault (k[, d]) → D.get(k,d), also set D[k] = d if k not in D
      update ([, E], **F) → None. Update D from mapping/iterable E and F.
      If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for
      (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
   values () → an object providing a view on D's values
Bases: NodeInfoBase
__getstate__() → None
   Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a
   '__dict__' slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all
   instances of a class.
__setstate__(state) → None
   Restore the attributes from a pickled state.
convert (node, val) → None
   csig
SCons.compat package

```python
current_version_id = 2
field_list = ['csig']
format(field_list=None, names: int = 0)
merge(other) → None
  Merge the fields of another object into this object. Already existing information is overwritten by the other instance’s data. WARNING: If a '__dict__' slot is added, it should be updated instead of replaced.
```

SCons.Node.FS module

File system nodes.

These Nodes represent the canonical external objects that people think of when they think of building software: files and directories.

This holds a “default_fs” variable that should be initialized with an FS that can be used by scripts or modules looking for the canonical default.

```python
class SCons.Node.FS.Base (name, directory, fs)
  Bases: Node
  A generic class for file system entries. This class is for when we don’t know yet whether the entry being looked up is a file or a directory. Instances of this class can morph into either Dir or File objects by a later, more precise lookup.
  Note: this class does not define __cmp__ and __hash__ for efficiency reasons. SCons does a lot of comparing of Node.FS.{Base,Entry,File,Dir} objects, so those operations must be as fast as possible, which means we want to use Python’s built-in object identity comparisons.
```

```python
class Attrs
  Bases: object
  shared
BuildInfo
  alias of BuildInfoBase
Decider (function) → None
GetTag (key)
  Return a user-defined tag.
NodeInfo
  alias of NodeInfoBase
RDirs (pathlist)
  Search for a list of directories in the Repository list.
Rfindalldirs (pathlist)
  Return all of the directories for a given path list, including corresponding “backing” directories in any repositories.
  The Node lookups are relative to this Node (typically a directory), so memoizing result saves cycles from looking up the same path for each target in a given directory.
Tag (key, value) → None
  Add a user-defined tag.
_Rfindalldirs_key (pathlist)
__getattr__ (attr)
  Together with the node_bwcomp dict defined below, this method provides a simple backward compatibility layer for the Node attributes ‘abspath’, ‘labspath’, ‘path’, ‘tpath’, ‘suffix’ and ‘path_elements’. These Node attributes used to be directly available in v2.3 and earlier, but have been replaced by getter methods that initialize the single variables lazily when required, in order to save memory. The redirection to the getters lets older Tools and SConstruct continue to work without any additional changes, fully transparent to the user. Note, that __getattr__ is only called as fallback when the requested attribute can’t be found, so there should be no speed performance penalty involved for standard builds.
__lt__ (other)
  less than operator used by sorting on py3
__str__ () → str
  A Node.FS.Base object’s string representation is its path name.
```
SCons.compat package

```python
_add_child (collection, set, child) → None
    Adds 'child' to 'collection', first checking 'set' to see if it's already present.
_children_get ()
_children_reset () → None
_func_exists
_func_get_contents
_func_is_derived
_func_reexists
_func_sconsign
_func_target_from_source
_get_scanner (env, initial_scanner, root_node_scanner, kw)
_get_str ()
glob1 (pattern, ondisk: bool = True, source: bool = False, strings: bool = False)
labspath
_local
_memo
_path
_path_elements
_proxy
_save_str ()
_specific_sources
_tags
_tpath
add_dependency (depend)
    Adds dependencies.
add_ignore (depend)
    Adds dependencies to ignore.
add_prerequisite (prerequisite) → None
    Adds prerequisites
add_source (source)
    Adds sources.
add_to_implicit (deps) → None
add_to_waiting_parents (node) → int
    Returns the number of nodes added to our waiting parents list: 1 if we add a unique waiting parent, 0 if not. (Note that the returned values are intended to be used to increment a reference count, so don't think you can "clean up" this function by using True and False instead...)
add_to_waiting_s_e (node) → None
add_wkid (wkid) → None
    Add a node to the list of kids waiting to be evaluated
all_children (scan: int = 1)
    Return a list of all the node's direct children.
alter_targets ()
    Return a list of alternate targets for this Node.
always_build
attributes
binfo
build (**kw)
    Actually build the node.
    This is called by the Taskmaster after it's decided that the Node is out-of-date and must be rebuilt, and after the prepare() method has gotten everything, uh, prepared.
    This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().
builder
builder_set (builder) → None
built () → None
    Called just after this node is successfully built.
```
SCons.compat package

cached
changed (node=None, allowcache: bool = False)
    Returns if the node is up-to-date with respect to the BuildInfo stored last time it was built. The default behavior is to
    compare it against our own previously stored BuildInfo, but the stored BuildInfo from another Node (typically one in
    a Repository) can be used instead.
    Note that we now always check every dependency. We used to short-circuit the check by returning as soon as we
detected any difference, but we now rely on checking every dependency to make sure that any necessary Node
    information (for example, the content signature of an #included .h file) is updated.
    The allowcache option was added for supporting the early release of the executor/builder structures, right after a
File target was built. When set to true, the return value of this changed method gets cached for File nodes. Like
this, the executor isn’t needed any longer for subsequent calls to changed().
@see: FS.File.changed(), FS.File.release_target_info()

changed_since_last_build
check_attributes (name)
    Simple API to check if the node.attributes for name has been set

children (scan: int = 1)
    Return a list of the node’s direct children, minus those that are ignored by this node.

children_are_up_to_date () → bool
    Alternate check for whether the Node is current: If all of our children were up-to-date, then this Node was
    up-to-date, too.

clear () → None
    Completely clear a Node of all its cached state (so that it can be re-evaluated by interfaces that do continuous
    integration builds).

clear_memoized_values () → None

cwd

del_binfo () → None
    Delete the build info from this node.

depends

depends_set

dir

disambiguate (must_exist=None)

duplicate

env
    env_set (env, safe: bool = False) → None

executor
 executor_cleanup () → None
    Let the executor clean up any cached information.

exists ()
    Reports whether node exists.

explain ()

for_signature ()
    Return a string representation of the Node that will always be the same for this particular Node, no matter what.
This is by contrast to the __str__() method, which might, for instance, return a relative path for a file Node. The
    purpose of this method is to generate a value to be used in signature calculation for the command line used to
    build a target, and we use this method instead of str() to avoid unnecessary rebuilds. This method does not need to
    return something that would actually work in a command line; it can return any kind of nonsense, so long as it does
    not change.

fs
    Reference to parent Node.FS object

get_abspath ()
    Get the absolute path of the file.

get_binfo ()
    Fetch a node’s build information.
    node - the node whose sources will be collected cache - alternate node to use for the signature cache returns - the
    build signature
This no longer handles the recursive descent of the node's children's signatures. We expect that they're already built and updated by someone else, if that's what's wanted.

```python
def get_build_env()
    Fetch the appropriate Environment to build this node.

def get_build_scanner_path(scanner)
    Fetch the appropriate scanner path for this node.

def get_builder(default_builder=None)
    Return the set builder, or a specified default value

def get_cachedir_csig()

def get_contents()
    Fetch the contents of the entry.

def get_csig()

def get_dir()

def get_env()

def get_env_scanner(env, kw={})

def get_executor(create: int = 1) -> Executor
    Fetch the action executor for this node. Create one if there isn't already one, and requested to do so.

def get_found_includes(env, scanner, path)
    Return the scanned include lines (implicit dependencies) found in this node.
    The default is no implicit dependencies. We expect this method to be overridden by any subclass that can be scanned for implicit dependencies.

def get_implicit_deps(env, initial_scanner, path_func, kw={})
    Return a list of implicit dependencies for this node.
    This method exists to handle recursive invocation of the scanner on the implicit dependencies returned by the scanner, if the scanner's recursive flag says that we should.

def get_internal_path()

def get_labspath()
    Get the absolute path of the file.

def get_ninfo()

def get_path(dir=None)
    Return path relative to the current working directory of the Node.FS.Base object that owns us.

def get_path_elements()

def get_relpath()
    Get the path of the file relative to the root SConstruct file's directory.

def get_source_scanner(node)
    Fetch the source scanner for the specified node
    NOTE: "self" is the target being built, "node" is the source file for which we want to fetch the scanner.
    Implies self.has_builder() is true; again, expect to only be called from locations where this is already verified.
    This function may be called very often; it attempts to cache the scanner found to improve performance.

def get_state()

def get_stored_implicit()
    Fetch the stored implicit dependencies

def get_stored_info()

def get_string(for_signature)
    This is a convenience function designed primarily to be used in command generators (i.e., CommandGeneratorActions or Environment variables that are callable), which are called with a for_signature argument that is nonzero if the command generator is being called to generate a signature for the command line, which determines if we should rebuild or not.
    Such command generators should use this method in preference to str(Node) when converting a Node to a string, passing in the for_signature parameter, such that we will call Node.for_signature() or str(Node) properly, depending on whether we are calculating a signature or actually constructing a command line.

def get_subst_proxy()
    This method is expected to return an object that will function exactly like this Node, except that it implements any additional special features that we would like to be in effect for Environment variable substitution. The principle use is that some Nodes would like to implement a __getattr__() method, but putting that in the Node type itself has a
tendency to kill performance. We instead put it in a proxy and return it from this method. It is legal for this method to
return self if no new functionality is needed for Environment substitution.

get_suffix ()
get_target_scanner ()
get_tpath ()
getmtime ()
getsize ()

has_builder () → bool
Return whether this Node has a builder or not.
In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly ("if
node.builder: ..."). When the builder attribute is examined directly, it ends up calling __getattr__ for both the
__len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and
slowing things down immensely.

has_explicit_builder () → bool
Return whether this Node has an explicit builder.
This allows an internal Builder created by SCons to be marked non-explicit, so that it can be overridden by an
explicit builder that the user supplies (the canonical example being directories).

ignore
ignore_set
implicit
implicit_set
includes

is_conftest () → bool
Returns true if this node is an conftest node

is_derived () → bool
Returns true if this node is derived (i.e. built).
This should return true only for nodes whose path should be in the variant directory when duplicate=0 and should
contribute their build signatures when they are used as source files to other derived files. For example: source with
source builders are not derived in this sense, and hence should not return true.

is_explicit
is_literal () → bool
Always pass the string representation of a Node to the command interpreter literally.

is_sconscript () → bool
Returns true if this node is an sconscript

is_under (dir) → bool

is_up_to_date () → bool
Default check for whether the Node is current: unknown Node subtypes are always out of date, so they will always
get built.

isdir () → bool
isfile () → bool
islink () → bool
linked

lstat ()

make_ready () → None
Get a Node ready for evaluation.
This is called before the Taskmaster decides if the Node is up-to-date or not. Overriding this method allows for a
Node subclass to be disambiguated if necessary, or for an implicit source builder to be attached.

missing () → bool

multiple_side_effect_has_builder () → bool
Return whether this Node has a builder or not.
In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly ("if
node.builder: ..."). When the builder attribute is examined directly, it ends up calling __getattr__ for both the
__len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and
slowing things down immensely.

must_be_same (klass)
This node, which already existed, is being looked up as the specified klass. Raise an exception if it isn’t.
name
new_binfo ()
new_ninfo ()
ninfo
nocache
clean
postprocess () \rightarrow None

Clean up anything we don’t need to hang onto after we’ve been built.

prepare ()

Prepare for this Node to be built.
This is called after the Taskmaster has decided that the Node is out-of-date and must be rebuilt, but before actually calling the method to build the Node.
This default implementation checks that explicit or implicit dependencies either exist or are derived, and initializes the BuildInfo structure that will hold the information about how this node is, uh, built.
(The existence of source files is checked separately by the Executor, which aggregates checks for all of the targets built by a specific action.)
Overriding this method allows for a Node subclass to remove the underlying file from the file system. Note that subclass methods should call this base class method to get the child check and the BuildInfo structure.

prerequisites

pseudo
push_to_cache () \rightarrow None
Try to push a node into a cache
ref_count
release_target_info () \rightarrow None
Called just after this node has been marked up-to-date or was built completely.
This is where we try to release as many target node infos as possible for clean builds and update runs, in order to minimize the overall memory consumption.
By purging attributes that aren’t needed any longer after a Node (=File) got built, we don’t have to care that much how many KBytes a Node actually requires…as long as we free the memory shortly afterwards.
@see: built() and File.release_target_info()

remove ()
Remove this Node: no-op by default.

render_include_tree ()
Return a text representation, suitable for displaying to the user, of the include tree for the sources of this node.

rentry ()
reset_executor () \rightarrow None
Remove cached executor; forces recompute when needed.

retrieve_from_cache () \rightarrow bool
Try to retrieve the node’s content from a cache
This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().
Returns true if the node was successfully retrieved.

rexists ()

Does this node exist locally or in a repository?

rfile ()
rstr () \rightarrow str
A Node.FS.Base object’s string representation is its path name.

sbuilder
scan () \rightarrow None
Scan this node’s dependents for implicit dependencies.

scanner_key ()
select_scanner (scanner)
Selects a scanner for this Node.
This is a separate method so it can be overridden by Node subclasses (specifically, Node.FS.Dir) that must use their own Scanner and don’t select one the Scanner.Selector that’s configured for the target.
set_always_build(always_build: int = 1) → None
Set the Node’s always_build value.

set_executor(executor: Executor) → None
Set the action executor for this node.

set_explicit(is_explicit) → None

set_local() → None

set_nocache(nocache: int = 1) → None
Set the Node’s nocache value.

set_noclean(noclean: int = 1) → None
Set the Node’s noclean value.

set_precious(precious: int = 1) → None
Set the Node’s precious value.

set_pseudo(pseudo: bool = True) → None
Set the Node’s pseudo value.

set_specific_source(source) → None
Set the source code builder for this node.

set_state(state) → None

side_effect
side_effects
sources
sources_set
src_builder()
Fetch the source code builder for this node.
If there isn’t one, we cache the source code builder specified for the directory (which in turn will cache the value
from its parent directory, and so on up to the file system root).

crnode()
If this node is in a build path, return the node corresponding to its source file. Otherwise, return ourselves.

stat()
state
store_info
str_for_display()

target_from_source(prefix, suffix[, splitext=<function splitext>])
Generates a target entry that corresponds to this entry (usually a source file) with the specified prefix and suffix.
Note that this method can be overridden dynamically for generated files that need different behavior. See
Tool/swig.py for an example.
target_peers
visited() → None
Called just after this node has been visited (with or without a build).

waiting_parents
waiting_s_e
wkids

class SCons.Node.FS.Dir(name, directory, fs)
Bases: Base
A class for directories in a file system.
class Attrs
Bases: object
shared
BuildInfo
  alias of DirBuildInfo
Decider(function) → None
Dir(name, create: bool = True)
  Looks up or creates a directory node named ‘name’ relative to this directory.
Entry(name)
  Looks up or creates an entry node named ‘name’ relative to this directory.
File(name)
Looks up or creates a file node named ‘name’ relative to this directory.

GetTag (key)
Return a user-defined tag.

NodeInfo
alias of DirNodeInfo

RDirs (pathlist)
Search for a list of directories in the Repository list.

Rfindalldirs (pathlist)
Return all of the directories for a given path list, including corresponding “backing” directories in any repositories. The Node lookups are relative to this Node (typically a directory), so memoizing result saves cycles from looking up the same path for each target in a given directory.

Tag (key, value) → None
Add a user-defined tag.

_Rfindalldirs_key (pathlist)
__clearRepositoryCache (duplicate=**None**) → **None**
Called when we change the repository(ies) for a directory. This clears any cached information that is invalidated by changing the repository.

getattr (attr)
Together with the node_bwcomp dict defined below, this method provides a simple backward compatibility layer for the Node attributes ‘abspath’, ‘labspath’, ‘path’, ‘t_path’, ‘suffix’ and ‘path_elements’. These Node attributes used to be directly available in v2.3 and earlier, but have been replaced by getter methods that initialize the single variables lazily when required, in order to save memory. The redirection to the getters lets older Tools and SConstruct continue to work without any additional changes, fully transparent to the user. Note, that __getattr__ is only called as fallback when the requested attribute can’t be found, so there should be no speed performance penalty involved for standard builds.

__lt__ (other)
less than operator used by sorting on py3

__resetDuplicate (node) → None

__str__() → str
A Node.FS.Base object’s string representation is its path name.

_abspath

_add_child (collection, set, child) → None
Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.

_children_get ()

_children_reset () → None

_create ()
Create this directory, silently and without worrying about whether the builder is the default or not.

__func_exists

__func_get_contents

__func_is_derived

__func_rexists

__func_sconsign

__func_target_from_source

_get_scanner (env, initial_scanner, root_node_scanner, kw)

_get_str ()

_glob1 (pattern, ondisk: bool = True, source: bool = False, strings: bool = False)
Globs for and returns a list of entry names matching a single pattern in this directory.
This searches any repositories and source directories for corresponding entries and returns a Node (or string) relative to the current directory if an entry is found anywhere.

TODO: handle pattern with no wildcard. Python’s glob.glob uses a separate __glob0 function to do this.

_labsopath

_local

_memo

_morph () → None
Turn a file system Node (either a freshly initialized directory object or a separate Entry object) into a proper directory object.
Set up this directory’s entries and hook it into the file system tree. Specify that directories (this Node) don’t use signatures for calculating whether they’re current.

_addPath
_addPathElements
_proxy
_relPathKey (other)
_saveStr ()
_sconsign
_specificSources
_scdirFindFileKey (filename)
_tags
_tpath
addRepository (dir) → None
add_dependency (depend)
  Adds dependencies.
add_ignore (depend) → None
  Adds dependencies to ignore.
add_prerequisite (prerequisite) → None
  Adds prerequisites
add_source (source)
  Adds sources.
add_to_implicit (deps) → None
add_to_waiting_parents (node) → int
  Returns the number of nodes added to our waiting parents list: 1 if we add a unique waiting parent, 0 if not. (Note that the returned values are intended to be used to increment a reference count, so don’t think you can “clean up” this function by using True and False instead…)
add_to_waiting_s_e (node) → None
add_wkid (wkid) → None
  Add a node to the list of kids waiting to be evaluated
all_children (scan: int = 1)
  Return a list of all the node’s direct children.
alter_targets ()
  Return any corresponding targets in a variant directory.
always_build
attributes
binfo
build (**kw) → None
  A null “builder” for directories.
builder
builder_set (builder) → None
built () → None
  Called just after this node is successfully built.
cached
cachedir_csig
cachesig
changed (node=None, allowcache: bool = False)
  Returns if the node is up-to-date with respect to the BuildInfo stored last time it was built. The default behavior is to compare it against our own previously stored BuildInfo, but the stored BuildInfo from another Node (typically one in a Repository) can be used instead.
Note that we now always check every dependency. We used to short-circuit the check by returning as soon as we detected any difference, but we now rely on checking every dependency to make sure that any necessary Node information (for example, the content signature of an #included .h file) is updated.
The allowcache option was added for supporting the early release of the executor/builder structures, right after a File target was built. When set to true, the return value of this changed method gets cached for File nodes. Like this, the executor isn’t needed any longer for subsequent calls to changed().
@see: FS.File.changed(), FS.File.release_target_info()
SCons.compat package

class Node:
  
  changed_since_last_build
  check_attributes(name)
    Simple API to check if the node attributes for name has been set
  children(scan: int = 1)
    Return a list of the node’s direct children, minus those that are ignored by this node.
  children_are_up_to_date() → bool
    Alternate check for whether the Node is current: If all of our children were up-to-date, then this Node was up-to-date, too.
  clear() → None
    Completely clear a Node of all its cached state (so that it can be re-evaluated by interfaces that do continuous integration builds).
  clear_memoized_values() → None
  cwd
    Delete the build info from this node.
  depends
  depends_set
  dir
  dir_on_disk(name)
  dirname
  disambiguate(must_exist=None)
  diskcheck_match() → None
  do_duplicate(src) → None
  duplicate
  entries
  entry_abspath(name)
  entry_exists_on_disk(name)
    Searches through the file/dir entries of the current directory, and returns True if a physical entry with the given name could be found.
    @see reentry_exists_on_disk
  entry_labspath(name)
  entry_path(name)
  entry_tpath(name)
  env
  env_set(env, safe: bool = False) → None
  executor
  executor_cleanup() → None
    Let the executor clean up any cached information.
  exists()
    Reports whether node exists.
  explain()
  for_signature()
    Return a string representation of the Node that will always be the same for this particular Node, no matter what.
    This is by contrast to the __str__() method, which might, for instance, return a relative path for a file Node. The purpose of this method is to generate a value to be used in signature calculation for the command line used to build a target, and we use this method instead of str() to avoid unnecessary rebuilds. This method does not need to return something that would actually work in a command line; it can return any kind of nonsense, so long as it does not change.
  fs
    Reference to parent Node.FS object
  getRepositories()
    Returns a list of repositories for this directory.
  get_abspath() → str
Get the absolute path of the file.
get_all_rdirs ()
get_binfo ()
Fetch a node’s build information.
node - the node whose sources will be collected cache - alternate node to use for the signature cache returns - the build signature
This no longer handles the recursive descent of the node’s children’s signatures. We expect that they’re already built and updated by someone else, if that’s what’s wanted.
get_build_env ()
Fetch the appropriate Environment to build this node.
get_build_scanner_path (scanner)
Fetch the appropriate scanner path for this node.
get_builder (default_builder=None)
Return the set builder, or a specified default value
get_cachedir_csig ()
get_contents ()
Return content signatures and names of all our children separated by new-lines. Ensure that the nodes are sorted.
get_csig ()
Compute the content signature for Directory nodes. In general, this is not needed and the content signature is not stored in the DirNodeInfo. However, if get_contents on a Dir node is called which has a child directory, the child directory should return the hash of its contents.
get_dir ()
get_env ()
get_env_scanner (env, kw={})
get_executor (create: int = 1) -> Executor
Fetch the action executor for this node. Create one if there isn’t already one, and requested to do so.
get_found_includes (env, scanner, path)
Return this directory’s implicit dependencies.
We don’t bother caching the results because the scan typically shouldn’t be requested more than once (as opposed to scanning .h file contents, which can be requested as many times as the files is #included by other files).
get_implicit_deps (env, initial_scanner, path_func, kw=())
Return a list of implicit dependencies for this node.
This method exists to handle recursive invocation of the scanner on the implicit dependencies returned by the scanner, if the scanner’s recursive flag says that we should.
get_internal_path ()
get_labspath () -> str
Get the absolute path of the file.
get_ninfo ()
get_path (dir=None)
Return path relative to the current working directory of the Node.FS.Base object that owns us.
get_path_elements ()
get_relpath ()
Get the path of the file relative to the root SConstruct file’s directory.
get_source_scanner (node)
Fetch the source scanner for the specified node
NOTE: “self” is the target being built, “node” is the source file for which we want to fetch the scanner.
Implies self.has_builder() is true; again, expect to only be called from locations where this is already verified.
This function may be called very often; it attempts to cache the scanner found to improve performance.
get_state ()
get_stored_implicit ()
Fetch the stored implicit dependencies
get_stored_info ()
get_string (for_signature)
This is a convenience function designed primarily to be used in command generators (i.e., CommandGeneratorActions or Environment variables that are callable), which are called with a for_signature
argument that is nonzero if the command generator is being called to generate a signature for the command line, which determines if we should rebuild or not.

Such command generators should use this method in preference to str(Node) when converting a Node to a string, passing in the for_signature parameter, such that we will call Node.for_signature() or str(Node) properly, depending on whether we are calculating a signature or actually constructing a command line.

get_subst_proxy ()

This method is expected to return an object that will function exactly like this Node, except that it implements any additional special features that we would like to be in effect for Environment variable substitution. The principle use is that some Nodes would like to implement a __getattr__() method, but putting that in the Node type itself has a tendency to kill performance. We instead put it in a proxy and return it from this method. It is legal for this method to return self if no new functionality is needed for Environment substitution.

glob (pathname, ondisk: bool = True, source: bool = False, strings: bool = False, exclude=None) → list

Returns a list of Nodes (or strings) matching a pathname pattern.
Pathname patterns follow POSIX shell syntax:

```
*      matches everything
?      matches any single character
[seq]  matches any character in seq (ranges allowed)
[!seq] matches any char not in seq
```

The wildcard characters can be escaped by enclosing in brackets. A leading dot is not matched by a wildcard, and needs to be explicitly included in the pattern to be matched. Matches also do not span directory separators.
The matches take into account Repositories, returning a local Node if a corresponding entry exists in a Repository (either an in-memory Node or something on disk). The underlying algorithm is adapted from a rather old version of glob.glob() function in the Python standard library (heavily modified), and uses fnmatch.fnmatch() under the covers.
This is the internal implementation of the external Glob API.

Parameters:

- **pattern** – pathname pattern to match.
- **ondisk** – if false, restricts matches to in-memory Nodes. By default, matches entries that exist on-disk in addition to in-memory Nodes.
- **source** – if true, corresponding source Nodes are returned if globbing in a variant directory. The default behavior is to return Nodes local to the variant directory.
- **strings** – if true, returns the matches as strings instead of Nodes. The strings are path names relative to this directory.
- **exclude** – if not None, must be a pattern or a list of patterns following the same POSIX shell semantics. Elements matching at least one pattern from exclude will be excluded from the result.

has_builder () → bool

Return whether this Node has a builder or not.
In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly ("if node.builder: ..."). When the builder attribute is examined directly, it ends up calling __getattr__ for both the
\_\_len\_ and \_\_bool\_ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.

\texttt{has\_explicit\_builder () \rightarrow bool}

Return whether this Node has an explicit builder.

This allows an internal Builder created by SCons to be marked non-explicit, so that it can be overridden by an explicit builder that the user supplies (the canonical example being directories).

\texttt{ignore}

\texttt{ignore\_set}

\texttt{implicit}

\texttt{implicit\_set}

\texttt{includes}

\texttt{is\_confest () \rightarrow bool}

Returns true if this node is an \texttt{confest} node

\texttt{is\_derived () \rightarrow bool}

Returns true if this node is derived (i.e. built).

This should return true only for nodes whose path should be in the variant directory when \texttt{duplicate=0} and should contribute their build signatures when they are used as source files to other derived files. For example: source with source builders are not derived in this sense, and hence should not return true.

\texttt{is\_explicit}

\texttt{is\_literal () \rightarrow bool}

Always pass the string representation of a Node to the command interpreter literally.

\texttt{is\_sconscript () \rightarrow bool}

Returns true if this node is an \texttt{sconscript}

\texttt{is\_under (dir) \rightarrow bool}

\texttt{is\_up\_to\_date () \rightarrow bool}

If any child is not up-to-date, then this directory isn’t, either.

\texttt{isd\_isdir () \rightarrow bool}

\texttt{is\_file () \rightarrow bool}

\texttt{is\_link () \rightarrow bool}

\texttt{link (sr\_dir, duplicate) \rightarrow None}

Set this directory as the variant directory for the supplied source directory.

\texttt{linked}

\texttt{lstat ()}

\texttt{make\_ready () \rightarrow None}

Get a Node ready for evaluation.

This is called before the Taskmaster decides if the Node is up-to-date or not. Overriding this method allows for a Node subclass to be disambiguated if necessary, or for an implicit source builder to be attached.

\texttt{missing () \rightarrow bool}

\texttt{multiple\_side\_effect\_has\_builder ()}

Return whether this Node has a builder or not.

In Boolean tests, this turns out to be a \texttt{lot} more efficient than simply examining the builder attribute directly (“if node.builder: …”). When the builder attribute is examined directly, it ends up calling \texttt{\_\_getattr\_} for both the \texttt{\_\_len\_} and \texttt{\_\_bool\_} attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.

\texttt{must\_be\_same (klass)}

This node, which already existed, is being looked up as the specified \texttt{klass}. Raise an exception if it isn’t.

\texttt{name}

\texttt{new\_binfo ()}

\texttt{new\_ninfo ()}

\texttt{ninfo}

\texttt{nocache}

\texttt{noclean}

\texttt{on\_disk\_entries}

\texttt{postprocess () \rightarrow None}

Clean up anything we don’t need to hang onto after we’ve been built.

\texttt{precious}
prepare () \rightarrow None
Prepare for this Node to be built.
This is called after the Taskmaster has decided that the Node is out-of-date and must be rebuilt, but before actually calling the method to build the Node.
This default implementation checks that explicit or implicit dependencies either exist or are derived, and initializes the BuildInfo structure that will hold the information about how this node is, uh, built.
(The existence of source files is checked separately by the Executor, which aggregates checks for all of the targets built by a specific action.)
Overriding this method allows for a Node subclass to remove the underlying file from the file system. Note that subclass methods should call this base class method to get the child check and the BuildInfo structure.

prerequisites
pseudo

push_to_cache () \rightarrow None
Try to push a node into a cache

rdir ()

ref_count

rel_path (other)
Return a path to “other” relative to this directory.

release_target_info () \rightarrow None
Called just after this node has been marked up-to-date or was built completely.
This is where we try to release as many target node infos as possible for clean builds and update runs, in order to minimize the overall memory consumption.
By purging attributes that aren’t needed any longer after a Node (=File) got built, we don’t have to care that much how many KBytes a Node actually requires...as long as we free the memory shortly afterwards.
@see: built() and File.release_target_info()

released_target_info
remove ()
Remove this Node: no-op by default.

render_include_tree ()
Return a text representation, suitable for displaying to the user, of the include tree for the sources of this node.

rentry ()

rentry_exists_on_disk (name)
Searches through the file/dir entries of the current and all its remote directories (repos), and returns True if a physical entry with the given name could be found. The local directory (self) gets searched first, so repositories take a lower precedence regarding the searching order.
@see entry_exists_on_disk

repositories
reset_executor () \rightarrow None
Remove cached executor; forces recompute when needed.

retrieve_from_cache () \rightarrow bool
Try to retrieve the node’s content from a cache
This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().
Returns true if the node was successfully retrieved.

rexists ()
Does this node exist locally or in a repository?

rfile ()

root

rstr () \rightarrow str
A Node.FS.Base object’s string representation is its path name.

sbuilder

scan () \rightarrow None
Scan this node’s dependents for implicit dependencies.

scanner_key ()
A directory does not get scanned.
sconsign ()
    Return the .sconsign file info for this directory.
searched
select_scanner (scanner)
    Selects a scanner for this Node.
    This is a separate method so it can be overridden by Node subclasses (specifically, Node.FS.Dir) that must use
    their own Scanner and don’t select one the Scanner.Selector that’s configured for the target.
set_always_build (always_build: int = 1) → None
    Set the Node’s always_build value.
set_executor (executor: Executor) → None
    Set the action executor for this node.
set_explicit (is_explicit) → None
set_local () → None
set_nocache (nocache: int = 1) → None
    Set the Node’s nocache value.
set_noclean (noclean: int = 1) → None
    Set the Node’s noclean value.
set_precious (precious: int = 1) → None
    Set the Node’s precious value.
set_pseudo (pseudo: bool = True) → None
    Set the Node’s pseudo value.
set_specific_source (source) → None
set_src_builder (builder) → None
    Set the source code builder for this node.
set_state (state) → None
side_effect
side_effects
sources
sources_set
src_builder ()
    Fetch the source code builder for this node.
    If there isn’t one, we cache the source code builder specified for the directory (which in turn will cache the value
    from its parent directory, and so on up to the file system root).
srcdir
srcdir_duplicate (name)
srcdir_find_file (filename)
srcdir_list ()
srcnode ()
    Dir has a special need for srcnode()…if we have a srcdir attribute set, then that is our srcnode.
stat ()
state
store_info
str_for_display ()
target_from_source (prefix, suffix, splitext=<function splitext>)
    Generates a target entry that corresponds to this entry (usually a source file) with the specified prefix and suffix.
    Note that this method can be overridden dynamically for generated files that need different behavior. See
    Tool/swig.py for an example.
target_peers
up ()
variant_dirs
visited () → None
    Called just after this node has been visited (with or without a build).
waiting_parents
waiting_s_e
walk (func, arg) → None
    Walk this directory tree by calling the specified function for each directory in the tree.
This behaves like the os.path.walk() function, but for in-memory Node.FS.Dir objects. The function takes the same arguments as the functions passed to os.path.walk():

```
func(arg, dirname, fnames)
```

Except that “dirname” will actually be the directory `Node`, not the string. The ‘.’ and ‘..’ entries are excluded from fnames. The fnames list may be modified in-place to filter the subdirectories visited or otherwise impose a specific order. The “arg” argument is always passed to func() and may be used in any way (or ignored, passing None is common).

```
wkids
class SCons.Node.FS.DirBuildInfo
Bases: BuildInfoBase
__getstate__ ()
Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a '__dict__' slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all instances of a class.
__setstate__ (state) → None
Restore the attributes from a pickled state.
```

```
bact
bactsig
bdepends
bdependssigs
bimplicit
bimplicitssigs
bsources
bsourcesigs
current_version_id = 2
merge (other) → None
Merge the fields of another object into this object. Already existing information is overwritten by the other instance’s data. WARNING: If a '__dict__' slot is added, it should be updated instead of replaced.
```

```
class SCons.Node.FS.DirNodeInfo
Bases: NodeInfoBase
__getstate__ ()
Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a '__dict__' slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all instances of a class.
__setstate__ (state) → None
Restore the attributes from a pickled state. The version is discarded.
convert (node, val) → None
format (field_list=field_list, names: int = 0)
fs = None
merge (other) → None
Merge the fields of another object into this object. Already existing information is overwritten by the other instance’s data. WARNING: If a '__dict__' slot is added, it should be updated instead of replaced.
str_to_node (s)
update (node) → None
```

```
class SCons.Node.FS.DiskChecker (disk_check_type, do_check_function, ignore_check_function)
Bases: object
Implement disk check variation.
This Class will hold functions to determine what this particular disk checking implementation should do when enabled or disabled.
```

```
enable (disk_check_type_list) → None
If the current object’s disk_check_type matches any in the list passed :param disk_check_type_list: List of disk checks to enable .return:
```

```
class SCons.Node.FS.Entry (name, directory, fs)
Bases: Base
```
This is the class for generic Node.FS entries—that is, things that could be a File or a Dir, but we're just not sure yet. Consequently, the methods in this class really exist just to transform their associated object into the right class when the time comes, and then call the same-named method in the transformed class.

class Attrs
  Bases: object

BuildInfo
  alias of BuildInfoBase

Decider (function) → None
  GetTag (key)
    Return a user-defined tag.

NodeInfo
  alias of NodeInfoBase

RDdirs (pathlist)
  Search for a list of directories in the Repository list.

Rfindalldirs (pathlist)
  Return all of the directories for a given path list, including corresponding “backing” directories in any repositories.
  The Node lookups are relative to this Node (typically a directory), so memoizing result saves cycles from looking up the same path for each target in a given directory.

Tag (key, value) → None
  Add a user-defined tag.

_Rfindalldirs_key (pathlist)

__getattr__ (attr)
  Together with the node_bwcomp dict defined below, this method provides a simple backward compatibility layer for the Node attributes ‘abspath’, ‘labspath’, ‘path’, ‘tpath’, ‘suffix’ and ‘path_elements’. These Node attributes used to be directly available in v2.3 and earlier, but have been replaced by getter methods that initialize the single variables lazily when required, in order to save memory. The redirection to the getters lets older Tools and SConstruct continue to work without any additional changes, fully transparent to the user. Note, that __getattr__ is only called as fallback when the requested attribute can’t be found, so there should be no speed performance penalty involved for standard builds.

__lt__ (other)
  less than operator used by sorting on py3

__str__ () → str
  A Node.FS.Base object’s string representation is its path name.

_abspath

_add_child (collection, set, child) → None
  Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.

_children_get ()
   _children_reset () → None

_func_exists

_func_get_contents

_func_is_derived

_func_rexists

_func_sconsign

_func_target_from_source

_get_scanner (env, initial_scanner, root_node_scanner, kw)

_get_str ()

_glob1 (pattern, ondisk: bool = True, source: bool = False, strings: bool = False)

_labspath

_local

_memo

_path

_path_elements

_proxy

_save_str ()

_sconsign
add_dependency (depend)
    Adds dependencies.
add_ignore (depend)
    Adds dependencies to ignore.
add_prerequisite (prerequisite) → None
    Adds prerequisites
add_source (source)
    Adds sources.
add_to_implicit (deps) → None
add_to_waiting_parents (node) → int
    Returns the number of nodes added to our waiting parents list: 1 if we add a unique waiting parent, 0 if not. (Note that the returned values are intended to be used to increment a reference count, so don’t think you can “clean up” this function by using True and False instead…)
add_to_waiting_s_e (node) → None
add_wkid (wkid) → None
    Add a node to the list of kids waiting to be evaluated
all_children (scan: int = 1)
    Return a list of all the node’s direct children.
alter_targets ()
    Return a list of alternate targets for this Node.
always_build
attributes
binfo
build (**kw)
    Actually build the node.
    This is called by the Taskmaster after it’s decided that the Node is out-of-date and must be rebuilt, and after the prepare() method has gotten everything, uh, prepared.
    This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().
builder
builder_set (builder) → None
built () → None
    Called just after this node is successfully built.
cached
cachedir_csig
cachesig
changed (node=None, allowcache: bool = False)
    Returns if the node is up-to-date with respect to the BuildInfo stored last time it was built. The default behavior is to compare it against our own previously stored BuildInfo, but the stored BuildInfo from another Node (typically one in a Repository) can be used instead.
    Note that we now always check every dependency. We used to short-circuit the check by returning as soon as we detected any difference, but we now rely on checking every dependency to make sure that any necessary Node information (for example, the content signature of an #included .h file) is updated.
    The allowcache option was added for supporting the early release of the executor/builder structures, right after a File target was built. When set to true, the return value of this changed method gets cached for File nodes. Like this, the executor isn’t needed any longer for subsequent calls to changed().
    @see: FS.File.changed(), FS.File.release_target_info()
changed_since_last_build
check_attributes (name)
    Simple API to check if the node.attributes for name has been set
children (scan: int = 1)
    Return a list of the node’s direct children, minus those that are ignored by this node.
Alternate check for whether the Node is current: If all of our children were up-to-date, then this Node was up-to-date, too.


clear () → None

Completely clear a Node of all its cached state (so that it can be re-evaluated by interfaces that do continuous integration builds).

clear_memoized_values () → None

contentsig

cwd
del_binfo () → None

Delete the build info from this node.
depends
depends_set
dir
dirname
disambiguate (must_exist=None)
diskcheck_match () → None

duplicate
entries

env
env_set (env, safe: bool = False) → None
executor
executor_cleanup () → None

Let the executor clean up any cached information.
exists ()

Reports whether node exists.
explain ()

for_signature ()

Return a string representation of the Node that will always be the same for this particular Node, no matter what. This is by contrast to the __str__() method, which might, for instance, return a relative path for a file Node. The purpose of this method is to generate a value to be used in signature calculation for the command line used to build a target, and we use this method instead of str() to avoid unnecessary rebuilds. This method does not need to return something that would actually work in a command line; it can return any kind of nonsense, so long as it does not change.

fs

Reference to parent Node.FS object
get_abspath ()

Get the absolute path of the file.

get_binfo ()

Fetch a node’s build information.
node - the node whose sources will be collected cache - alternate node to use for the signature cache returns - the build signature

This no longer handles the recursive descent of the node’s children’s signatures. We expect that they’re already built and updated by someone else, if that’s what’s wanted.

get_build_env ()

Fetch the appropriate Environment to build this node.

get_build_scanner_path (scanner)

Fetch the appropriate scanner path for this node.

get_builder (default_builder=None)

Return the set builder, or a specified default value

get_cachedir_csig ()
get_contents ()

Fetch the contents of the entry. Returns the exact binary contents of the file.

get_csig ()
get_dir ()
get_env ()
get_env_scanner (env, kw={})
get_executor (create: int = 1) → Executor
Fetch the action executor for this node. Create one if there isn’t already one, and requested to do so.
get_found_includes (env, scanner, path)
Return the scanned include lines (implicit dependencies) found in this node.
The default is no implicit dependencies. We expect this method to be overridden by any subclass that can be scanned for implicit dependencies.
get_implicit_deps (env, initial_scanner, path_func, kw={})
Return a list of implicit dependencies for this node.
This method exists to handle recursive invocation of the scanner on the implicit dependencies returned by the scanner, if the scanner’s recursive flag says that we should.
get_internal_path ()
get_labspath ()
Get the absolute path of the file.
get_ninfo ()
get_path (dir=None)
Return path relative to the current working directory of the Node.FS.Base object that owns us.
get_path_elements ()
get_relpath ()
Get the path of the file relative to the root SConstruct file’s directory.
get_source_scanner (node)
Fetch the source scanner for the specified node
NOTE: “self” is the target being built, “node” is the source file for which we want to fetch the scanner.
Implies self.has_builder() is true; again, expect to only be called from locations where this is already verified.
This function may be called very often; it attempts to cache the scanner found to improve performance.
get_state ()
get_stored_implicit ()
Fetch the stored implicit dependencies
get_stored_info ()
get_string (for_signature)
This is a convenience function designed primarily to be used in command generators (i.e., CommandGeneratorActions or Environment variables that are callable), which are called with a for_signature argument that is nonzero if the command generator is being called to generate a signature for the command line, which determines if we should rebuild or not.
Such command generators should use this method in preference to str(Node) when converting a Node to a string, passing in the for_signature parameter, such that we will call Node.for_signature() or str(Node) properly, depending on whether we are calculating a signature or actually constructing a command line.
get_subst_proxy ()
This method is expected to return an object that will function exactly like this Node, except that it implements any additional special features that we would like to be in effect for Environment variable substitution. The principle use is that some Nodes would like to implement a __getattr__() method, but putting that in the Node type itself has a tendency to kill performance. We instead put it in a proxy and return it from this method. It is legal for this method to return self if no new functionality is needed for Environment substitution.
get_suffix ()
get_target_scanner ()
get_text_contents () → str
Fetch the decoded text contents of a Unicode encoded Entry.
Since this should return the text contents from the file system, we check to see into what sort of subclass we should morph this Entry.
get_path ()
getmtime ()
getsize ()
has_builder () → bool
Return whether this Node has a builder or not.
In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly (“if node.builder: …”). When the builder attribute is examined directly, it ends up calling __getattr__ for both the
SCons.comapt package

__len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.

has_explicit_builder () → bool
    Return whether this Node has an explicit builder.
    This allows an internal Builder created by SCons to be marked non-explicit, so that it can be overridden by an explicit builder that the user supplies (the canonical example being directories).

ignore
ignore_set
implicit
implicit_set
includes
is_conftest () → bool
    Returns true if this node is a conftest node
is Derived () → bool
    Returns true if this node is derived (i.e. built).
    This should return true only for nodes whose path should be in the variant directory when duplicate=0 and should contribute their build signatures when they are used as source files to other derived files. For example: source with source builders are not derived in this sense, and hence should not return true.

is_explicit
is_literal () → bool
    Always pass the string representation of a Node to the command interpreter literally.

is_scons script () → bool
    Returns true if this node is a scons script

is_under (dir) → bool

is_up_to_date () → bool
    Default check for whether the Node is current: unknown Node subtypes are always out of date, so they will always get built.

isdir () → bool
isfile () → bool
islink () → bool
linked
lstat ()

make_ready () → None
    Get a Node ready for evaluation.
    This is called before the Taskmaster decides if the Node is up-to-date or not. Overriding this method allows for a Node subclass to be disambiguated if necessary, or for an implicit source builder to be attached.

missing () → bool
multiple_side_effect_has_builder () → bool
    Return whether this Node has a builder or not.
    In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly ("if node.builder: ... "). When the builder attribute is examined directly, it ends up calling __getattr__ for both the __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.

must_be_same (klass) → None
    Called to make sure a Node is a Dir. Since we're an Entry, we can morph into one.

name
new_binfo ()
new_ninfo ()
ninfo
nocache
noclean
on_disk_entries
postprocess () → None
    Clean up anything we don't need to hang onto after we've been built.

precious
prepare ()

217
Prepare for this Node to be built.
This is called after the Taskmaster has decided that the Node is out-of-date and must be rebuilt, but before actually calling the method to build the Node.
This default implementation checks that explicit or implicit dependencies either exist or are derived, and initializes the BuildInfo structure that will hold the information about how this node is, uh, built.
(The existence of source files is checked separately by the Executor, which aggregates checks for all of the targets built by a specific action.)
Overriding this method allows for a Node subclass to remove the underlying file from the file system. Note that subclass methods should call this base class method to get the child check and the BuildInfo structure.

prerequisites
pseudo
push_to_cache () → None
Try to push a node into a cache
ref_count
rel_path (other)
release_target_info () → None
Called just after this node has been marked up-to-date or was built completely.
This is where we try to release as many target node infos as possible for clean builds and update runs, in order to minimize the overall memory consumption.
By purging attributes that aren't needed any longer after a Node (=File) got built, we don't have to care that much how many KBytes a Node actually requires...as long as we free the memory shortly afterwards.
@see: built() and File.release_target_info()
released_target_info
remove ()
Remove this Node: no-op by default.
render_include_tree ()
Return a text representation, suitable for displaying to the user, of the include tree for the sources of this node.
rentry ()
repositories
reset_executor () → None
Remove cached executor; forces recompute when needed.
retrieve_from_cache () → bool
Try to retrieve the node's content from a cache
This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().
Returns true if the node was successfully retrieved.
rexists ()
Does this node exist locally or in a repository?
rfile ()
We're a generic Entry, but the caller is actually looking for a File at this point, so morph into one.
root
rstr () → str
A Node.FS.Base object's string representation is its path name.
sbuilder
scan () → None
Scan this node's dependents for implicit dependencies.
scanner_key ()
scanner_paths
searched
select_scanner (scanner)
Selects a scanner for this Node.
This is a separate method so it can be overridden by Node subclasses (specifically, Node.FS.Dir) that must use their own Scanner and don't select one the Scanner.Selector that's configured for the target.
set_always_build (always_build: int = 1) → None
Set the Node's always_build value.
set_executor (executor: Executor) → None
SCons.compat package

Set the action executor for this node.
\[ \text{set}\_\text{explicit}(\text{is}\_\text{explicit}) \rightarrow \text{None} \]
\[ \text{set}\_\text{local}() \rightarrow \text{None} \]
\[ \text{set}\_\text{nocache}(\text{nocache: }\text{int}=1) \rightarrow \text{None} \]
Set the Node’s nocache value.
\[ \text{set}\_\text{noclean}(\text{noclean: }\text{int}=1) \rightarrow \text{None} \]
Set the Node’s noclean value.
\[ \text{set}\_\text{precious}(\text{precious: }\text{int}=1) \rightarrow \text{None} \]
Set the Node’s precious value.
\[ \text{set}\_\text{pseudo}(\text{pseudo: }\text{bool}=\text{True}) \rightarrow \text{None} \]
Set the Node’s pseudo value.
\[ \text{set}\_\text{specific}\_\text{source}(\text{source}) \rightarrow \text{None} \]
\[ \text{set}\_\text{src}\_\text{builder}(\text{builder}) \rightarrow \text{None} \]
Set the source code builder for this node.
\[ \text{set}\_\text{state}(\text{state}) \rightarrow \text{None} \]
\[ \text{side}\_\text{effect} \]
\[ \text{side}\_\text{effects} \]
\[ \text{sources} \]
\[ \text{sources}\_\text{set} \]
\[ \text{src}\_\text{builder}() \]
Fetch the source code builder for this node.
If there isn’t one, we cache the source code builder specified for the directory (which in turn will cache the value from its parent directory, and so on up to the file system root).
\[ \text{srcdir} \]
\[ \text{srcnode}() \]
If this node is in a build path, return the node corresponding to its source file. Otherwise, return ourself.
\[ \text{stat}() \]
\[ \text{state} \]
\[ \text{store}\_\text{info} \]
\[ \text{str}\_\text{for}\_\text{display}() \]
\[ \text{target}\_\text{from}\_\text{source}(\text{prefix, suffix, splitext=<function splitext>} \}) \]
Generates a target entry that corresponds to this entry (usually a source file) with the specified prefix and suffix.
Note that this method can be overridden dynamically for generated files that need different behavior. See Tool/swig.py for an example.
\[ \text{target}\_\text{peers} \]
\[ \text{variant}\_\text{dirs} \]
\[ \text{visited}() \rightarrow \text{None} \]
Called just after this node has been visited (with or without a build).
\[ \text{waiting}\_\text{parents} \]
\[ \text{waiting}\_\text{s_e} \]
\[ \text{wkids} \]
class SCons.Node.FS.EntryProxy(\text{subject})
Bases: Proxy
\[ \_\_\text{get}\_\text{abspath}() \]
\[ \_\_\text{get}\_\text{base}\_\text{path}() \]
Return the file’s directory and file name, with the suffix stripped.
\[ \_\_\text{get}\_\text{dir}() \]
\[ \_\_\text{get}\_\text{file}() \]
\[ \_\_\text{get}\_\text{filebase}() \]
\[ \_\_\text{get}\_\text{posix}\_\text{path}() \]
Return the path with / as the path separator, regardless of platform.
\[ \_\_\text{get}\_\text{relpath}() \]
\[ \_\_\text{get}\_\text{rsrdir}() \]
Returns the directory containing the source node linked to this node via VariantDir(), or the directory of this node if not linked.
\[ \_\_\text{get}\_\text{rsrnode}() \]
__get_srcdir ()
Returns the directory containing the source node linked to this node via VariantDir(), or the directory of this node if
not linked.
__get_srsnode ()
__get_suffix ()
__get_windows_path ()
Return the path with as the path separator, regardless of platform.

dictSpecialAttrs = {'abspath': <function EntryProxy.__get_abspath>, 'base': <function
EntryProxy.__get_base_path>, 'dir': <function EntryProxy.__get_dir>, 'file': <function EntryProxy.__get_file>,
'filebase': <function EntryProxy.__get_filebase>, 'posix': <function EntryProxy.__get_posix_path>, 'relpath': <function
EntryProxy.__get_relpath>, 'rsrtdir': <function EntryProxy.__get_rsrcdir>, 'rsrtpath': <function
EntryProxy.__get_rsrcnode>, 'suffix': <function EntryProxy.__get_suffix>, 'win32': <function
EntryProxy.__get_windows_path>, 'windows': <function EntryProxy.__get_windows_path>}

get ()
Retrieve the entire wrapped object

eXception SCons.Node.FS.EntryProxyAttributeError (entry_proxy, attribute)
Bases: AttributeError
An AttributeError subclass for recording and displaying the name of the underlying Entry involved in an AttributeError
exception.
add_note ()
Exception.add_note(note) – add a note to the exception
args
name
attribute name
obj
object
with_traceback ()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.
class SCons.Node.FS.FS (path=None)
Bases: LocalFS
Dir (name, directory=None, create: bool = True)
Look up or create a Dir node with the specified name. If the name is a relative path (begins with ., ../, or a file
name), then it is looked up relative to the supplied directory node, or to the top level directory of the FS (supplied at
construction time) if no directory is supplied.
This method will raise TypeError if a normal file is found at the specified path.
Entry (name, directory=None, create: bool = True)
Look up or create a generic Entry node with the specified name. If the name is a relative path (begins with ., ../, or
a file name), then it is looked up relative to the supplied directory node, or to the top level directory of the FS
(supplied at construction time) if no directory is supplied.
File (name, directory=None, create: bool = True)
Look up or create a File node with the specified name. If the name is a relative path (begins with ., ../, or a file
name), then it is looked up relative to the supplied directory node, or to the top level directory of the FS (supplied at
construction time) if no directory is supplied.
This method will raise TypeError if a directory is found at the specified path.
Glob (pathname, ondisk: bool = True, source: bool = True, strings: bool = False, exclude=None, cwd=None)
Globs
This is mainly a shim layer
PyPackageDir (modulename) → Dir | None
Locate the directory of Python module modulename.
For example ‘SCons’ might resolve to Windows: C:Python311Libsite-packagesSCons Linux:
/usr/lib64/python3.11/site-packages/SCons
Can be used to determine a toolpath based on a Python module name.
This is the backend called by the public API function PyPackageDir().
Repository (*dirs) → None

SCons.compat package
Specify Repository directories to search.

```python
VariantDir (variant_dir, src_dir, duplicate: int = 1)
```

Link the supplied variant directory to the source directory for purposes of building files.

```python
_lookup (p, directory, fsclass, create: bool = True)
```

The generic entry point for Node lookup with user-supplied data.

This translates arbitrary input into a canonical Node.FS object of the specified fsclass. The general approach for strings is to turn it into a fully normalized absolute path and then call the root directory’s lookup_abs() method for the heavy lifting.

- If the path name begins with ‘#’, it is unconditionally interpreted relative to the top-level directory of this FS. ‘#’ is treated as a synonym for the top-level SConstruct directory, much like ‘~’ is treated as a synonym for the user’s home directory in a UNIX shell. So both ‘#foo’ and ‘#/foo’ refer to the ‘foo’ subdirectory underneath the top-level SConstruct directory.
- If the path name is relative, then the path is looked up relative to the specified directory, or the current directory (self._cwd, typically the SConscript directory) if the specified directory is None.

```python
chdir (dir, change_os_dir: bool = False)
```

Change the current working directory for lookups. If change_os_dir is true, we will also change the “real” cwd to match.

```python
chmod (path, mode)
copy (src, dst)
copy2 (src, dst)
extists (path)
get_max_drift ()
get_root (drive)
```

Returns the root directory for the specified drive, creating it if necessary.

```python
getcwd ()
getmtime (path)
getsize (path)
isdir (path) → bool
isfile (path) → bool
islink (path) → bool
link (src, dst)
listdir (path)
llstat (path)
makedirs (path, mode: int = 511, exist_ok: bool = False)
mkdir (path, mode: int = 511)
open (path)
readlink (file) → str
rename (old, new)
scandir (path)
set_SConstruct_dir (dir) → None
set_max_drift (max_drift) → None
stat (path)
symlink (src, dst)
unlink (path)
```

```python
variant_dir_target_climb (orig, dir, tail)
```

Create targets in corresponding variant directories.

Climb the directory tree, and look up path names relative to any linked variant directories we find.

- Even though this loops and walks up the tree, we don’t memoize the return value because this is really only used to process the command-line targets.

```python
class SCons.Node.FS.File (name, directory, fs)
```

Bases: Base

A class for files in a file system.

```python
class Attrs
```

Bases: object

shared

BuildInfo
alias of FileBuildInfo

Decider (function) \rightarrow\ None

Dir (name, create: bool = True)
Create a directory node named ‘name’ relative to the directory of this file.

Dirs (pathlist)
Create a list of directories relative to the SConscript directory of this file.

Entry (name)
Create an entry node named ‘name’ relative to the directory of this file.

File (name)
Create a file node named ‘name’ relative to the directory of this file.

GetTag (key)
Return a user-defined tag.

NodeInfo
alias of FileNodeInfo

RDirs (pathlist)
Search for a list of directories in the Repository list.

Rfindalldirs (pathlist)
Return all of the directories for a given path list, including corresponding “backing” directories in any repositories.

The Node lookups are relative to this Node (typically a directory), so memoizing result saves cycles from looking up the same path for each target in a given directory.

Tag (key, value) \rightarrow\ None
Add a user-defined tag.

_Rfindalldirs_key (pathlist)
__dmap_cache = {}
__dmap_sig_cache = {}
__getattr__ (attr)
Together with the node_bwcomp dict defined below, this method provides a simple backward compatibility layer for the Node attributes ‘abspath’, ‘labspath’, ‘path’, ‘tpath’, ‘suffix’ and ‘path_elements’. These Node attributes used to be directly available in v2.3 and earlier, but have been replaced by getter methods that initialize the single variables lazily when required, in order to save memory. The redirection to the getters lets older Tools and SConstruct continue to work without any additional changes, fully transparent to the user. Note, that __getattr__ is only called as fallback when the requested attribute can’t be found, so there should be no speed performance penalty involved for standard builds.

__lt__ (other)
less than operator used by sorting on py3
__str__ () \rightarrow\ str
A Node.FS.Base object’s string representation is its path name.

_abspath

_add_child (collection, set, child) \rightarrow\ None
Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.

_add_strings_to_dependency_map (dmap)
In the case comparing node objects isn’t sufficient, we’ll add the strings for the nodes to the dependency map.

_build_dependency_map (binfo)
Build mapping from file -> signature

Parameters:

• self (self) –

• considered (binfo - buildinfo from node being) –

Returns: dictionary of file->signature mappings

_children_get ()

_children_reset () \rightarrow None

_createDir () \rightarrow None

_func_exists

_func_get_contents

_func_is_derived
_func_rexists
_func_sconsign
_func_target_from_source
_get_found_includes_key (env, scanner, path)
_get_previous_signatures (dmap)

Return a list of corresponding csigs from previous build in order of the node/files in children.

Parameters:

- self (self --)
- csig (dmap - Dictionary of file ->)

Returns: List of csigs for provided list of children

_get_scanner (env, initial_scanner, root_node_scanner, kw)
_get_str ()
_glob1 (pattern, ondisk: bool = True, source: bool = False, strings: bool = False)
_labspath
_local
_memo
_morph () → None

Turn a file system node into a File object.

_path
_path_elements
_proxy
_rmv_existing ()
_save_str ()
_sconsign
_specific_sources
_tags
_tpath

add_dependency (depend)

Adds dependencies.

add_ignore (depend)

Adds dependencies to ignore.

add_prerequisite (prerequisite) → None

Adds prerequisites.

add_source (source)

Adds sources.

add_to_implicit (deps) → None

add_to_waiting_parents (node) → int

Returns the number of nodes added to our waiting parents list: 1 if we add a unique waiting parent, 0 if not. (Note that the returned values are intended to be used to increment a reference count, so don’t think you can “clean up” this function by using True and False instead…)

add_to_waiting_s_e (node) → None

Add a node to the list of kids waiting to be evaluated

all_children (scan: int = 1)

Return a list of all the node’s direct children.

alter_targets ()

Return any corresponding targets in a variant directory.

always_build
attributes
binfo
build (**kw)

Actually build the node.
This is called by the Taskmaster after it’s decided that the Node is out-of-date and must be rebuilt, and after the prepare() method has gotten everything, uh, prepared.
This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().

builder

builder_set (builder) → None

build () → None

Called just after this File node is successfully built.

Just like for ‘release_target_info’ we try to release some more target node attributes in order to minimize the overall memory consumption.

@see: release_target_info
cached
cachedir_csig
cachesig

changed (node=None, allowcache: bool = False) → bool

Returns if the node is up-to-date with respect to the BuildInfo stored last time it was built.

For File nodes this is basically a wrapper around Node.changed(), but we allow the return value to get cached after the reference to the Executor got released in release_target_info().

@see: Node.changed()

changed_content (target, prev_ni, repo_node=None) → bool

changed_since_last_build

changed_timestamp_match (target, prev_ni, repo_node=None) → bool

Return True if the timestamps don't match or if there is no previous timestamp

:call param: target: :call param prev_ni:

Information about the node from the previous build

:returns:

changed_timestamp_newer (target, prev_ni, repo_node=None) → bool

changed_timestamp_then_content (target, prev_ni, node=None) → bool

Used when deciding for file is Timestamp-MD5

NOTE: If the timestamp hasn’t changed this will skip md5'ing the

file and just copy the prev_ni provided. If the prev_ni is wrong. It will propagate it. See: https://github.com/SCons/scons/issues/2980

Parameters:

- dependency (self) –
- target (target) –
- .sconsign (prev_ni - The NodeInfo object loaded from previous builds) –
- existence/timestamp (node - Node instance. Check this node for file) – if specified.

Returns:

Boolean - Indicates if node(File) has changed.

check_attributes (name)

Simple API to check if the node.attributes for name has been set

children (scan: int = 1)

Return a list of the node’s direct children, minus those that are ignored by this node.

children_are_up_to_date () → bool

Alternate check for whether the Node is current: If all of our children were up-to-date, then this Node was up-to-date, too.


clear () → None

Completely clear a Node of all its cached state (so that it can be re-evaluated by interfaces that do continuous integration builds).

clear_memoized_values () → None

cconvert_copy_attrs = ['bsources', 'bimplicit', 'bdepends', 'bact', 'bactsig', 'ninfo']

cconvert_old_entry (old_entry)

cconvert_sig_attrs = ['bsourcesigs', 'bimplicit_sigs', 'bdependsigs']

cwd
del_binfo () → None
Delete the build info from this node.
depends
depends_set
dir
dirname
disambiguate (must_exist=None)
diskcheck_match () → None
do_duplicate (src)
duplicate
entries
eqv
eqv_set (env, safe: bool = False) → None
executor
executor_cleanup () → None
Let the executor clean up any cached information.
exists ()
Reports whether node exists.
explain ()
find_repo_file ()
For this node, find if there exists a corresponding file in one or more repositories :return: list of corresponding files in repositories
find_src_builder ()
for_signature ()
Return a string representation of the Node that will always be the same for this particular Node, no matter what. This is by contrast to the __str__() method, which might, for instance, return a relative path for a file Node. The purpose of this method is to generate a value to be used in signature calculation for the command line used to build a target, and we use this method instead of str() to avoid unnecessary rebuilds. This method does not need to return something that would actually work in a command line; it can return any kind of nonsense, so long as it does not change.
fs
Reference to parent Node.FS object
get_abspath ()
Get the absolute path of the file.
get_binfo ()
Fetch a node's build information.
  node - the node whose sources will be collected cache - alternate node to use for the signature cache returns - the build signature
  This no longer handles the recursive descent of the node's children's signatures. We expect that they're already built and updated by someone else, if that's what's wanted.
get_build_env ()
Fetch the appropriate Environment to build this node.
get_build_scanner_path (scanner)
Fetch the appropriate scanner path for this node.
get_builder (default_builder=None)
Return the set builder, or a specified default value
get_cachedir_bsig ()
Return the signature for a cached file, including its children.
It adds the path of the cached file to the cache signature, because multiple targets built by the same action will all have the same build signature, and we have to differentiate them somehow.
Signature should normally be string of hex digits.
get_cachedir_csig ()
Fetch a Node's content signature for purposes of computing another Node's cachesig.
This is a wrapper around the normal get_csig() method that handles the somewhat obscure case of using CacheDir with the -n option. Any files that don’t exist would normally be “built” by fetching them from the cache, but the normal get_csig() method will try to open up the local file, which doesn’t exist because the -n option meant we
didn’t actually pull the file from cachedir. But since the file *does* actually exist in the cachedir, we can use its contents for the csig.

```python
def get_content_hash() → str:
    Compute and return the hash for this file.

def get_contents() → bytes:
    Return the contents of the file as bytes.

def get_contents_sig() -> str:
    A helper method for get_cachedir_bsig.
    It computes and returns the signature for this node’s contents.

def get_csig() → str:
    Generate a node’s content signature.

def get_dir() -> str:

def get_env():

def get_env_scanner(env, kw={}) -> Executor:
    Fetch the action executor for this node. Create one if there isn’t already one, and requested to do so.

```
is that some Nodes would like to implement a __getattr__() method, but putting that in the Node type itself has a
tendency to kill performance. We instead put it in a proxy and return it from this method. It is legal for this method to
return self if no new functionality is needed for Environment substitution.

get_suffix ()
geget_text_contents () → str
   Return the contents of the file as text.
get_timestamp () → int
get_tpath ()
getmtime ()
getsize ()
has_builder () → bool
   Return whether this Node has a builder or not.
   In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly ("if
node.builder: …"). When the builder attribute is examined directly, it ends up calling __getattr__ for both the
__len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and
slowing things down immensely.
has_explicit_builder () → bool
   Return whether this Node has an explicit builder.
   This allows an internal Builder created by SCons to be marked non-explicit, so that it can be overridden by an
explicit builder that the user supplies (the canonical example being directories).
has_src_builder () → bool
   Return whether this Node has a source builder or not.
   If this Node doesn’t have an explicit source code builder, this is where we figure out, on the fly, if there’s a
transparent source code builder for it.
   Note that if we found a source builder, we also set the self.builder attribute, so that all of the methods that actually
build this file don’t have to do anything different.

hash_chunksize = 65536
ignore
ignore_set
implicit
implicit_set
includes
is_conftest () → bool
   Returns true if this node is an conftest node
is_derived () → bool
   Returns true if this node is derived (i.e. built).
   This should return true only for nodes whose path should be in the variant directory when duplicate=0 and should
contribute their build signatures when they are used as source files to other derived files. For example: source with
source builders are not derived in this sense, and hence should not return true.
is_explicit
is_literal () → bool
   Always pass the string representation of a Node to the command interpreter literally.
is_sconscript () → bool
   Returns true if this node is an sconscript
is_under (dir) → bool
is_up_to_date () → bool
   Check for whether the Node is current.
   In all cases self is the target we’re checking to see if it’s up to date
isdirent () → bool
isfile () → bool
islink () → bool
linked
lstat ()
make_ready () → None
   Get a Node ready for evaluation.
missing() → bool

multiple_side_effect_has_builder() → bool

Return whether this Node has a builder or not.

In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly ("if node.builder: ... "). When the builder attribute is examined directly, it ends up calling __getattr__ for both the __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.

must_be_same(klass)

This node, which already existed, is being looked up as the specified klass. Raise an exception if it isn’t.

name
new_binfo()

new_ninfo()
ninfo
nocache
noclean
on_disk_entries
postprocess() → None

Clean up anything we don’t need to hang onto after we’ve been built.

precious
prepare()

prerequisites
pseudo
push_to_cache() → None

Try to push the node into a cache

ref_count
rel_path(other)

release_target_info() → None

Called just after this node has been marked up-to-date or was built completely.

This is where we try to release as many target node infos as possible for clean builds and update runs, in order to minimize the overall memory consumption.

We’d like to remove a lot more attributes like self.sources and self.sources_set, but they might get used in a next build step. For example, during configuration the source files for a built E(*).o file are used to figure out which linker to use for the resulting Program (gcc vs. g++)! That’s why we check for the ‘keep_targetinfo’ attribute, config Nodes and the Interactive mode just don’t allow an early release of most variables.

In the same manner, we can’t simply remove the self.attributes here. The smart linking relies on the shared flag, and some parts of the java Tool use it to transport information about nodes...

@see: built() and Node.release_target_info()

released_target_info
remove()

Remove this file.

render_include_tree()

Return a text representation, suitable for displaying to the user, of the include tree for the sources of this node.

rentry()

repositories
reset_executor() → None

Remove cached executor; forces recompute when needed.

retrieve_from_cache() → bool

Try to retrieve the node’s content from a cache

This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().

Returns True if the node was successfully retrieved.

rexists()

Does this node exist locally or in a repository?
A Node.FS.Base object's string representation is its path name.

Scan this node's dependents for implicit dependencies.

Select a scanner for this Node.

Selects a scanner for this Node.

This is a separate method so it can be overridden by Node subclasses (specifically, Node.FS.Dir) that must use their own Scanner and don't select one the ScannerSelector that's configured for the target.

Set the Node's always_build value.

Set the action executor for this node.

Set the source code builder for this node.

Fetch the source code builder for this node.

If there isn't one, we cache the source code builder specified for the directory (which in turn will cache the value from its parent directory, and so on up to the file system root).

If this node is in a build path, return the node corresponding to its source file. Otherwise, return ourself.

Generates a target entry that corresponds to this entry (usually a source file) with the specified prefix and suffix.

Note that this method can be overridden dynamically for generated files that need different behavior. See Tool/swig.py for an example.

Called just after this node has been visited (with or without a build).

Called just after this node has been visited (with or without a build).
Waiting for sconsign.

class SCons.Node.FS.FileBuildInfo
Bases: BuildInfoBase
This is info loaded from sconsign.

Attributes unique to FileBuildInfo:

dependency_map : Caches file->csig mapping
for all dependencies. Currently this is only used when using MD5-timestamp decider. It’s used to ensure that we copy the correct csig from the previous build to be written to .sconsign when current build is done. Previously the matching of csig to file was strictly by order they appeared in bdepends, bsources, or bimplicit, and so a change in order or count of any of these could yield writing wrong csig, and then false positive rebuilds

__getstate__() → None
Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a '__dict__' slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all instances of a class.

__setstate__(state) → None
Restore the attributes from a pickled state.

bact
bactsig
bdepends
bdependsigs
bimplicit
bimplicitsigs
bsources
bsourcesigs
convert_from_sconsign (dir, name) → None
Converts a newly-read FileBuildInfo object for in-SCons use
For normal up-to-date checking, we don’t have any conversion to perform—but we’re leaving this method here to make that clear.

convert_to_sconsign () → None
Converts this FileBuildInfo object for writing to a .sconsign file
This replaces each Node in our various dependency lists with its usual string representation: relative to the top-level SConstruct directory, or an absolute path if it’s outside.

current_version_id = 2
dependency_map
format(names: int = 0)
merge(other) → None
Merge the fields of another object into this object. Already existing information is overwritten by the other instance’s data. WARNING: If a '__dict__' slot is added, it should be updated instead of replaced.

prepare_dependencies () → None
Prepares a FileBuildInfo object for explaining what changed
The bsources, bdepends and bimplicit lists have all been stored on disk as paths relative to the top-level SConstruct directory. Convert the strings to actual Nodes (for use by the --debug=explain code and --implicit-cache).

exception SCons.Node.FS.FileBuildInfoFileToCsigMappingError
Bases: Exception
add_note ()
Exception.add_note(note) – add a note to the exception

args
with_traceback ()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class SCons.Node.FS.FileFinder
Bases: object
_find_file_key (filename, paths, verbose=None)
filedir_lookup(p, fd=None)
A helper method for find_file() that looks up a directory for a file we’re trying to find. This only creates the Dir Node if it exists on-disk, since if the directory doesn’t exist we know we won’t find any files in it... :-) It would be more compact to just use this as a nested function with a default keyword argument (see the commented-out version below), but that doesn’t work unless you have nested scopes, so we define it here just so this work under Python 1.5.2.

find_file(filename, paths, verbose=None)
Find a node corresponding to either a derived file or a file that exists already.
Only the first file found is returned, and none is returned if no file is found.
filename: A filename to find paths: A list of directory path nodes to search in. Can be represented as a list, a tuple, or a callable that is called with no arguments and returns the list or tuple.
returns The node created from the found file.

class SCons.Node.FS.FileNodeInfo
Bases: NodeInfoBase
__getstate__() Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a '__dict__' slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all instances of a class.
__setstate__(state) → None
Restore the attributes from a pickled state.
convert(node, val) → None
csig
current_version_id = 2
field_list = ['csig', 'timestamp', 'size']
format(field_list=field_list, names: int = 0)
fs = None
merge(other) → None
Merge the fields of another object into this object. Already existing information is overwritten by the other instance's data. WARNING: If a '__dict__' slot is added, it should be updated instead of replaced.
size
str_to_node(s)
timestamp
update(node) → None
SCons.Node.FS.LinkFunc (target, source, env) → int
Relative paths cause problems with symbolic links, so we use absolute paths, which may be a problem for people who want to move their soft-linked src-trees around. Those people should use the 'hard-copy' mode, softlinks cannot be used for that; at least I have no idea how ...
class SCons.Node.FS.LocalFS
Bases: object
This class implements an abstraction layer for operations involving a local file system. Essentially, this wraps any function in the os, os.path or shutil modules that we use to actually go do anything with or to the local file system. Note that there’s a very good chance we’ll refactor this part of the architecture in some way as we really implement the interface(s) for remote file system Nodes. For example, the right architecture might be to have this be a subclass instead of a base class. Nevertheless, we’re using this as a first step in that direction. We’re not using chdir() yet because the calling subclass method needs to use os.chdir() directly to avoid recursion. Will we really need this one?
chmod(path, mode)
copy(src, dst)
copy2(src, dst)
extists(path)
getmtime(path)
getsize(path)
isdir(path) → bool
isfile(path) → bool
islink(path) → bool
link(src, dst)
listdir (path)
lstat (path)
makedirs (path, mode: int = 511, exist_ok: bool = False)
mkdir (path, mode: int = 511)
open (path)
readlink (file) \to str
rename (old, new)
scandir (path)
stat (path)
symlink (src, dst)
unlink (path)
SCons.Node.FS.LocalString (target, source, env) \to str
SCons.Node.FS.MkdirFunc (target, source, env) \to int
class SCons.Node.FS.RootDir (drive, fs)
    Bases: Dir
    A class for the root directory of a file system.
    This is the same as a Dir class, except that the path separator ("\" or ") is actually part of the name, so we don't need to add a separator when creating the path names of entries within this directory.
class Attrs
    Bases: object
    shared
BuildInfo
    alias of DirBuildInfo
Decider (function) \to None
Dir (name, create: bool = True)
    Looks up or creates a directory node named 'name' relative to this directory.
Entry (name)
    Looks up or creates an entry node named 'name' relative to this directory.
File (name)
    Looks up or creates a file node named 'name' relative to this directory.
GetTag (key)
    Return a user-defined tag.
NodeInfo
    alias of DirNodeInfo
RDirs (pathlist)
    Search for a list of directories in the Repository list.
Rfindalldirs (pathlist)
    Return all of the directories for a given path list, including corresponding "backing" directories in any repositories.
The Node lookups are relative to this Node (typically a directory), so memoizing result saves cycles from looking up the same path for each target in a given directory.
Tag (key, value) \to None
    Add a user-defined tag.
_Rfindalldirs_key (pathlist)
__getattr__ (attr)
    Together with the node_bwcomp dict defined below, this method provides a simple backward compatibility layer for the Node attributes ‘abspath’, ‘labspath’, ‘path’, ‘tpath’, ‘suffix’ and ‘path_elements’. These Node attributes used to be directly available in v2.3 and earlier, but have been replaced by getter methods that initialize the single variables lazily when required, in order to save memory. The redirection to the getters lets older Tools and SConstruct continue to work without any additional changes, fully transparent to the user. Note, that __getattr__ is only called as fallback when the requested attribute can’t be found, so there should be no speed performance penalty involved for standard builds.
__lt__ (other)
    less than operator used by sorting on py3
_abspath
    __add_child (collection, set, child) \to None
    Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.
_create ()
Create this directory, silently and without worrying about whether the builder is the default or not.

_func_exists
_func_get_contents
_func_is_derived
_func_rexists
_func_sconsign
_func_target_from_source
_get_scanner (env, initial_scanner, root_node_scanner, kw)
_get_str ()
_glob1 (pattern, ondisk: bool = True, source: bool = False, strings: bool = False)
Globs for and returns a list of entry names matching a single pattern in this directory.
This searches any repositories and source directories for corresponding entries and returns a Node (or string)
relative to the current directory if an entry is found anywhere.
TODO: handle pattern with no wildcard. Python’s glob.glob uses a separate _glob0 function to do this.

_labspath
_local
_lookupDict
_lookup_abs (p, klass, create: bool = True)
Fast (?) lookup of a normalized absolute path.
This method is intended for use by internal lookups with already-normalized path data. For general-purpose
lookups, use the FS.Entry(), FS.Dir() or FS.File() methods.
The caller is responsible for making sure we’re passed a normalized absolute path; we merely let Python’s
dictionary look up and return the One True Node.FS object for the path.
If a Node for the specified “p” doesn’t already exist, and “create” is specified, the Node may be created after
recursive invocation to find or create the parent directory or directories.

_memo
_morph () → None
Turn a file system Node (either a freshly initialized directory object or a separate Entry object) into a proper
directory object.
Set up this directory’s entries and hook it into the file system tree. Specify that directories (this Node) don’t use
signatures for calculating whether they’re current.

_path
_path_elements
_proxy
_rel_path_key (other)
_save_str ()
_sconsign
_specific_sources
_srcdir_find_file_key (filename)
_tags
_tpath
.abspath
addRepository (dir) → None
add_dependency (depend)
Adds dependencies.
add_ignore (depend)
Adds dependencies to ignore.
add_prerequisite (prerequisite) → None
Adds prerequisites
add_source (source)
Adds sources.
add_to_implicit (deps) → None
add_to_waiting_parents (node) → int
Returns the number of nodes added to our waiting parents list: 1 if we add a unique waiting parent, 0 if not. (Note that the returned values are intended to be used to increment a reference count, so don’t think you can “clean up” this function by using True and False instead…)

```
add_to_waiting_s_e (node) → None
add_wkid (wkid) → None
```

Add a node to the list of kids waiting to be evaluated

```
all_children (scan: int = 1)
```

Return a list of all the node’s direct children.

```
alter_targets ()
```

Return any corresponding targets in a variant directory.

```
always_build
attributes
binfo
build (**kw) → None
```

A null “builder” for directories.

```
builder
builder_set (builder) → None
built () → None
```

Called just after this node is successfully built.

```
cached
cachedir_csig
cachesig
changed (node=None, allowcache: bool = False)
```

Returns if the node is up-to-date with respect to the BuildInfo stored last time it was built. The default behavior is to compare it against our own previously stored BuildInfo, but the stored BuildInfo from another Node (typically one in a Repository) can be used instead.

Note that we now always check every dependency. We used to short-circuit the check by returning as soon as we detected any difference, but we now rely on checking every dependency to make sure that any necessary Node information (for example, the content signature of an #included .h file) is updated.

The allowcache option was added for supporting the early release of the executor/builder structures, right after a File target was built. When set to true, the return value of this changed method gets cached for File nodes. Like this, the executor isn’t needed any longer for subsequent calls to changed().

@see: FS.File.changed(), FS.File.release_target_info()

```
changed_since_last_build
```

```
check_attributes (name)
```

Simple API to check if the node.attributes for name has been set

```
children (scan: int = 1)
```

Return a list of the node’s direct children, minus those that are ignored by this node.

```
children_are_up_to_date () → bool
```

Alternate check for whether the Node is current: If all of our children were up-to-date, then this Node was up-to-date, too.


```
clear () → None
```

Completely clear a Node of all its cached state (so that it can be re-evaluated by interfaces that do continuous integration builds).

```
clear_memoized_values () → None
```

```
contentsig
cwd
del_binfo () → None
```

Delete the build info from this node.

```
depends
depends_set
dir
dir_on_disk (name)
dirname
disambiguate (must_exist=None)
```
diskcheck_match () → None
do_duplicate (src) → None
duplicate
entries
entry_abspath (name)
entry_exists_on_disk (name)
    Searches through the file/dir entries of the current directory, and returns True if a physical entry with the given
    name could be found.
    @see entry_exists_on_disk
entry_labspath (name)
entry_path (name)
entry_tpath (name)
env
env_set (env, safe: bool = False) → None
executor
executor_cleanup () → None
    Let the executor clean up any cached information.
exists ()
    Reports whether node exists.
explain ()
file_on_disk (name)
for_signature ()
    Return a string representation of the Node that will always be the same for this particular Node, no matter what.
    This is by contrast to the __str__() method, which might, for instance, return a relative path for a file Node. The
    purpose of this method is to generate a value to be used in signature calculation for the command line used to
    build a target, and we use this method instead of str() to avoid unnecessary rebuilds. This method does not need to
    return something that would actually work in a command line; it can return any kind of nonsense, so long as it does
    not change.
fs
    Reference to parent Node.FS object
getRepositories ()
    Returns a list of repositories for this directory.
get_abspath () → str
    Get the absolute path of the file.
get_all_rdirs ()
get_binfo ()
    Fetch a node’s build information.
    node - the node whose sources will be collected cache - alternate node to use for the signature cache returns - the
    build signature
    This no longer handles the recursive descent of the node’s children’s signatures. We expect that they’re already
    built and updated by someone else, if that’s what’s wanted.
get_build_env ()
    Fetch the appropriate Environment to build this node.
get_build_scanner_path (scanner)
    Fetch the appropriate scanner path for this node.
get_builder (default_builder=None)
    Return the set builder, or a specified default value
get_cachedir_csig ()
get_contents ()
    Return content signatures and names of all our children separated by new-lines. Ensure that the nodes are sorted.
get_csig ()
    Compute the content signature for Directory nodes. In general, this is not needed and the content signature is not
    stored in the DirNodeInfo. However, if get_contents on a Dir node is called which has a child directory, the child
    directory should return the hash of its contents.
get_dir ()
get_env ()
SCons.compat package

```python
get_env_scanner (env, kw={})
get_executor (create: int = 1) → Executor
    Fetch the action executor for this node. Create one if there isn’t already one, and requested to do so.
get_found_includes (env, scanner, path)
    Return this directory’s implicit dependencies.
    We don’t bother caching the results because the scan typically shouldn’t be requested more than once (as opposed to scanning .h file contents, which can be requested as many times as the file is #included by other files).
get_implicit_deps (env, initial_scanner, path_func, kw={})
    Return a list of implicit dependencies for this node.
    This method exists to handle recursive invocation of the scanner on the implicit dependencies returned by the scanner, if the scanner’s recursive flag says that we should.
get_internal_path ()
get_labspath () → str
    Get the absolute path of the file.
get_ninfo ()
get_path (dir=None)
    Return path relative to the current working directory of the Node.FS.Base object that owns us.
get_path_elements ()
get_relpath ()
    Get the path of the file relative to the root SConstruct file’s directory.
get_source_scanner (node)
    Fetch the source scanner for the specified node
    NOTE: “self” is the target being built, “node” is the source file for which we want to fetch the scanner.
    Implies self.has_builder() is true; again, expect to only be called from locations where this is already verified.
    This function may be called very often; it attempts to cache the scanner found to improve performance.
get_state ()
get_stored_implicit ()
    Fetch the stored implicit dependencies
get_stored_info ()
get_string (for_signature)
    This is a convenience function designed primarily to be used in command generators (i.e., CommandGeneratorActions or Environment variables that are callable), which are called with a for_signature argument that is nonzero if the command generator is being called to generate a signature for the command line, which determines if we should rebuild or not.
    Such command generators should use this method in preference to str(Node) when converting a Node to a string, passing in the for_signature parameter, such that we will call Node.for_signature() or str(Node) properly, depending on whether we are calculating a signature or actually constructing a command line.
get_subst_proxy ()
    This method is expected to return an object that will function exactly like this Node, except that it implements any additional special features that we would like to be in effect for Environment variable substitution. The principle use is that some Nodes would like to implement a __getattr__() method, but putting that in the Node type itself has a tendency to kill performance. We instead put it in a proxy and return it from this method. It is legal for this method to return self if no new functionality is needed for Environment substitution.
get_suffix ()
get_target_scanner ()
get_text_contents ()
    We already emit things in text, so just return the binary version.
get_timestamp () → int
    Return the latest timestamp from among our children
get_path ()
getmtime ()
getsize ()
glob (pathname, ondisk: bool = True, source: bool = False, strings: bool = False, exclude=None) → list
    Returns a list of Nodes (or strings) matching a pathname pattern.
```
Pathname patterns follow POSIX shell syntax:

*  matches everything
?  matches any single character
[seq] matches any character in seq (ranges allowed)
[!seq] matches any char not in seq

The wildcard characters can be escaped by enclosing in brackets. A leading dot is not matched by a wildcard, and needs to be explicitly included in the pattern to be matched. Matches also do not span directory separators.

The matches take into account Repositories, returning a local Node if a corresponding entry exists in a Repository (either an in-memory Node or something on disk).

The underlying algorithm is adapted from a rather old version of glob.glob() function in the Python standard library (heavily modified), and uses fnmatch.fnmatch() under the covers.

This is the internal implementation of the external Glob API.

**Parameters:**

- **pattern** – pathname pattern to match.
- **ondisk** – if false, restricts matches to in-memory Nodes. By default, matches entries that exist on-disk in addition to in-memory Nodes.
- **source** – if true, corresponding source Nodes are returned if globbing in a variant directory. The default behavior is to return Nodes local to the variant directory.
- **strings** – if true, returns the matches as strings instead of Nodes. The strings are path names relative to this directory.
- **exclude** – if not None, must be a pattern or a list of patterns following the same POSIX shell semantics. Elements matching at least one pattern from exclude will be excluded from the result.

**has_builder () → bool**

Return whether this Node has a builder or not.

In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly ("if node.builder: ...”). When the builder attribute is examined directly, it ends up calling __getattr__ for both the __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.

**has_explicit_builder () → bool**

Return whether this Node has an explicit builder.

This allows an internal Builder created by SCons to be marked non-explicit, so that it can be overridden by an explicit builder that the user supplies (the canonical example being directories).

**ignore**
**ignore_set**
**implicit**
**implicit_set**
**includes**

**is_conftest () → bool**

Returns true if this node is an conftest node

**is_derived () → bool**

Returns true if this node is derived (i.e. built).

This should return true only for nodes whose path should be in the variant directory when duplicate=0 and should contribute their build signatures when they are used as source files to other derived files. For example: source with source builders are not derived in this sense, and hence should not return true.

**is_explicit**

**is_literal () → bool**

Always pass the string representation of a Node to the command interpreter literally.

**is_sconscript () → bool**

Returns true if this node is an sconscript

**is_under (dir) → bool**
is_up_to_date () → bool
    If any child is not up-to-date, then this directory isn’t, either.
isdir () → bool
isfile () → bool
islink () → bool
link (srcdir, duplicate) → None
    Set this directory as the variant directory for the supplied source directory.
linked
lstat ()
make_ready () → None
    Get a Node ready for evaluation.
    This is called before the Taskmaster decides if the Node is up-to-date or not. Overriding this method allows for a Node subclass to be disambiguated if necessary, or for an implicit source builder to be attached.
missing () → bool
multiple_side_effect_has_builder ()
    Return whether this Node has a builder or not.
    In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly (“if node.builder: …”). When the builder attribute is examined directly, it ends up calling __getattr__ for both the __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.
must_be_same (klass) → None
    This node, which already existed, is being looked up as the specified klass. Raise an exception if it isn’t.
name
new_binfo ()
new_ninfo ()
ninfo
nocache
noclean
on_disk_entries
path
postprocess () → None
    Clean up anything we don’t need to hang onto after we’ve been built.
precious
prepare () → None
    Prepare for this Node to be built.
    This is called after the Taskmaster has decided that the Node is out-of-date and must be rebuilt, but before actually calling the method to build the Node.
    This default implementation checks that explicit or implicit dependencies either exist or are derived, and initializes the BuildInfo structure that will hold the information about how this node is, uh, built.
    (The existence of source files is checked separately by the Executor, which aggregates checks for all of the targets built by a specific action.)
    Overriding this method allows for for a Node subclass to remove the underlying file from the file system. Note that subclass methods should call this base class method to get the child check and the BuildInfo structure.
prerequisites
pseudo
push_to_cache () → None
    Try to push a node into a cache
rdir ()
ref_count
rel_path (other)
    Return a path to “other” relative to this directory.
release_target_info () → None
    Called just after this node has been marked up-to-date or was built completely.
    This is where we try to release as many target node infos as possible for clean builds and update runs, in order to minimize the overall memory consumption.
By purging attributes that aren’t needed any longer after a Node (=File) got built, we don’t have to care that much how many KBytes a Node actually requires…as long as we free the memory shortly afterwards.

@see: built() and File.release_target_info()

remove()
Remove this Node: no-op by default.

render_include_tree()
Return a text representation, suitable for displaying to the user, of the include tree for the sources of this node.

rentry()

rentry_exists_on_disk(name)
Searches through the file/dir entries of the current and all its remote directories (repos), and returns True if a physical entry with the given name could be found. The local directory (self) gets searched first, so repositories take a lower precedence regarding the searching order.

@see entry_exists_on_disk

repositories

reset_executor() \rightarrow None
Remove cached executor; forces recompute when needed.

retrieve_from_cache() \rightarrow bool
Try to retrieve the node’s content from a cache
This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().
Returns true if the node was successfully retrieved.

rexists()
Does this node exist locally or in a repository?

rfile()

root

rstr() \rightarrow str
A Node.FS.Base object’s string representation is its path name.

sbuilder

scan() \rightarrow None
Scan this node’s dependents for implicit dependencies.

scanner_key()
A directory does not get scanned.

scanner_paths

sconsign()
Return the .sconsign file info for this directory.

searched

select_scanner(scanner)
Selects a scanner for this Node.
This is a separate method so it can be overridden by Node subclasses (specifically, Node.FS.Dir) that must use their own Scanner and don’t select one the Scanner.Selector that’s configured for the target.

set_always_build(always_build: int = 1) \rightarrow None
Set the Node’s always_build value.

set_executor(executor: Executor) \rightarrow None
Set the action executor for this node.

set_explicit(is_explicit) \rightarrow None

set_local() \rightarrow None

set_nocache(nocache: int = 1) \rightarrow None
Set the Node’s nocache value.

set_nocheck(noclean: int = 1) \rightarrow None
Set the Node’s noclean value.

set_precious(precious: int = 1) \rightarrow None
Set the Node’s precious value.

set_pseudo(pseudo: bool = True) \rightarrow None
Set the Node’s pseudo value.

set_specific_source(source) \rightarrow None
SCons.compat package

set_src_builder (builder) → None
    Set the source code builder for this node.
set_state (state) → None
side_effect
side_effects
sources
sources_set
src_builder ()
    Fetch the source code builder for this node.
    If there isn’t one, we cache the source code builder specified for the directory (which in turn will cache the value
    from its parent directory, and so on up to the file system root).
srcdir
srcdir_duplicate (name)
srcdir_find_file (filename)
srcdir_list ()
srcnode ()
    Dir has a special need for srcnode()...if we have a srcdir attribute set, then that is our srcnode.
stat ()
state
store_info
str_for_display ()
target_from_source (prefix, suffix, splitext=<function splitext>)
    Generates a target entry that corresponds to this entry (usually a source file) with the specified prefix and suffix.
    Note that this method can be overridden dynamically for generated files that need different behavior. See
    Tool/swig.py for an example.
target_peers
up ()
variant_dirs
visited () → None
    Called just after this node has been visited (with or without a build).
waiting_parents
waiting_s_e
walk (func, arg) → None
    Walk this directory tree by calling the specified function for each directory in the tree.
    This behaves like the os.path.walk() function, but for in-memory Node.FS.Dir objects. The function takes the same
    arguments as the functions passed to os.path.walk():
        func(arg, dirname, fnames)
    Except that “dirname” will actually be the directory Node, not the string. The ‘.’ and ‘..’ entries are excluded from
    fnames. The fnames list may be modified in-place to filter the subdirectories visited or otherwise impose a specific
    order. The “arg” argument is always passed to func() and may be used in any way (or ignored, passing None is
    common).
wkids
SCons.Node.FS.UnlinkFunc (target, source, env) → int
class SCons.Node.FS._Null
    Bases: object
SCons.Node.FS._classEntry
    alias of Entry
SCons.Node.FS._copy_func (fs, src, dest) → None
SCons.Node.FS._hardlink_func (fs, src, dst) → None
SCons.Node.FS._my_normcase (x)
SCons.Node.FS._my_splitdrive (p)
SCons.Node.FS._softlink_func (fs, src, dst) → None
SCons.Node.FS.diskcheck_types ()
SCons.Node.FS.do_diskcheck_match (node, predicate, errorfmt)
SCons.Node.FS.find_file (filename, paths, verbose=None)
    Find a node corresponding to either a derived file or a file that exists already.
Only the first file found is returned, and none is returned if no file is found.
filename: A filename to find paths: A list of directory path nodes to search in. Can be represented as a list, a tuple, or a callable that is called with no arguments and returns the list or tuple.
returns The node created from the found file.
SCons.Node.FS.get_MkdirBuilder ()
SCons.Node.FS.get_default_fs ()
SCons.Node.FS.has_glob_magic (s) → bool
SCons.Node.FS.ignore_diskcheck_match (node, predicate, errorfmt) → None
SCons.Node.FS.initialize_do_splitdrive () → None
SCons.Node.FS.invalidate_node_memos (targets) → None
   Invalidate the memoized values of all Nodes (files or directories) that are associated with the given entries. Has been added to clear the cache of nodes affected by a direct execution of an action (e.g. Delete/Copy/Chmod). Existing Node caches become inconsistent if the action is run through Execute(). The argument targets can be a single Node object or filename, or a sequence of Nodes/filenames.
SCons.Node.FS.needs_normpath_match (string, pos=0, endpos=9223372036854775807)
   Matches zero or more characters at the beginning of the string.
SCons.Node.FS.save_strings (val) → None
SCons.Node.FS.sconsign_dir (node)
   Return the .sconsign file info for this directory, creating it first if necessary.
SCons.Node.FS.sconsign_none (node)
SCons.Node.FS.set_diskcheck (enabled_checkers) → None
SCons.Node.FS.set_duplicate (duplicate)

SCons.Node.FS needs normpath_match (string, pos=0, endpos=9223372036854775807)

SCons.Node Python module

Python nodes.
class SCons.Node.Python.Value (value, built_value=None, name=None)
   Bases: Node
   A Node class for values represented by Python expressions.
   Values are typically passed on the command line or generated by a script, but not from a file or some other source.
   Changed in version 4.0: the name parameter was added.
class Attrs
   Bases: object
   shared
BuildInfo
   alias of ValueBuildInfo
Decider (function) → None
GetTag (key)
   Return a user-defined tag.
NodelInfo
   alias of ValueNodelInfo
Tag (key, value) → None
   Add a user-defined tag.
   _add_child (collection, set, child) → None
      Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.
   _children_get ()
   _children_reset () → None
   _func_exists
   _func_get_contents
   _func_is_derived
   _func_rexists
   _func_target_from_source
   _get_scanner (env, initial_scanner, root_node_scanner, kw)
   _memo
   _specific_sources
   _tags

241
SCons.compat package

add_dependency (depend)
    Adds dependencies.
add_ignore (depend)
    Adds dependencies to ignore.
add_prerequisite (prerequisite) → None
    Adds prerequisites
add_source (source)
    Adds sources.
add_to_implicit (deps) → None
add_to_waiting_parents (node) → int
    Returns the number of nodes added to our waiting parents list: 1 if we add a unique waiting parent, 0 if not. (Note that the returned values are intended to be used to increment a reference count, so don’t think you can “clean up” this function by using True and False instead…)
add_to_waiting_s_e (node) → None
add_wkid (wkid) → None
    Add a node to the list of kids waiting to be evaluated
all_children (scan: int = 1)
    Return a list of all the node’s direct children.
alter_targets ()
    Return a list of alternate targets for this Node.
always_build
attributes
bininfo
build (**kw) → None
    Actually build the node.
    This is called by the Taskmaster after it’s decided that the Node is out-of-date and must be rebuilt, and after the prepare() method has gotten everything, uh, prepared.
    This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().
builder
builder_set (builder) → None
built () → None
    Called just after this node is successfully built.
cached
changed (node=None, allowcache: bool = False)
    Returns if the node is up-to-date with respect to the BuildInfo stored last time it was built. The default behavior is to compare it against our own previously stored BuildInfo, but the stored BuildInfo from another Node (typically one in a Repository) can be used instead.
    Note that we now always check every dependency. We used to short-circuit the check by returning as soon as we detected any difference, but we now rely on checking every dependency to make sure that any necessary Node information (for example, the content signature of an #included .h file) is updated.
    The allowcache option was added for supporting the early release of the executor/builder structures, right after a File target was built. When set to true, the return value of this changed method gets cached for File nodes. Like this, the executor isn’t needed any longer for subsequent calls to changed().
@see: FS.File.changed(), FS.File.release_target_info()
changed_since_last_build
check_attributes (name)
    Simple API to check if the node.attributes for name has been set
children (scan: int = 1)
    Return a list of the node’s direct children, minus those that are ignored by this node.
children_are_up_to_date () → bool
    Alternate check for whether the Node is current: If all of our children were up-to-date, then this Node was up-to-date, too.
clear () → None

242
Completely clear a Node of all its cached state (so that it can be re-evaluated by interfaces that do continuous integration builds).

- `clear_memoized_values () → None`
- `del_binfo () → None`
  
  Delete the build info from this node.
- `depends`
- `depends_set`
- `disambiguate (must_exist=None)`
- `env`
  
  `env_set (env, safe: bool = False) → None`
- `executor`
  
  `executor_cleanup () → None`
  
  Let the executor clean up any cached information.
- `exists () → bool`
  
  Reports whether node exists.
- `explain ()`
- `for_signature ()`
  
  Return a string representation of the Node that will always be the same for this particular Node, no matter what. This is by contrast to the `__str__() method, which might, for instance, return a relative path for a file Node. The purpose of this method is to generate a value to be used in signature calculation for the command line used to build a target, and we use this method instead of `str()` to avoid unnecessary rebuilds. This method does not need to return something that would actually work in a command line; it can return any kind of nonsense, so long as it does not change.

- `get_abspath ()`
  
  Return an absolute path to the Node. This will return simply `str(Node)` by default, but for Node types that have a concept of relative path, this might return something different.
- `get_binfo ()`
  
  Fetch a node’s build information.
  
  `node - the node whose sources will be collected cache - alternate node to use for the signature cache returns - the build signature`
  
  This no longer handles the recursive descent of the node’s children’s signatures. We expect that they’re already built and updated by someone else, if that’s what’s wanted.
- `get_build_env ()`
  
  Fetch the appropriate Environment to build this node.
- `get_build_scanner_path (scanner)`
  
  Fetch the appropriate scanner path for this node.
- `get_builder (default_builder=None)`
  
  Return the set builder, or a specified default value
- `get_cachedir_csig ()`
- `get_contents () → bytes`
  
  Get contents for signature calculations.
- `get_csig (calc=None)`
  
  Because we’re a Python value node and don’t have a real timestamp, we get to ignore the calculator and just use the value contents.
  
  Returns string. Ideally string of hex digits. (Not bytes)
- `get_env ()`
- `get_env_scanner (env, kw={})`
- `get_executor (create: int = 1) → Executor`
  
  Fetch the action executor for this node. Create one if there isn’t already one, and requested to do so.
- `get_found_includes (env, scanner, path)`
  
  Return the scanned include lines (implicit dependencies) found in this node.
  
  The default is no implicit dependencies. We expect this method to be overridden by any subclass that can be scanned for implicit dependencies.
- `get_implicit_deps (env, initial_scanner, path_func, kw={})`
  
  Return a list of implicit dependencies for this node.
This method exists to handle recursive invocation of the scanner on the implicit dependencies returned by the scanner, if the scanner's recursive flag says that we should.

get_ninfo ()
get_source_scanner (node)
    Fetch the source scanner for the specified node
    NOTE: "self" is the target being built, "node" is the source file for which we want to fetch the scanner.
    Implies self.has_builder() is true; again, expect to only be called from locations where this is already verified.
    This function may be called very often; it attempts to cache the scanner found to improve performance.
get_state ()
get_stored_implicit ()
get_stored_info ()
get_string (for_signature)
    This is a convenience function designed primarily to be used in command generators (i.e., CommandGeneratorActions or Environment variables that are callable), which are called with a for_signature argument that is nonzero if the command generator is being called to generate a signature for the command line, which determines if we should rebuild or not.
    Such command generators should use this method in preference to str(Node) when converting a Node to a string, passing in the for_signature parameter, such that we will call Node.for_signature() or str(Node) properly, depending on whether we are calculating a signature or actually constructing a command line.
get_subst_proxy ()
    This method is expected to return an object that will function exactly like this Node, except that it implements any additional special features that we would like to be in effect for Environment variable substitution. The principle use is that some Nodes would like to implement a __getattr__() method, but putting that in the Node type itself has a tendency to kill performance. We instead put it in a proxy and return it from this method. It is legal for this method to return self if no new functionality is needed for Environment substitution.
get_suffix () → str
get_target_scanner ()
get_text_contents () → str
    By the assumption that the node.built_value is a deterministic product of the sources, the contents of a Value are the concatenation of all the contents of its sources. As the value need not be built when get_contents() is called, we cannot use the actual node.built_value.

has_builder () → bool
    Return whether this Node has a builder or not.
    In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly ("if node.builder: …"). When the builder attribute is examined directly, it ends up calling __getattr__ for both the __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.

has_explicit_builder () → bool
    Return whether this Node has an explicit builder.
    This allows an internal Builder created by SCons to be marked non-explicit, so that it can be overridden by an explicit builder that the user supplies (the canonical example being directories).

ignore
ignore_set
implicit
implicit_set
includes
is_conftest () → bool
    Returns true if this node is an conftest node
is-derived () → bool
    Returns true if this node is derived (i.e. built).
    This should return true only for nodes whose path should be in the variant directory when duplicate=0 and should contribute their build signatures when they are used as source files to other derived files. For example: source with source builders are not derived in this sense, and hence should not return true.

is_explicit
is_literal () → bool
Always pass the string representation of a Node to the command interpreter literally.

is_sconscript () → bool
Returns true if this node is an sconscript

is_under (dir) → bool

is_up_to_date () → bool
Alternate check for whether the Node is current: If all of our children were up-to-date, then this Node was up-to-date, too.


linked

make_ready () → None
Get a Node ready for evaluation.

This is called before the Taskmaster decides if the Node is up-to-date or not. Overriding this method allows for a Node subclass to be disambiguated if necessary, or for an implicit source builder to be attached.

missing () → bool

multiple_side_effect_has_builder () → bool
Return whether this Node has a builder or not.

In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly ("if node.builder: ..."). When the builder attribute is examined directly, it ends up calling __getattr__ for both the __len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.

new_binfo ()

new_ninfo ()
ninfo
nocache
nocrack

postprocess () → None
Clean up anything we don’t need to hang onto after we’ve been built.

precious

prepare ()
Prepare for this Node to be built.

This is called after the Taskmaster has decided that the Node is out-of-date and must be rebuilt, but before actually calling the method to build the Node.

This default implementation checks that explicit or implicit dependencies either exist or are derived, and initializes the BuildInfo structure that will hold the information about how this node is, uh, built.

(The existence of source files is checked separately by the Executor, which aggregates checks for all of the targets built by a specific action.)

Overriding this method allows for a Node subclass to remove the underlying file from the file system. Note that subclass methods should call this base class method to get the child check and the BuildInfo structure.

prerequisites

pseudo

push_to_cache () → None
Try to push a node into a cache

read ()
Return the value. If necessary, the value is built.

ref_count

release_target_info () → None
Called just after this node has been marked up-to-date or was built completely.

This is where we try to release as many target node infos as possible for clean builds and update runs, in order to minimize the overall memory consumption.

By purging attributes that aren’t needed any longer after a Node (=File) got built, we don’t have to care that much how many KBytes a Node actually requires...as long as we free the memory shortly afterwards.

@see: built() and File.release_target_info()

remove ()
Remove this Node: no-op by default.

render_include_tree ()
Return a text representation, suitable for displaying to the user, of the include tree for the sources of this node.
reset_executor () \rightarrow None
    Remove cached executor; forces recompute when needed.
retrieve_from_cache () \rightarrow bool
    Try to retrieve the node’s content from a cache
    This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().
    Returns true if the node was successfully retrieved.
rexists ()
    Does this node exist locally or in a repository?
scan () \rightarrow None
    Scan this node’s dependents for implicit dependencies.
scanner_key ()
select_scanner (scanner)
    Selects a scanner for this Node.
    This is a separate method so it can be overridden by Node subclasses (specifically, Node.FS.Dir) that must use their own Scanner and don’t select one the Scanner.Selector that’s configured for the target.
set_always_build (always_build: int = 1) \rightarrow None
    Set the Node’s always_build value.
set_executor (executor: Executor) \rightarrow None
    Set the action executor for this node.
set_explicit (is_explicit) \rightarrow None
set_nocache (nocache: int = 1) \rightarrow None
    Set the Node’s nocache value.
set_no_clean (noclean: int = 1) \rightarrow None
    Set the Node’s noclean value.
set_precious (precious: int = 1) \rightarrow None
    Set the Node’s precious value.
set_pseudo (pseudo: bool = True) \rightarrow None
    Set the Node’s pseudo value.
set_specific_source (source) \rightarrow None
set_state (state) \rightarrow None
side_effect
side_effects
sources
sources_set
state
store_info
str_for_display ()
target_peers
visited () \rightarrow None
    Called just after this node has been visited (with or without a build).
waiting_parents
waiting_s_e
wkids
write (built_value) \rightarrow None
    Set the value of the node.

class SCons.Node.Python.ValueBuildInfo
Bases: BuildInfoBase
    Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a 'dict' slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all instances of a class.
_setstate_ (state) \rightarrow None
    Restore the attributes from a pickled state.
SCons.compat package

bdepends
bdependsigns
bimplicit
bimplicitsigns
bsources
bsourcestags
current_version_id = 2
merge (other) → None
  Merge the fields of another object into this object. Already existing information is overwritten by the other instance’s data. WARNING: If a '__dict__' slot is added, it should be updated instead of replaced.
  Bases: NodeInfoBase
  __getstate__()  
    Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a '__dict__' slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all instances of a class.
  __setstate__(state) → None
    Restore the attributes from a pickled state.
  convert (node, val) → None
  csig
  current_version_id = 2
  field_list = ['csig']
  format (field_list=None, names: int = 0)
  merge (other) → None
    Merge the fields of another object into this object. Already existing information is overwritten by the other instance’s data. WARNING: If a '__dict__' slot is added, it should be updated instead of replaced.
  str_to_node (s)
  update (node) → None
SCons.Node.Python.ValueWithMemo (value, built_value=None, name=None)
  Memoized Value node factory.
  Changed in version 4.0: the name parameter was added.

Module contents

The Node package for the SCons software construction utility.

This is, in many ways, the heart of SCons.

A Node is where we encapsulate all of the dependency information about any thing that SCons can build, or about any thing which SCons can use to build some other thing. The canonical “thing,” of course, is a file, but a Node can also represent something remote (like a web page) or something completely abstract (like an Alias).

Each specific type of “thing” is specifically represented by a subclass of the Node base class: Node.FS.File for files, Node.Alias for aliases, etc. Dependency information is kept here in the base class, and information specific to files/aliases/etc. is in the subclass. The goal, if we’ve done this correctly, is that any type of “thing” should be able to depend on any other type of “thing.”

SCons.Node.Annotate (node) → None

class SCons.Node.BuildInfoBase
  Bases: object
  The generic base class for build information for a Node.
  This is what gets stored in a .sconsign file for each target file. It contains a NodelInfo instance for this node (signature information that’s specific to the type of Node) and direct attributes for the generic build stuff we have to track: sources, explicit dependencies, implicit dependencies, and action information.
  __getstate__()  
    Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a '__dict__' slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all instances of a class.
  __setstate__(state) → None

247
Restore the attributes from a pickled state.

```
bact
bactsig
bdepends
bdependsigs
bimplicit
bimplicitigs
bsources
bsourcesigs
```

current_version_id = 2

merge (other) → None

Merge the fields of another object into this object. Already existing information is overwritten by the other instance’s data. WARNING: If a `__dict__` slot is added, it should be updated instead of replaced.

class SCons.Node.Node

Bases: object

The base Node class, for entities that we know how to build, or use to build other Nodes.

class Attrs

Bases: object

shared

BuildInfo

alias of BuildInfoBase

Decider (function) → None

GetTag (key)

Return a user-defined tag.

NodeInfo

alias of NodeInfoBase

Tag (key, value) → None

Add a user-defined tag.

```
_add_child (collection, set, child) → None
Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.
```

_children_get ()

_children_reset () → None

_func_exists

_func_get_contents

_func_is_derived

_func_rexists

_func_target_from_source

_get_scanner (env, initial_scanner, root_node_scanner, kw)

_memo

_specific_sources

tags

add_dependency (depend)

Adds dependencies.

add_ignore (depend)

Adds dependencies to ignore.

add_prerequisite (prerequisite) → None

Adds prerequisites

add_source (source)

Adds sources.

add_to_implicit (deps) → None

add_to_waiting_parents (node) → int

Returns the number of nodes added to our waiting parents list: 1 if we add a unique waiting parent, 0 if not. (Note that the returned values are intended to be used to increment a reference count, so don’t think you can “clean up” this function by using True and False instead…)

add_to_waiting_s_e (node) → None

add_wkid (wkid) → None
Add a node to the list of kids waiting to be evaluated
all_children (scan: int = 1)
    Return a list of all the node’s direct children.
alter_targets ()
    Return a list of alternate targets for this Node.
always_build
attributes
binfo
build (**kw)
    Actually build the node.
    This is called by the Taskmaster after it’s decided that the Node is out-of-date and must be rebuilt, and after the
    prepare() method has gotten everything, uh, prepared.
    This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe
    stuff in built().
builder
builder_set (builder) → None
built () → None
    Called just after this node is successfully built.
cached
changed (node=None, allowcache: bool = False)
    Returns if the node is up-to-date with respect to the BuildInfo stored last time it was built. The default behavior is to
    compare it against our own previously stored BuildInfo, but the stored BuildInfo from another Node (typically one in
    a Repository) can be used instead.
    Note that we now always check every dependency. We used to short-circuit the check by returning as soon as we
    detected any difference, but we now rely on checking every dependency to make sure that any necessary Node
    information (for example, the content signature of an #included .h file) is updated.
    The allowcache option was added for supporting the early release of the executor/builder structures, right after a
    File target was built. When set to true, the return value of this changed method gets cached for File nodes. Like
    this, the executor isn’t needed any longer for subsequent calls to changed().
@see: FS.File.changed(), FS.File.release_target_info()
changed_since_last_build
check_attributes (name)
    Simple API to check if the node.attributes for name has been set
children (scan: int = 1)
    Return a list of the node’s direct children, minus those that are ignored by this node.
children_are_up_to_date () → bool
    Alternate check for whether the Node is current: If all of our children were up-to-date, then this Node was
    up-to-date, too.
clear () → None
    Completely clear a Node of all its cached state (so that it can be re-evaluated by interfaces that do continuous
    integration builds).
clear_memoized_values () → None
del_binfo () → None
    Delete the build info from this node.
depends
depends_set
disambiguate (must_exist=None)
env
    env_set (env, safe: bool = False) → None
executor
executor_cleanup () → None
    Let the executor clean up any cached information.
exists () → bool
    Reports whether node exists.
explain ()
for_signature()
return a string representation of the Node that will always be the same for this particular Node, no matter what.
This is by contrast to the __str__() method, which might, for instance, return a relative path for a file Node. The
purpose of this method is to generate a value to be used in signature calculation for the command line used to
build a target, and we use this method instead of str() to avoid unnecessary rebuilds. This method does not need to
return something that would actually work in a command line; it can return any kind of nonsense, so long as it does
not change.
get_abspath()
return an absolute path to the Node. This will return simply str(Node) by default, but for Node types that have a
concept of relative path, this might return something different.
get_binfo()
fetch a node’s build information.
node - the node whose sources will be collected cache - alternate node to use for the signature cache returns - the
build signature
This no longer handles the recursive descent of the node’s children’s signatures. We expect that they’re already
built and updated by someone else, if that’s what’s wanted.
get_build_env()
fetch the appropriate Environment to build this node.
get_build_scanner_path(scanner)
fetch the appropriate scanner path for this node.
get_builder(default_builder=None)
return the set builder, or a specified default value
get_cachedir_csig()
get_contents()
fetch the contents of the entry.
get_csig()
get_env()
get_env_scanner(env, kw={})
get_executor(create: int = 1) → Executor
fetch the action executor for this node. Create one if there isn’t already one, and requested to do so.
get_found_includes(env, scanner, path)
return the scanned include lines (implicit dependencies) found in this node.
The default is no implicit dependencies. We expect this method to be overridden by any subclass that can be
scanned for implicit dependencies.
get_implicit_deps(env, initial_scanner, path_func, kw={})
return a list of implicit dependencies for this node.
This method exists to handle recursive invocation of the scanner on the implicit dependencies returned by the
scanner, if the scanner’s recursive flag says that we should.
get_ninfo()
get_source_scanner(node)
fetch the source scanner for the specified node
NOTE: “self” is the target being built, “node” is the source file for which we want to fetch the scanner.
implies self.has_builder() is true; again, expect to only be called from locations where this is already verified.
This function may be called very often; it attempts to cache the scanner found to improve performance.
get_state()
get_stored_implicit()
fetch the stored implicit dependencies
get_stored_info()
get_string(for_signature)
this is a convenience function designed primarily to be used in command generators (i.e.,
CommandGeneratorActions or Environment variables that are callable), which are called with a for_signature
argument that is nonzero if the command generator is being called to generate a signature for the command line,
which determines if we should rebuild or not.
such command generators should use this method in preference to str(Node) when converting a Node to a string,
passing in the for_signature parameter, such that we will call Node.for_signature() or str(Node) properly,
depending on whether we are calculating a signature or actually constructing a command line.
get_subst_proxy ()
    This method is expected to return an object that will function exactly like this Node, except that it implements any
    additional special features that we would like to be in effect for Environment variable substitution. The principle use
    is that some Nodes would like to implement a __getattr__() method, but putting that in the Node type itself has a
tendency to kill performance. We instead put it in a proxy and return it from this method. It is legal for this method to
return self if no new functionality is needed for Environment substitution.

get_suffix () → str

get_target_scanner ()

has_builder () → bool
    Return whether this Node has a builder or not.
    In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly ("if
node.builder: ..."). When the builder attribute is examined directly, it ends up calling __getattr__ for both the
__len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and
slowing things down immensely.

has_explicit_builder () → bool
    Return whether this Node has an explicit builder.
    This allows an internal Builder created by SCons to be marked non-explicit, so that it can be overridden by an
explicit builder that the user supplies (the canonical example being directories).

ignore
ignore_set
implicit
implicit_set
includes

is_conftest () → bool
    Returns true if this node is an conftest node

is_derived () → bool
    Returns true if this node is derived (i.e. built).
    This should return true only for nodes whose path should be in the variant directory when duplicate=0 and should
contribute their build signatures when they are used as source files to other derived files. For example: source with
source builders are not derived in this sense, and hence should not return true.

is_explicit
is_literal () → bool
    Always pass the string representation of a Node to the command interpreter literally.

is_sconscript () → bool
    Returns true if this node is an sconscript

is_up_to_date () → bool
    Default check for whether the Node is current: unknown Node subtypes are always out of date, so they will always
get built.

linked

make_ready () → None
    Get a Node ready for evaluation.
    This is called before the Taskmaster decides if the Node is up-to-date or not. Overriding this method allows for a
Node subclass to be disambiguated if necessary, or for an implicit source builder to be attached.

missing () → bool

multiple_side_effect_has_builder () → bool
    Return whether this Node has a builder or not.
    In Boolean tests, this turns out to be a lot more efficient than simply examining the builder attribute directly ("if
node.builder: ..."). When the builder attribute is examined directly, it ends up calling __getattr__ for both the
__len__ and __bool__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and
slowing things down immensely.

new_binfo ()
new_ninfo ()
ninfo
nocache
noclean

postprocess () → None
Clean up anything we don’t need to hang onto after we’ve been built.

precious
prepare()

Prepare for this Node to be built.

This is called after the Taskmaster has decided that the Node is out-of-date and must be rebuilt, but before actually calling the method to build the Node.

This default implementation checks that explicit or implicit dependencies either exist or are derived, and initializes the BuildInfo structure that will hold the information about how this node is, uh, built.

(The existence of source files is checked separately by the Executor, which aggregates checks for all of the targets built by a specific action.)

Overriding this method allows for a Node subclass to remove the underlying file from the file system. Note that subclass methods should call this base class method to get the child check and the BuildInfo structure.

prerequisites
pseudo
push_to_cache() → None
Try to push a node into a cache
ref_count
release_target_info() → None
Called just after this node has been marked up-to-date or was built completely.

This is where we try to release as many target node infos as possible for clean builds and update runs, in order to minimize the overall memory consumption.

By purging attributes that aren’t needed any longer after a Node (=File) got built, we don’t have to care that much how many KBytes a Node actually requires…as long as we free the memory shortly afterwards.

@see: built() and File.release_target_info()
remove()
Remove this Node: no-op by default.
render_include_tree() Return a text representation, suitable for displaying to the user, of the include tree for the sources of this node.
reset_executor() → None
Remove cached executor; forces recompute when needed.
retrieve_from_cache() → bool
Try to retrieve the node’s content from a cache

This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in built().

Returns true if the node was successfully retrieved.
rexists()
Does this node exist locally or in a repository?
scan() → None
Scan this node’s dependents for implicit dependencies.
scanner_key()
select_scanner(scanner)
Selects a scanner for this Node.

This is a separate method so it can be overridden by Node subclasses (specifically, Node.FS.Dir) that must use their own Scanner and don’t select one the Scanner.Selector that’s configured for the target.

set_always_build(always_build: int = 1) → None
Set the Node’s always_build value.
set_executor(executor: Executor) → None
Set the action executor for this node.
set_explicit(is_explicit) → None
set_nocache(nocache: int = 1) → None
Set the Node’s nocache value.
set_noclean(noclean: int = 1) → None
Set the Node’s noclean value.
set_precious(precious: int = 1) → None
Set the Node’s precious value.
set_pseudo(pseudo: bool = True) → None
Set the Node’s pseudo value.

```python
set_specific_source(source) → None
set_state(state) → None
side_effect
side_effects
sources
sources_set
state
store_info
target_peers
visited() → None
```

Called just after this node has been visited (with or without a build).

waiting_parents
waiting_s_e
wkids

class SCons.Node.NodeInfoBase

Bases: object

The generic base class for signature information for a Node. Node subclasses should subclass NodeInfoBase to provide their own logic for dealing with their own Node-specific signature information.

```python
__getstate__() → None

Return all fields that shall be pickled. Walk the slots in the class hierarchy and add those to the state dictionary. If a '__dict__' slot is available, copy all entries to the dictionary. Also include the version id, which is fixed for all instances of a class.

__setstate___(state) → None

Restore the attributes from a pickled state. The version is discarded.

convert(node, val) → None

current_version_id = 2

format(field_list=None, names: int = 0) → None

merge(other) → None

Merge the fields of another object into this object. Already existing information is overwritten by the other instance’s data. WARNING: If a '__dict__' slot is added, it should be updated instead of replaced.

update(node) → None

class SCons.Node.NodeList(initlist=None)

Bases: UserList

_abc_impl = _abc._abc_data object>

append(item)

S.append(value) – append value to the end of the sequence

clear() → None -- remove all items from S

copy()

count(value) → integer -- return number of occurrences of value

extend(other)

S.extend(iterable) – extend sequence by appending elements from the iterable

index(value[, start[, stop]]) → integer -- return first index of value.

Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

insert(i, item)

S.insert(index, value) – insert value before index

pop([, index]) → item -- remove and return item at index (default last).

Raise IndexError if list is empty or index is out of range.

remove(item)

S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

reverse()

S.reverse() – reverse IN PLACE

sort(*args, **kwds)
class SCons.Node.Walker(node, kids_func=<function get_children>, cycle_func=<function ignore_cycle>, eval_func=<function do_nothing>)

Bases: object

An iterator for walking a Node tree.
This is depth-first, children are visited before the parent. The Walker object can be initialized with any node, and returns the next node on the descent with each get_next() call. get the children of a node instead of calling ‘children’.

cycle_func is an optional function that will be called when a cycle is detected.

This class does not get caught in node cycles caused, for example, by C header file include loops.

get_next()

Return the next node for this walk of the tree.

This function is intentionally iterative, not recursive, to sidestep any issues of stack size limitations.

is_done() → bool

SCons.Node.changed_since_last_build_alias (node, target, prev_ni, repo_node=None) → bool
SCons.Node.changed_since_last_build_entry (node, target, prev_ni, repo_node=None) → bool
SCons.Node.changed_since_last_build_node (node, target, prev_ni, repo_node=None) → bool

Must be overridden in a specific subclass to return True if this Node (a dependency) has changed since the last time it was used to build the specified target. prev_ni is this Node’s state (for example, its file timestamp, length, maybe content signature) as of the last time the target was built.

Note that this method is called through the dependency, not the target, because a dependency Node must be able to use its own logic to decide if it changed. For example, File Nodes need to obey if we’re configured to use timestamps, but Python Value Nodes never use timestamps and always use the content. If this method were called through the target, then each Node’s implementation of this method would have to have more complicated logic to handle all the different Node types on which it might depend.

SCons.Node.changed_since_last_build_python (node, target, prev_ni, repo_node=None) → bool
SCons.Node.changed_since_last_build_state_changed (node, target, prev_ni, repo_node=None) → bool

SCons.Node.classname (obj) → str
SCons.Node.decide_source (node, target, prev_ni, repo_node=None) → bool
SCons.Node.decide_target (node, target, prev_ni, repo_node=None) → bool
SCons.Node.do_nothing (node, parent) → None
SCons.Node.do_nothing_node (node) → None
SCons.Node.exists_always (node) → bool
SCons.Node.exists_base (node) → bool
SCons.Node.exists_entry (node) → bool

Return if the Entry exists. Check the file system to see what we should turn into first. Assume a file if there’s no directory.

SCons.Node.exists_file (node) → bool
SCons.Node.exists_none (node) → bool
SCons.Node.get_children (node, parent)
SCons.Node.get_contents_dir (node)

Return content signatures and names of all our children separated by new-lines. Ensure that the nodes are sorted.

SCons.Node.get_contents_entry (node)

Fetch the contents of the entry. Returns the exact binary contents of the file.

SCons.Node.get_contents_file (node)
SCons.Node.get_contents_none (node)
SCons.Node.ignore_cycle (node, stack) → None
SCons.Node.is_derived_node (node) → bool

Returns true if this node is derived (i.e. built).

SCons.Node.is_derived_none (node)
SCons.Node.reexists_base (node)
SCons.Node.reexists_node (node)
SCons.Node.reexists_none (node)
SCons.Node.store_info_file (node) → None
SCons.Node.store_info_pass (node) → None
SCons.Node.target_from_source_base (node, prefix, suffix, splitext)
SCons.Node.target_from_source_none (node, prefix, suffix, splitext)
SCons.Platform package

Submodules

SCons.Platform.aix module
Platform-specific initialization for IBM AIX systems.
There normally shouldn’t be any need to import this module directly. It will usually be imported through the generic SCons.Platform.Platform() selection method.
SCons.Platform.aix.generate (env) → None
SCons.Platform.aix.get_xlc (env, xlc=None, packages=[]) → None

SCons.Platform.cygwin module
Platform-specific initialization for Cygwin systems.
There normally shouldn’t be any need to import this module directly. It will usually be imported through the generic SCons.Platform.Platform() selection method.
SCons.Platform.cygwin.generate (env) → None

SCons.Platform.darwin module
Platform-specific initialization for Mac OS X systems.
There normally shouldn’t be any need to import this module directly. It will usually be imported through the generic SCons.Platform.Platform() selection method.
SCons.Platform.darwin.generate (env) → None

SCons.Platform.hpux module
Platform-specific initialization for HP-UX systems.
There normally shouldn’t be any need to import this module directly. It will usually be imported through the generic SCons.Platform.Platform() selection method.
SCons.Platform.hpux.generate (env) → None

SCons.Platform.irix module
Platform-specific initialization for SGI IRIX systems.
There normally shouldn’t be any need to import this module directly. It will usually be imported through the generic SCons.Platform.Platform() selection method.
SCons.Platform.irix.generate (env) → None

SCons.Platform.mingw module
Platform-specific initialization for the MinGW system.

SCons.Platform.os2 module
Platform-specific initialization for OS/2 systems.
There normally shouldn’t be any need to import this module directly. It will usually be imported through the generic SCons.Platform.Platform() selection method.
SCons.Platform.os2.generate (env) → None

SCons.Platform.posix module
Platform-specific initialization for POSIX (Linux, UNIX, etc.) systems.
SCons.Platform package

There normally shouldn’t be any need to import this module directly. It will usually be imported through the generic SCons.Platform.Platform() selection method.
SCons.Platform.posix.escape (arg)
  escape shell special characters
SCons.Platform.posix.exec_popen3 (l, env, stdout, stderr)
SCons.Platform.posix.exec_subprocess (l, env)
SCons.Platform.posix.generate (env) → None
SCons.Platform.posix.piped_env_spawn (sh, escape, cmd, args, env, stdout, stderr)
SCons.Platform.posix.subprocess_spawn (sh, escape, cmd, args, env)

SCons.Platform.sunos module
Platform-specific initialization for Sun systems.

There normally shouldn’t be any need to import this module directly. It will usually be imported through the generic SCons.Platform.Platform() selection method.
SCons.Platform.sunos.generate (env) → None

SCons.Platform.virtualenv module

"Platform" support for a Python virtualenv.
SCons.Platform.virtualenv.ImportVirtualenv (env) → None
  Copies virtualenv-related environment variables from OS environment to env['ENV'] and prepends virtualenv's PATH to env['ENV']['PATH'].
SCons.Platform.virtualenv.IsInVirtualenv (path)
  Returns True, if path is under virtualenv's home directory. If not, or if we don't use virtualenv, returns False.
SCons.Platform.virtualenv._enable_virtualenv_default ()
SCons.Platform.virtualenv._ignore_virtualenv_default ()
SCons.Platform.virtualenv._inject_venv_path (env, path_list=None) → None
  Modify environment such that SCons will take into account its virtualenv when running external tools.
SCons.Platform.virtualenv._inject_venv_variables (env) → None
SCons.Platform.virtualenv._is_path_in (path, base) → bool
  Returns true if path is located under the base directory.
SCons.Platform.virtualenv._running_in_virtualenv ()
  Returns True if scons is executed within a virtualenv
SCons.Platform.virtualenv.select_paths_in_venv (path_list)
  Returns a list of paths from path_list which are under virtualenv's home directory.

SCons.Platform.win32 module
Platform-specific initialization for Win32 systems.

There normally shouldn’t be any need to import this module directly. It will usually be imported through the generic SCons.Platform.Platform() selection method.
class SCons.Platform.win32.ArchDefinition (arch, synonyms=[])
  Bases: object
  Determine which windows CPU were running on. A class for defining architecture-specific settings and logic.
SCons.Platform.win32.escape (x)
SCons.Platform.win32.exec_spawn (l, env)
SCons.Platform.win32.generate (env)
SCons.Platform.win32.get_architecture (arch=None)
  Returns the definition for the specified architecture string.
  If no string is specified, the system default is returned (as defined by the registry PROCESSOR_ARCHITECTURE value, PROCESSOR_ARCHITEW6432 environment variable, PROCESSOR_ARCHITECTURE environment variable, or the platform machine).
SCons.Platform.win32.get_program_files_dir ()
SCons.Platform package

Get the location of the program files directory
SCons.Platform.win32.get_system_root()
SCons.Platform.win32.piped_spawn (sh, escape, cmd, args, env, stdout, stderr)
SCons.Platform.win32.spawn (sh, escape, cmd, args, env)
SCons.Platform.win32.spawnve (mode, file, args, env)

Module contents
SCons platform selection.

Looks for modules that define a callable object that can modify a construction environment as appropriate for a given platform.

Note that we take a more simplistic view of “platform” than Python does. We’re looking for a single string that determines a set of tool-independent variables with which to initialize a construction environment. Consequently, we’ll examine both sys.platform and os.name (and anything else that might come in to play) in order to return some specification which is unique enough for our purposes.

Note that because this subsystem just selects a callable that can modify a construction environment, it’s possible for people to define their own “platform specification” in an arbitrary callable function. No one needs to use or tie in to this subsystem in order to roll their own platform definition.

SCons.Platform.DefaultToolList (platform, env)
Select a default tool list for the specified platform.
SCons.Platform.Platform (name='darwin')
Select a canned Platform specification.
class SCons.Platform.PlatformSpec (name, generate)
  Bases: object
class SCons.Platform.TempFileMunge (cmd, cmdstr=None)
  Bases: object
Convert long command lines to use a temporary file.
You can set an Environment variable (usually TEMPFILE) to this, then call it with a string argument, and it will perform temporary file substitution on it. This is used to circumvent limitations on the length of command lines. Example:

```python
env['TEMPFILE'] = TempFileMunge
env['LINKCOM'] = '${TEMPFILE(''$LINK $TARGET $SOURCES','$LINKCOMSTR')}'
```

By default, the name of the temporary file used begins with a prefix of ‘@’. This may be configured for other tool chains by setting the TEMPFILEPREFIX variable. Example:

```python
env['TEMPFILEPREFIX'] = '-@'  # diab compiler
env['TEMPFILEPREFIX'] = '-via'  # arm tool chain
env['TEMPFILEPREFIX'] = ''  # (the empty string) PC Lint
```

You can configure the extension of the temporary file through the TEMPFILESUFFIX variable, which defaults to ‘.lnk’ (see comments in the code below). Example:

```python
env['TEMPFILESUFFIX'] = '.lnt'  # PC Lint
```

Entries in the temporary file are separated by the value of the TEMPFILEARGJOIN variable, which defaults to an OS-appropriate value.
A default argument escape function is SCons.Subst.quote_spaces. If you need to apply extra operations on a command argument before writing to a temporary file (fix Windows slashes, normalize paths, etc.), please set TEMPFILEARGESCFUNC variable to a custom function. Example:
import sys
import re
from SCons.Subst import quote_spaces

WINPATHSEP_RE = re.compile(r'\(['+'\'+'
']\)|\$')

def tempfile_arg_esc_func(arg):
    arg = quote_spaces(arg)
    if sys.platform != "win32":
        # GCC requires double Windows slashes, let's use UNIX separator
        return WINPATHSEP_RE.sub(r'[/\]', arg)
    return arg

env["TEMPFILEARGESCFUNC"] = tempfile_arg_esc_func

_print_cmd_str (target, source, env, cmdstr) → None
SCons.Platform.platform_default ()
    Return the platform string for our execution environment.
    The returned value should map to one of the SCons/Platform/*.py files. Since scons is architecture independent,
    though, we don't care about the machine architecture.
SCons.Platform.platform_module (name='darwin')
    Return the imported module for the platform.
    This looks for a module name that matches the specified argument. If the name is unspecified, we fetch the
    appropriate default for our execution environment.

SCons.Scanner package

Submodules

SCons.Scanner.C module

Dependency scanner for C/C++ code.

Two scanners are defined here: the default CScanner, and the optional CConditionalScanner, which must be explicitly
selected by calling add_scanner() for each affected suffix.
SCons.Scanner.C.CConditionalScanner ()
    Return an advanced conditional Scanner instance for scanning source files
    Interprets C/C++ Preprocessor conditional syntax (#ifdef, #if, defined, #else, #elif, etc.).
SCons.Scanner.C.CScanner ()
    Return a prototype Scanner instance for scanning source files that use the C pre-processor
class SCons.Scanner.C.SConsCPPConditionalScanner (*args, **kwargs)
    Bases: PreProcessor
    SCons-specific subclass of the cpp.py module's processing.
    We subclass this so that: 1) we can deal with files represented by Nodes, not strings; 2) we can keep track of the files
    that are missing.
__call__ (file)
    Pre-processes a file.
    This is the main public entry point.
_do_if_else_condition (condition) → None
    Common logic for evaluating the conditions on #if, #ifdef and #ifndef lines.
_match_tuples (tuples)
_parse_tuples (contents)
_process_tuples (tuples, file=None)
all_include (t) → None
  Default handling of a #include line.
do_define (t) → None
  Default handling of a #define line.
do_elif (t) → None
  Default handling of a #elif line.
do_else (t) → None
  Default handling of a #else line.
do endif (t) → None
  Default handling of a #endif line.
do_if (t) → None
  Default handling of a #if line.
do ifndef (t) → None
  Default handling of a #ifndef line.
do_import (t) → None
  Default handling of a #import line.
do_include (t) → None
  Default handling of a #include line.
do_include_next (t) → None
  Default handling of a #include line.
do_nothing (t) → None
  Null method for when we explicitly want the action for a specific preprocessor directive to do nothing.
do_UNDEF (t) → None
  Default handling of a #undef line.
eval_expression (t)
  Evaluates a C preprocessor expression.
  This is done by converting it to a Python equivalent and eval()ing it in the C preprocessor namespace we use to
  track #define values.
finalize_result (fname)
find_include_file (t)
  Finds the #include file for a given preprocessor tuple.
initialize_result (fname) → None
process_contents (contents)
  Pre-processes a file contents.
  Is used by tests
process_file (file)
  Pre-processes a file.
  This is the main internal entry point.
read_file (file) → str
resolve_include (t)
  Resolve a tuple-ized #include line.
  This handles recursive expansion of values without "" or <> surrounding the name until an initial " or < is found, to
  handle #include FILE where FILE is a #define somewhere else.
restore () → None
  Pops the previous dispatch table off the stack and makes it the current one.
save () → None
  Pushes the current dispatch table on the stack and re-initializes the current dispatch table to the default.
scons_current_file (t) → None
start_handling_includes (t=None) → None
  Causes the PreProcessor object to start processing #import, #include and #include_next lines.
  This method will be called when a #if, #ifdef, #ifndef or #elif evaluates True, or when we reach the #else in a #if,
  #ifdef, #ifndef or #elif block where a condition already evaluated False.
stop_handling_includes (t=None) → None
  Causes the PreProcessor object to stop processing #import, #include and #include_next lines.
This method will be called when a #if, #ifdef, #ifndef or #elif evaluates False, or when we reach the #else in a #if, #ifdef, #ifndef or #elif block where a condition already evaluated True.

tupleize (contents)

Turns the contents of a file into a list of easily-processed tuples describing the CPP lines in the file.

The first element of each tuple is the line’s preprocessor directive (#if, #include, #define, etc., minus the initial ‘#’).

The remaining elements are specific to the type of directive, as pulled apart by the regular expression.

class SCons.Scanner.C.SConsCPPConditionalScannerWrapper (name, variable)

Bases: object

The SCons wrapper around a cpp.py scanner.

This is the actual glue between the calling conventions of generic SCons scanners, and the (subclass of) cpp.py class that knows how to look for #include lines with reasonably real C-preprocessor-like evaluation of #if/#ifdef/#else/#elif lines.

recursively (nodes)

select (node)

class SCons.Scanner.C.SConsCPPScanner (*args, **kwargs)

Bases: PreProcessor

SCons-specific subclass of the cpp.py module’s processing.

We subclass this so that: 1) we can deal with files represented by Nodes, not strings; 2) we can keep track of the files that are missing.

__call__ (file)

Pre-processes a file.

This is the main public entry point.

__do_if else condition (condition) → None

Common logic for evaluating the conditions on #if, #ifdef and #ifndef lines.

__match_tuples (tuples)

__parse_tuples (contents)

__process_tuples (tuples, file=None)

all_include (t) → None

do_define (t) → None

default handling of a #define line.

do_elif (t) → None

default handling of a #elif line.

do_else (t) → None

default handling of a #else line.

do_endif (t) → None

default handling of a #endif line.

do_if (t) → None

default handling of a #if line.

do_ifdef (t) → None

default handling of a #ifdef line.

do ifndef (t) → None

default handling of a #ifndef line.

do_import (t) → None

default handling of a #import line.

do_include (t) → None

default handling of a #include line.

do_next (t) → None

default handling of a #include line.

do_nothing (t) → None

Null method for when we explicitly want the action for a specific preprocessor directive to do nothing.

do_undef (t) → None

default handling of a #undef line.

eval_expression (t)

Evaluates a C preprocessor expression.

This is done by converting it to a Python equivalent and eval()ing it in the C preprocessor namespace we use to track #define values.
finalize_result (fname)
find_include_file (t)
   Finds the #include file for a given preprocessor tuple.
initialize_result (fname) → None
process_contents (contents)
   Pre-processes a file contents.
   Is used by tests
process_file (file)
   Pre-processes a file.
   This is the main internal entry point.
read_file (file) → str
resolve_include (t)
   Resolve a tuple-ized #include line.
   This handles recursive expansion of values without “” or <> surrounding the name until an initial “” or < is found, to
   handle #include FILE where FILE is a #define somewhere else.
restore () → None
   Pops the previous dispatch table off the stack and makes it the current one.
save () → None
   Pushes the current dispatch table on the stack and re-initializes the current dispatch table to the default.
scons_current_file (t) → None
start_handling_includes (t=None) → None
   Causes the PreProcessor object to start processing #import, #include and #include_next lines.
   This method will be called when a #if, #ifdef, #ifndef or #elif evaluates True, or when we reach the #else in a #if,
   #ifdef, #ifndef or #elif block where a condition already evaluated False.
stop_handling_includes (t=None) → None
   Causes the PreProcessor object to stop processing #import, #include and #include_next lines.
   This method will be called when a #if, #ifdef, #ifndef or #elif evaluates False, or when we reach the #else in a #if,
   #ifdef, #ifndef or #elif block where a condition already evaluated True.
tupleize (contents)
   Turns the contents of a file into a list of easily-processed tuples describing the CPP lines in the file.
   The first element of each tuple is the line’s preprocessor directive (#if, #include, #define, etc., minus the initial ‘#’).
   The remaining elements are specific to the type of directive, as pulled apart by the regular expression.
class SCons.Scanner.C.SConsCPPScannerWrapper (name, variable)
Bases: object
   The SCons wrapper around a cpp.py scanner.
   This is the actual glue between the calling conventions of generic SCons scanners, and the (subclass of) cpp.py
   class that knows how to look for #include lines with reasonably real C-preprocessor-like evaluation of
   #if/#ifdef/#else/#elif lines.
   recurse_nodes (nodes)
   select (node)
SCons.Scanner.C.dictify_CPPDEFINES (env) → dict
   Returns CPPDEFINES converted to a dict.
   This should be similar to processDefines(). Unfortunately, we can’t do the simple thing of calling that routine and
   passing the result to the dict() constructor, because it turns the defines into a list of “name=value” pairs, which the
dict constructor won’t consume correctly. Also cannot just call dict on CPPDEFINES itself - it’s fine if it’s stored in the
converted form (currently deque of tuples), but CPPDEFINES could be in other formats too.
   So we have to do all the work here - keep concepts in sync with processDefines.

SCons.Scanner.D module

Scanner for the Digital Mars “D” programming language.

Coded by Andy Friesen, 17 Nov 2003
class SCons.Scanner.D.D
Bases: Classic
   __call__ (node, env, path=()) → list
   Scans a single object.
SCons.Scanner package

Parameters:
- **node** – the node that will be passed to the scanner function
- **env** – the environment that will be passed to the scanner function.
- **path** – tuple of paths from the `path_function`

Returns: A list of direct dependency nodes for the specified node.

```python
def add_scanner(skey, scanner):
    return None
```

```python
def add_skey(skey):
    return None
```

```python
def scan(node, path=None):
    return list
```

```python
def select(node):
```

```python
def sort_key(include):
```

SCons.Scanner.D.DScanner()

Return a prototype Scanner instance for scanning D source files

SCons.Scanner.Dir module

SCons.Scanner.Dir.DirEntryScanner(**kwargs)

Return a prototype Scanner instance for “scanning” directory Nodes for their in-memory entries

SCons.Scanner.Dir.DirScanner(**kwargs)

Return a prototype Scanner instance for scanning directories for on-disk files

SCons.Scanner.Dir.do_not_scan(k)

SCons.Scanner.Dir.only_dirs(nodes)

SCons.Scanner.Dir.scan_in_memory(node, env, path=None)

“Scans” a Node.FS.Dir for its in-memory entries.

SCons.Scanner.Dir.scan_on_disk(node, env, path=None)

Scans a directory for on-disk files and directories therein.

Looking up the entries will add these to the in-memory Node tree representation of the file system, so all we have to do is just that and then call the in-memory scanning function.

SCons.Scanner.Fortran module

Dependency scanner for Fortran code.

```python
class SCons.Scanner.Fortran.F90Scanner(name, suffixes, path_variable, use_regex, incl_regex, def_regex, *args, **kwargs):
```

Bases: Classic

A Classic Scanner subclass for Fortran source files which takes into account both USE and INCLUDE statements. This scanner will work for both F77 and F90 (and beyond) compilers. Currently, this scanner assumes that the include files do not contain USE statements. To enable the ability to deal with USE statements in include files, add logic right after the module names are found to loop over each include file, search for and locate each USE statement, and append each module name to the list of dependencies. Caching the search results in a common dictionary somewhere so that the same include file is not searched multiple times would be a smart thing to do.

```python
__call__(node, env, path=None) → list
```

Scans a single object.

Parameters:
- **node** – the node that will be passed to the scanner function
- **env** – the environment that will be passed to the scanner function.
- **path** – tuple of paths from the `path_function`
SCons.Scanner package

**Returns:** A list of direct dependency nodes for the specified node.

```python
class SCons.Scanner.Fortran.FortranScan (path_variable: str = 'FORTRANPATH')
    Return a prototype Scanner instance for scanning source files for Fortran USE & INCLUDE statements

class SCons.Scanner.IDL.IDLScan ()
    Return a prototype Scanner instance for scanning IDL source files

class SCons.Scanner.Java.JavaScanner ()
    Scanner for .java files.
    New in version 4.4.

class SCons.Scanner.Java._collect_classes (classlist, dirname, files) → None

class SCons.Scanner.Java._subst_paths (env, paths) → list
    Return a list of substituted path elements.
    If paths is a string, it is split on the search-path separator. Otherwise, substitution is done on string-valued list elements but they are not split.
    Note helps support behavior like pulling in the external CLASSPATH and setting it directly into JAVACLASSPATH, however splitting on os.pathsep makes the interpretation system-specific (this is warned about in the manpage entry for JAVACLASSPATH).

SCons.Scanner.Java.scan (node, env, libpath=()) → list
    Scan for files both on JAVACLASSPATH and JAVAPROCESSORPATH.
    JAVACLASSPATH/JAVAPROCESSORPATH path can contain:
    - Explicit paths to JAR/Zip files
    - Wildcards (*)
    - Directories which contain classes in an unnamed package
    - Parent directories of the root package for classes in a named package
    Class path entries that are neither directories nor archives (.zip or JAR files) nor the asterisk (*) wildcard character are ignored.

SCons.Scanner.LaTeX module

Dependency scanner for LaTeX code.

```python
class SCons.Scanner.LaTeX.FindENVPathDirs (variable)
    Bases: object
    A class to bind a specific E[*]PATH variable name to a function that will return all of the E[*]path directories.

class SCons.Scanner.LaTeX.LaTeX (name, suffixes, graphics_extensions, *args, **kwargs)
    Bases: ScannerBase
    Class for scanning LaTeX files for included files.
```
Unlike most scanners, which use regular expressions that just return the included file name, this returns a tuple consisting of the keyword for the inclusion ("include", "includegraphics", "input", or "bibliography"), and then the file name itself. Based on a quick look at LaTeX documentation, it seems that we should append .tex suffix for the "include" keywords, append .tex if there is no extension for the "input" keyword, and need to add .bib for the "bibliography" keyword that does not accept extensions by itself.

Finally, if there is no extension for an "includegraphics" keyword latex will append .ps or .eps to find the file, while pdftex may use .pdf, .jpg, .tif, .mps, or .png.

The actual subset and search order may be altered by DeclareGraphicsExtensions command. This complication is ignored. The default order corresponds to experimentation with TeTeX:

```
$ latex --version
pdfeTeX 3.141592-1.21a-2.2 (Web2C 7.5.4)
kpathsea version 3.5.4
```

The order is:

['.eps', '.ps'] for latex ['.png', '.pdf', '.jpg', '.tif']

Another difference is that the search path is determined by the type of the file being searched: env['TEXINPUTS'] for "input" and "include" keywords env['TEXINPUTS'] for "includegraphics" keyword env['TEXINPUTS'] for "listinputlisting" keyword env['BIBINPUTS'] for "bibliography" keyword env['BSTINPUTS'] for "bibliographystyle" keyword env['INDEXSTYLE'] for "makeindex" keyword, no scanning support needed just allows user to set it if needed.

FIXME: also look for the class or style in document{class|style}{}  FIXME: also look for the argument of bibliographystyle{}  __call__ (node, env, path=())  →  list

Scans a single object.

**Parameters:**

- **node** – the node that will be passed to the scanner function
- **env** – the environment that will be passed to the scanner function.
- **path** – tuple of paths from the path function

**Returns:** A list of direct dependency nodes for the specified node.

_class_names (include_type, filename)

_static _recurse_all_nodes (nodes)

_static _recurse_no_nodes (nodes)

add_scanner (skey, scanner)  →  None

add_skey (skey)  →  None

Add a skey to the list of skyes

canonical_text (text)

Standardize an input TeX-file contents.

**Currently:**

- removes comments, unwrapping comment-wrapped lines.

```python
env_variables = [‘TEXINPUTS’, ‘BIBINPUTS’, ‘BSTINPUTS’, ‘INDEXSTYLE’]
find_include (include, source_dir, path)
get_skeys (env=None)
path (env, dir=None, target=None, source=None)
scan (node, subdir: str = ‘.’)
scan_recurse (node, path=())
do a recursive scan of the top level target file This lets us search for included files based on the directory of the main file just as latex does
select (node)
```

264
SCons.Scanner package

    static sort_key(include)
        two_arg_commands = ['import', 'subimport', 'includefrom', 'subincludefrom', 'inputfrom', 'subinputfrom']
SCons.Scanner.LaTeX.LaTeXScanner ()
    Return a prototype Scanner instance for scanning LaTeX source files when built with latex.
SCons.Scanner.LaTeX.PDFLaTeXScanner ()
    Return a prototype Scanner instance for scanning LaTeX source files when built with pdflatex.
class SCons.Scanner.LaTeX._Null
    Bases: object
SCons.Scanner.LaTeX._null
    alias of _Null
SCons.Scanner.LaTeX.modify_env_var (env, var, abspath)

SCons.Scanner.Prog module

Dependency scanner for program files.
SCons.Scanner.Prog.ProgramScanner (**kwargs)
    Return a prototype Scanner instance for scanning executable files for static-lib dependencies
SCons.Scanner.Prog._subst_libs (env, libs)
    Substitute environment variables and split into list.
SCons.Scanner.Prog.scan (node, env, libpath=())
    Scans program files for static-library dependencies.
    It will search the LIBPATH environment variable for libraries specified in the LIBS variable, returning any files it finds as dependencies.

SCons.Scanner.RC module

Dependency scanner for RC (Interface Definition Language) files.
SCons.Scanner.RC.RCScan ()
    Return a prototype Scanner instance for scanning RC source files
SCons.Scanner.RC.no_tlb (nodes)
    Filter out .tlb files as they are binary and shouldn't be scanned.

SCons.Scanner.SWIG module

Dependency scanner for SWIG code.
SCons.Scanner.SWIG.SWIGScanner ()

Module contents

The Scanner package for the SCons software construction utility.
SCons.Scanner.Base
    alias of ScannerBase
class SCons.Scanner.Classic (name, suffixes, path_variable, regex, *args, **kwargs)
    Bases: Current
    A Scanner subclass to contain the common logic for classic CPP-style include scanning, but which can be customized to use different regular expressions to find the includes.
    Note that in order for this to work "out of the box" (without overriding the find_include() and sort_key1() methods), the regular expression passed to the constructor must return the name of the include file in group 0.
    __call__ (node, env, path=()) → list
        Scans a single object.
     Parameters:
            • node – the node that will be passed to the scanner function
            • env – the environment that will be passed to the scanner function.
            • path – tuple of paths from the path_function
    Returns:  A list of direct dependency nodes for the specified node.
static _recurse_all_nodes (nodes)
SCons.Scanner package

```python
static _recurse_no_nodes(nodes)
add_scanner(skey, scanner) → None
add_skey(skey) → None

Add a skey to the list of skeys

static find_include(include, source_dir, path)
find_include_names(node)
generate_skeys(env= None)
path(env, dir= None, target= None, source= None)
scan(node, path=())
select(node)

static sort_key(include)

class SCons.Scanner.ClassicCPP(name, suffixes, path_variable, regex, *args, **kwargs)

Bases: Classic
A Classic Scanner subclass which takes into account the type of bracketing used to include the file, and uses classic
CPP rules for searching for the files based on the bracketing.
Note that in order for this to work, the regular expression passed to the constructor must return the leading bracket in
group 0, and the contained filename in group 1.
__call__(node, env, path=()) → list
Scans a single object.

Parameters:

- `node` – the node that will be passed to the scanner function
- `env` – the environment that will be passed to the scanner function.
- `path` – tuple of paths from the `path_function`

Returns: A list of direct dependency nodes for the specified node.

static _recurse_all_nodes(nodes)
static _recurse_no_nodes(nodes)
add_scanner(skey, scanner) → None
add_skey(skey) → None

Add a skey to the list of skeys

static find_include(include, source_dir, path)
find_include_names(node)
generate_skeys(env= None)
path(env, dir= None, target= None, source= None)
scan(node, path=())
select(node)

static sort_key(include)

class SCons.Scanner.Current(*args, **kwargs)

Bases: ScannerBase
A class for scanning files that are source files (have no builder) or are derived files and are current (which implies that
they exist, either locally or in a repository).
__call__(node, env, path=()) → list
Scans a single object.

Parameters:

- `node` – the node that will be passed to the scanner function
- `env` – the environment that will be passed to the scanner function.
- `path` – tuple of paths from the `path_function`

Returns: A list of direct dependency nodes for the specified node.

static _recurse_all_nodes(nodes)
static _recurse_no_nodes(nodes)
add_scanner(skey, scanner) → None
add_skey(skey) → None

Add a skey to the list of skeys

generate_skeys(env= None)
```

266
SCons.Scanner package

    path (env, dir=None, target=None, source=None)
    select (node)

class SCons.Scanner.FindPathDirs (variable)
    Bases: object
    Class to bind a specific E[*]PATH variable name to a function that will return all of the E[*]path directories.

SCons.Scanner.Scanner (function, *args, **kwargs)
    Factory function to create a Scanner Object.
    Creates the appropriate Scanner based on the type of “function”.
    TODO: Deprecate this some day. We've moved the functionality inside the ScannerBase class and really don't need
    this factory function any more. It was, however, used by some of our Tool modules, so the call probably ended up in
    various people's custom modules patterned on SCons code.

class SCons.Scanner.ScannerBase (function, name: str = 'NONE', argument=<class 
SCons.Scanner._Null'>, skeys=<class 'SCons.Scanner._Null'>, path_function=None,
node_class=<class 'SCons.Node.FS.Base'>, node_factory=None, scan_check=None,
recursive=None)
    Bases: object
    Base class for dependency scanners.
    Implements straightforward, single-pass scanning of a single file.
    A Scanner is usually set up with a scanner function (and optionally a path function), but can also be a kind of
dispenser which passes control to other Scanners.
    A scanner function takes three arguments: a Node to scan for dependencies, the construction environment to use, and
an optional tuple of paths (as generated by the optional path function). It must return a list containing the Nodes for all
the direct dependencies of the file.
    The optional path function is called to return paths that can be searched for implicit dependency files. It takes five
arguments: a construction environment, a Node for the directory containing the SConscript file that defined the
primary target, a list of target nodes, a list of source nodes, and the optional argument for this instance.
    Examples:

        s = Scanner(my_scanner_function)
        s = Scanner(function=my_scanner_function)
        s = Scanner(function=my_scanner_function, argument='foo')
Parameters:

- **function** – either a scanner function taking two or three arguments and returning a list of File Nodes; or a mapping of keys to other Scanner objects.
- **name** – an optional name for identifying this scanner object (defaults to “NONE”).
- **argument** – an optional argument that will be passed to both function and path_function.
- **skeys** – an optional list argument that can be used to determine if this scanner can be used for a given Node. In the case of File nodes, for example, the skeys would be file suffixes.
- **path_function** – an optional function which returns a tuple of the directories that can be searched for implicit dependency files. May also return a callable which is called with no args and returns the tuple (supporting Bindable class).
- **node_class** – optional class of Nodes which this scan will return. If not specified, defaults to SCons.Node.FS.Base. If node_class is None, then this scanner will not enforce any Node conversion and will return the raw results from function.
- **node_factory** – optional factory function to be called to translate the raw results returned by function into the expected node_class objects.
- **scan_check** – optional function to be called to first check whether this node really needs to be scanned.
- **recursive** – optional specifier of whether this scanner should be invoked recursively on all of the implicit dependencies it returns (for example #include lines in C source files, which may refer to header files which should themselves be scanned). May be a callable, which will be called to filter the list of nodes found to select a subset for recursive scanning (the canonical example being only recursively scanning subdirectories within a directory). The default is to not do recursive scanning.

```
__call__ (node, env, path=()) → list
```

Scans a single object.

Parameters:

- **node** – the node that will be passed to the scanner function
- **env** – the environment that will be passed to the scanner function.
- **path** – tuple of paths from the path_function

Returns: A list of direct dependency nodes for the specified node.

```
static _recurse_all_nodes (nodes)
static _recurse_no_nodes (nodes)
add_scanner (skey, scanner) → None
add_skey (skey) → None
  Add a key to the list of skeys
get_skeys (env=None)
path (env, dir=None, target=None, source=None)
select (node)
```

class SCons.Scanner.Selector (mapping, *args, **kwargs)

Bases: ScannerBase

A class for selecting a more specific scanner based on the scanner_key() (suffix) for a specific Node.

TODO: This functionality has been moved into the inner workings of the ScannerBase class, and this class will be deprecated at some point. (It was never exposed directly as part of the public interface, although it is used by the Scanner() factory function that was used by various Tool modules and therefore was likely a template for custom modules that may be out there.)

```
static _recurse_all_nodes (nodes)
static _recurse_no_nodes (nodes)
add_scanner (skey, scanner) → None
add_skey (skey) → None
  Add a key to the list of skeys
```
SCons.Script package

```python
get_keys (env=None)
path (env, dir=None, target=None, source=None)
select (node)
```

```python
class SCons.Scanner._Null
    Bases: object
    SCons.Scanner._null
    alias of _Null
```

SCons.Script package

Submodules

SCons.Script.Interactive module

SCons interactive mode.

```python
class SCons.Script.Interactive.SConsInteractiveCmd (**kw)
    Bases: Cmd
    build [TARGETS] Build the specified TARGETS and their dependencies. 'b' is a synonym.
    clean [TARGETS] Clean (remove) the specified TARGETS and their dependencies. 'c' is a synonym.
    exit Exit SCons interactive mode.
    help [COMMAND] Prints help for the specified COMMAND. 'h' and '?' are synonyms.
    shell [COMMANDLINE] Execute COMMANDLINE in a subshell. 'sh' and '!' are synonyms.
    version Prints SCons version information.
```

```python
_do_one_help (arg) → None
_doc_to_help (obj)
_strip_initial_spaces (s)
```

```python
cmdloop (intro=None)
    Repeatedly issue a prompt, accept input, parse an initial prefix off the received input, and dispatch to action
    methods, passing them the remainder of the line as argument.
```

```python
columnize (list, displaywidth=80)
    Display a list of strings as a compact set of columns.
    Each column is only as wide as necessary. Columns are separated by two spaces (one was not legible enough).
```

```python
complete (text, state)
    Return the next possible completion for 'text'.
    If a command has not been entered, then complete against command list. Otherwise try to call
    complete_<command> to get list of completions.
```

```python
complete_help (*args)
    completedefault (*ignored)
    Method called to complete an input line when no command-specific complete_*(*) method is available.
    By default, it returns an empty list.
```

```python
completenames (text, *ignored)
    default (argv) → None
    Called on an input line when the command prefix is not recognized.
    If this method is not overridden, it prints an error message and returns.
```

```python
do_EOF (argv) → None
```

```python
do_build (argv) → None
    build [TARGETS] Build the specified TARGETS and their dependencies. 'b' is a synonym.
```

```python
do_clean (argv)
    clean [TARGETS] Clean (remove) the specified TARGETS and their dependencies. 'c' is a synonym.
```

```python
do_exit (argv) → None
    exit Exit SCons interactive mode.
```

```python
do_help (argv) → None
    help [COMMAND] Prints help for the specified COMMAND. 'h' and '?' are synonyms.
```

```python
do_shell (argv) → None
    shell [COMMANDLINE] Execute COMMANDLINE in a subshell. 'sh' and '!' are synonyms.
```

```python
do_version (argv) → None
    version Prints SCons version information.
```

```python
doc_header = 'Documented commands (type help <topic>):'
```
SCons.Script package

```
doc_leader = ''
emptyline ()
    Called when an empty line is entered in response to the prompt.
    If this method is not overridden, it repeats the last nonempty command entered.
get_names ()
identchars = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_'
intro = None
lastcmd = ''
misc_header = 'Miscellaneous help topics:'
nohelp = '*** No help on %s'
onecmd (line)
    Interpret the argument as though it had been typed in response to the prompt.
    This may be overridden, but should not normally need to be; see the precmd() and postcmd() methods for useful
    execution hooks. The return value is a flag indicating whether interpretation of commands by the interpreter should
    stop.
parseline (line)
    Parse the line into a command name and a string containing the arguments. Returns a tuple containing (command,
    args, line). 'command' and 'args' may be None if the line couldn't be parsed.
postcmd (stop, line)
    Hook method executed just after a command dispatch is finished.
postloop ()
    Hook method executed once when the cmdloop() method is about to return.
predcm (line)
    Hook method executed just before the command line is interpreted, but after the input prompt is generated and
    issued.
preloop ()
    Hook method executed once when the cmdloop() method is called.
print_topics (header, cmds, cmdlen, maxcol)
prompt = '(Cmd) '
ruler = '='
synonyms = { 'b': 'build', 'c': 'clean', 'h': 'help', 'scons': 'build', 'sh': 'shell' }
undoc_header = 'Undocumented commands:'
use_rawinput = 1
SCons.Script.Interactive.interact (fs, parser, options, targets, target_top) → None
```

SCons.Script.Main module

The main() function used by the scons script.

Architecturally, this is the scons script, and will likely only be called from the external “scons” wrapper. Consequently,
anything here should not be, or be considered, part of the build engine. If it’s something that we expect other software
to want to use, it should go in some other module. If it’s specific to the “scons” script invocation, it goes here.

SCons.Script.Main.AddOption (*args, **kw)
class SCons.Script.Main.BuildTask (tm, targets, top, node)
    Bases: OutOfDateTask
    An SCons build task.
    LOGGER = None
    _abc_impl = <_abc._abc_data object>
    _exception_raise ()
        Raises a pending exception that was recorded while getting a Task ready for execution.
    _no_exception_to_raise () → None
    display (message) → None
        Hook to allow the calling interface to display a message.
        This hook gets called as part of preparing a task for execution (that is, a Node to be built). As part of figuring out
        what Node should be built next, the actual target list may be altered, along with a message describing the
        alteration. The calling interface can subclass Task and provide a concrete implementation of this method to see
        those messages.
SCons.Script package

do_failed (status: int = 2) → None
exc_clear () → None
    Clears any recorded exception.
    This also changes the “exception_raise” attribute to point to the appropriate do-nothing method.
exc_info ()
    Returns info about a recorded exception.
exception_set (exception=None) → None
    Records an exception to be raised at the appropriate time.
    This also changes the “exception_raise” attribute to point to the method that will, in fact
execute () → None
    Called to execute the task.
    This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in prepare(), executed() or failed().
executed ()
    Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s callback methods.
    This may have been a do-nothing operation (to preserve build order), so we must check the node’s state before deciding whether it was “built”, in which case we call the appropriate Node method. In any event, we always call “visited()”, which will handle any post-visit actions that must take place regardless of whether or not the target was an actual built target or a source Node.
executed_with_callbacks () → None
    Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s callback methods.
    This may have been a do-nothing operation (to preserve build order), so we must check the node’s state before deciding whether it was “built”, in which case we call the appropriate Node method. In any event, we always call “visited()”, which will handle any post-visit actions that must take place regardless of whether or not the target was an actual built target or a source Node.
executed_without_callbacks () → None
    Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s callback methods.
fail_continue () → None
    Explicit continue-the-build failure.
    This sets failure status on the target nodes and all of their dependent parent nodes.
    Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().
fail_stop () → None
    Explicit stop-the-build failure.
    This sets failure status on the target nodes and all of their dependent parent nodes.
    Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().
failed () → None
    Default action when a task fails: stop the build.
    Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().
get_target ()
    Fetch the target being built or updated by this task.
make_ready () → None
    Make a task ready for execution
make_ready_all () → None
    Marks all targets in a task ready for execution.
    This is used when the interface needs every target Node to be visited—the canonical example being the “scons -c” option.
make_ready_current ()
    Marks all targets in a task ready for execution if any target is not current.
    This is the default behavior for building only what’s necessary.
needs_execute () → bool
Returns True (indicating this Task should be executed) if this Task's target state indicates it needs executing, which has already been determined by an earlier up-to-date check.

postprocess () → None
Post-processes a task after it's been executed.
This examines all the targets just built (or not, we don’t care if the build was successful, or even if there was no build because everything was up-to-date) to see if they have any waiting parent Nodes, or Nodes waiting on a common side effect, that can be put back on the candidates list.

prepare ()
Called just before the task is executed.
This is mainly intended to give the target Nodes a chance to unlink underlying files and make all necessary directories before the Action is actually called to build the targets.

trace_message (node, description: str = 'node') → None

class SCons.Script.Main.CleanTask (tm, targets, top, node)
Bases: AlwaysTask
An SCons clean task.
LOGGER = None
_abc_impl = <_abc._abc_data object>
_clean_targets (remove: bool = True) → None
_exception_raise ()
 Raises a pending exception that was recorded while getting a Task ready for execution.
_get_files_to_clean ()
_no_exception_to_raise () → None
display (message) → None
Hook to allow the calling interface to display a message.
This hook gets called as part of preparing a task for execution (that is, a Node to be built). As part of figuring out what Node should be built next, the actual target list may be altered, along with a message describing the alteration. The calling interface can subclass Task and provide a concrete implementation of this method to see those messages.
exc_clear () → None
Clears any recorded exception.
This also changes the “exception_raise” attribute to point to the appropriate do-nothing method.
exc_info ()
Returns info about a recorded exception.
exception_set (exception=None) → None
Records an exception to be raised at the appropriate time.
This also changes the “exception_raise” attribute to point to the method that will, in fact
execute () → None
Called to execute the task.
This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in prepare(), executed() or failed().
executed () → None
Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s callback methods.
executed_with_callbacks () → None
Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s callback methods.
executed_without_callbacks () → None
Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s callback methods.
fail_continue () → None
Explicit continue-the-build failure.
This sets failure status on the target nodes and all of their dependent parent nodes.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

fail_stop () → None
    Explicit stop-the-build failure.
    This sets failure status on the target nodes and all of their dependent parent nodes.

failed () → None
    Default action when a task fails: stop the build.
    Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

fs_delete (path, pathstr, remove: bool = True)

get_target ()
    Fetch the target being built or updated by this task.

make_ready () → None
    Marks all targets in a task ready for execution.
    This is used when the interface needs every target Node to be visited—the canonical example being the “scons -c” option.

make_ready_all () → None
    Marks all targets in a task ready for execution.
    This is used when the interface needs every target Node to be visited—the canonical example being the “scons -c” option.

make_ready_current ()
    Marks all targets in a task ready for execution if any target is not current.
    This is the default behavior for building only what’s necessary.

needs_execute () → bool
    Always returns True (indicating this Task should always be executed).
    Subclasses that need this behavior (as opposed to the default of only executing Nodes that are out of date w.r.t. their dependencies) can use this as follows:

    class MyTaskSubclass(SCons.Taskmaster.Task):
        needs_execute = SCons.Taskmaster.AlwaysTask.needs_execute

postprocess () → None
    Post-processes a task after it’s been executed.
    This examines all the targets just built (or not, we don’t care if the build was successful, or even if there was no build because everything was up-to-date) to see if they have any waiting parent Nodes, or Nodes waiting on a common side effect, that can be put back on the candidates list.

prepare () → None
    Called just before the task is executed.
    This is mainly intended to give the target Nodes a chance to unlink underlying files and make all necessary directories before the Action is actually called to build the targets.

remove () → None

show () → None

trace_message (node, description: str = 'node') → None

SCons.Script.Main.DebugOptions (json=None)
    API to allow specifying options to SCons debug logic Currently only json is supported which changes the json file written by –debug=json from the default

class SCons.Script.Main.FakeOptionParser
    Bases: object
    A do-nothing option parser, used for the initial OptionsParser value.
    During normal SCons operation, the OptionsParser is created right away by the main() function. Certain test scripts however, can introspect on different Tool modules, the initialization of which can try to add a new, local option to an otherwise uninitialized OptionsParser object. This allows that introspection to happen without blowing up.

class FakeOptionValues
    Bases: object
    add_local_option (*args, **kw) → None
SCons.Script package

values = <SCons.Script.Main.FakeOptionParser.FakeOptionValues object>
SCons.Script.Main.GetBuildFailures ()
SCons.Script.Main.GetOption (name)
SCons.Script.Main.PrintHelp (file=None, local_only: bool = False) → None
SCons.Script.Main.Progress (*args, **kw) → None
class SCons.Script.Main.Progressor (obj, interval: int = 1, file=None, overwrite: bool = False)
   Bases: object
   count = 0
   erase_previous () → None
   prev = ''
   replace_string (node) → None
   spinner (node) → None
   string (node) → None
   target_string = ['$TARGET'
   write (s) → None
class SCons.Script.Main.QuestionTask (tm, targets, top, node)
   Bases: AlwaysTask
   An SCons task for the -q (question) option.
   LOGGER = None
   _abc_impl = <_abc._abc_data object>
   _exception_raise ()
      Raises a pending exception that was recorded while getting a Task ready for execution.
   _no_exception_to_raise () → None
   display (message) → None
      Hook to allow the calling interface to display a message.
      This hook gets called as part of preparing a task for execution (that is, a Node to be built). As part of figuring out
      what Node should be built next, the actual target list may be altered, along with a message describing the
      alteration. The calling interface can subclass Task and provide a concrete implementation of this method to see
      those messages.
   exc_clear () → None
      Clears any recorded exception.
      This also changes the "exception_raise" attribute to point to the appropriate do-nothing method.
   exc_info ()
      Returns info about a recorded exception.
   exception_set (exception=None) → None
      Records an exception to be raised at the appropriate time.
      This also changes the "exception_raise" attribute to point to the method that will, in fact
   execute () → None
      Called to execute the task.
      This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe
      stuff in prepare(), executed() or failed().
   executed () → None
      Called when the task has been successfully executed and the Taskmaster instance wants to call the Node's
      callback methods.
      This may have been a do-nothing operation (to preserve build order), so we must check the node's state before
      deciding whether it was "built", in which case we call the appropriate Node method. In any event, we always call
      "visited()", which will handle any post-visit actions that must take place regardless of whether or not the target was
      an actual built target or a source Node.
   executed_with_callbacks () → None
      Called when the task has been successfully executed and the Taskmaster instance wants to call the Node's
      callback methods.
      This may have been a do-nothing operation (to preserve build order), so we must check the node's state before
      deciding whether it was "built", in which case we call the appropriate Node method. In any event, we always call
      "visited()", which will handle any post-visit actions that must take place regardless of whether or not the target was
      an actual built target or a source Node.
   executed_without_callbacks () → None
Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s callback methods.

```
fail_continue () → None
```

Explicit continue-the-build failure.

This sets failure status on the target nodes and all of their dependent parent nodes.

Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

```
fail_stop () → None
```

Explicit stop-the-build failure.

This sets failure status on the target nodes and all of their dependent parent nodes.

Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

```
failed () → None
```

Default action when a task fails: stop the build.

Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

```
get_target ()
```

Fetch the target being built or updated by this task.

```
make_ready ()
```

Marks all targets in a task ready for execution if any target is not current.

This is the default behavior for building only what’s necessary.

```
make_ready_all () → None
```

Marks all targets in a task ready for execution.

This is used when the interface needs every target Node to be visited—the canonical example being the “scons -c” option.

```
make_ready_current ()
```

Marks all targets in a task ready for execution if any target is not current.

This is the default behavior for building only what’s necessary.

```
needs_execute () → bool
```

Always returns True (indicating this Task should always be executed).

Subclasses that need this behavior (as opposed to the default of only executing Nodes that are out of date w.r.t. their dependencies) can use this as follows:

```python
class MyTaskSubclass(SCons.Taskmaster.Task):
    needs_execute = SCons.Taskmaster.AlwaysTask.needs_execute
```

```
postprocess () → None
```

Post-processes a task after it’s been executed.

This examines all the targets just built (or not, we don’t care if the build was successful, or even if there was no build because everything was up-to-date) to see if they have any waiting parent Nodes, or Nodes waiting on a common side effect, that can be put back on the candidates list.

```
prepare () → None
```

Called just before the task is executed.

This is mainly intended to give the target Nodes a chance to unlink underlying files and make all necessary directories before the Action is actually called to build the targets.

```
trace_message (node, description: str = 'node') → None
```

**Classes**

*SCons.Script.Main.TreePrinter*

```
Bases: object
```

```
```

```
SCons.Script.Main.SetOption (name, value)
```

**Exceptions**

*SCons.Script.Main.SConsPrintHelpException*

```
Bases: Exception
```

```
add_note ()
```

Exception.add__note(note) – add a note to the exception

```
args
```

with_traceback ()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.
display (t) → None
get_all_children (node)
get_derived_children (node)

SCons.Script.Main.ValidateOptions (throw_exception: bool = False) → None
Validate options passed to SCons on the command line.
Checks that all options given on the command line are known to this instance of SCons. Call after all of the cli options have been set up through AddOption() calls. For example, if you added an option --xyz and you call SCons with --xyy you can cause SCons to issue an error message and exit by calling this function.

Parameters:  
  throw_exception – if an invalid option is present on the command line, raises an exception if this optional parameter evaluates true; if false (the default), issue a message and exit with error status.

Raises:  
  SConsBadOptionError – If throw_exception is true and there are invalid options on the command line.

New in version 4.5.0.

SCons.Script.Main._SConstruct_exists (dirname: str, repositories: List[str], filelist: List[str]) → str | None
Check that an SConstruct file exists in a directory.

Parameters:
  
  * dirname – the directory to search. If empty, look in cwd.
  * repositories – a list of repositories to search in addition to the project directory tree.
  * filelist – names of SConstruct file(s) to search for. If empty list, use the built-in list of names.

Returns:  
  The path to the located SConstruct file, or None.

SCons.Script.Main._build_targets (fs, options, targets, target_top)
SCons.Script.Main._create_path (plist)
SCons.Script.Main._exec_main (parser, values) → None
SCons.Script.Main._load_all_site_scons_dirs (topdir, verbose: bool = False) → None
  Load all of the predefined site_scons dir. Order is significant; we load them in order from most generic (machine-wide) to most specific (topdir). The verbose argument is only for testing.
SCons.Script.Main._load_site_scons_dir (topdir, site_dir_name=None)
  Load the site directory under todir.
  If a site dir name is supplied use it, else use default “site_scons” Prepend site dir to sys.path. If a “site_tools” subdir exists, prepend to toolpath. Import “site_init.py” from site dir if it exists.
SCons.Script.Main._main (parser)
SCons.Script.Main._scons_internal_error () → None
  Handle all errors but user errors. Print out a message telling the user what to do in this case and print a normal trace.
SCons.Script.Main._scons_internal_warning (e) → None
  Slightly different from _scons_user_warning in that we use the current call stack rather than sys.exc_info() to get our stack trace. This is used by the warnings framework to print warnings.
SCons.Script.Main._scons_syntax_error (e) → None
  Handle syntax errors. Print out a message and show where the error occurred.
SCons.Script.Main._scons_user_error (e) → None
  Handle user errors. Print out a message and a description of the error, along with the line number and routine where it occurred. The file and line number will be the deepest stack frame that is not part of SCons itself.
SCons.Script.Main._scons_user_warning (e) → None
  Handle user warnings. Print out a message and a description of the warning, along with the line number and routine where it occurred. The file and line number will be the deepest stack frame that is not part of SCons itself.
SCons.Script.Main._set_debug_values (options) → None
SCons.Script.Main.find_deepest_user_frame (tb)
  Find the deepest stack frame that is not part of SCons.
  Input is a “pre-processed” stack trace in the form returned by traceback.extract_tb() or traceback.extract_stack()
SCons.Script.Main.main () → None
SCons.Script.Main.path_string (label, module) → str
SCons.Script package

SCons.Script.Main.python_version_deprecated (version=sys.version_info(major=3, minor=11, micro=8, releaselevel='final', serial=0))
SCons.Script.Main.python_version_string ()
SCons.Script.Main.python_version_unsupported (version=sys.version_info(major=3, minor=11, micro=8, releaselevel='final', serial=0))
SCons.Script.Main.revert_io () → None
SCons.Script.Main.test_load_all_site_scons_dirs (d) → None
SCons.Script.Main.version_string (label, module)

SCons.Script.SConsOptions module

SCons.Script.SConsOptions.Parser (version)
Returns a parser object initialized with the standard SCons options.
Add options in the order we want them to show up in the -H help text, basically alphabetical. Each op.add_option() call should have a consistent format:

```python
op.add_option("-L", "--long-option-name",
    nargs=1, type="string",
    dest="long_option_name", default='foo',
    action="callback", callback=opt_long_option,
    help="help text goes here",
    metavar="VAR")
```

Even though the optparse module constructs reasonable default destination names from the long option names, we’re going to be explicit about each one for easier readability and so this code will at least show up when grepping the source for option attribute names, or otherwise browsing the source code.

**exception**  SCons.Script.SConsOptions.SConsBadOptionError (opt_str, parser=None)
Bases: BadOptionError
Exception used to indicate that invalid command line options were specified

**Variables:**
- `opt_str` (str) – The offending option specified on command line which is not recognized
- `parser` (OptionParser) – The active argument parser

```python
Exception.add_note(note) – add a note to the exception
args
with_traceback ()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.
```

class SCons.Script.SConsOptions.SConsIndentedHelpFormatter (Indent_increment=2,
    max_help_position=24, width=None, short_first=1)
Bases: IndentedHelpFormatter
NO_DEFAULT_VALUE = 'none'
_format_text (text)
- Format a paragraph of free-form text for inclusion in the help output at the current indentation level.
dedent ()
expand_default (option)
format_description (description)
format_epilog (epilog)
format_heading (heading)
- Translates heading to “SCons Options”
Heating of “Options” changed to “SCons Options.” Unfortunately, we have to do this here, because those titles are hard-coded in the optparse calls.
format_option (option)
- Customized option formatter.
A copy of the normal optparse.IndentedHelpFormatter.format_option() method. This has been snarfed so we can modify text wrapping to our liking:
• add our own regular expression that doesn’t break on hyphens (so things like `--no-print-directory` don’t get broken).

• wrap the list of options themselves when it’s too long (the `wrapper.fill(opts)` call below).

• set the subsequent indent when wrapping the help_text. The help for each option consists of two parts:
  
  • the opt strings and metavars e.g. (“-x”, or “-fFILENAME, –file=FILENAME”)
  
  • the user-supplied help string e.g. (“turn on expert mode”, “read data from FILENAME”)

If possible, we write both of these on the same line:

```
-x      turn on expert mode
```

But if the opt string list is too long, we put the help string on a second line, indented to the same column it would start in if it fit on the first line:

```
-fFILENAME, --file=FILENAME
read data from FILENAME
```

format_option_strings (option)
Return a comma-separated list of option strings & metavariables.

format_usage (usage) → str
Formats the usage message.

indent ()

set_long_opt_delimiter (delim)

set_parser (parser)

set_short_opt_delimiter (delim)

store_local_option_strings (parser, group)
Local-only version of store_option_strings.

We need to replicate this so the formatter will be set up properly if we didn’t go through the “normal”
store_option_strings.

New in version 4.6.0.

store_option_strings (parser)

class SCons.Script.SConsOptions.SConsOption (*opts, **attrs)
Bases: Option

ACTIONS = ('store', 'store_const', 'store_true', 'store_false', 'append', 'append_const', 'count', 'callback', 'help', 'version')

ALWAYS_TYPED_ACTIONS = ('store', 'append')

ATTRS = ['action', 'type', 'dest', 'default', 'nargs', 'const', 'choices', 'callback', 'callback_args', 'callback_kwvars', 'help', 'metavar']

CHECK_METHODS = [function Option._check_action>, <function Option._check_type>, <function Option._check_choice>, <function Option._check_dest>, <function Option._check_const>, <function Option._check_nargs>, <function SConsOption._check_nargs_optional>]

CONST_ACTIONS = ('store_const', 'append_const', 'store', 'append', 'callback')

STORE_ACTIONS = ('store', 'store_const', 'store_true', 'store_false', 'append', 'append_const', 'count')

TYPED_ACTIONS = ('store', 'append', 'callback')

TYPE_CHECKER = (string', 'int', 'long', 'float', 'complex', 'choice')

TYPE_CHECKER = (choice': <function check_choice>, 'complex': <function check_builtin>, 'float': <function check_builtin>, 'int': <function check_builtin>, 'long': <function check_builtin>)
class SCons.Script.SConsOptions.SConsOptionGroup (parser, title, description=None)

    A subclass for SCons-specific option groups.
    The only difference between this and the base class is that we print the group's help text flush left, underneath their own title but lined up with the normal "SCons Options".

    _check_conflict (option)
    _create_option_list ()
    _create_option_mappings ()
    _share_option_mappings (parser)
    add_option (Option)
    add_option (opt_str, ..., kwarg=val, ...) → None
    add_options (option_list)
    destroy ()
    see OptionParser.destroy().
    format_description (formatter)
    format_help (formatter)
    Format an option group's help text.
    The title is dedented so it's flush with the "SCons Options" title we print at the top.
    format_option_help (formatter)
    get_description ()
    get_option (opt_str)
    has_option (opt_str)
    remove_option (opt_str)
    set_conflict_handler (handler)
    set_description (description)
    set_title (title)

class SCons.Script.SConsOptions.SConsOptionParser (usage=None, option_list=None,
    option_class=<class 'optparse.Option'>, version=None, conflict_handler='error',
    description=None, formatter=None, add_help_option=True, prog=None, epilog=None)

    Bases: OptionParser
    _add_help_option ()
    _add_version_option ()
    _check_conflict (option)
    _create_option_list ()
    _create_option_mappings ()
    _get_all_options ()
    _get_args (args)
    _init_parsing_state ()
    _match_long_opt (opt: string) → string
    Determine which long option string 'opt' matches, ie. which one it is an unambiguous abbreviation for. Raises BadOptionError if 'opt' doesn't unambiguously match any long option string.
    _populate_option_list (option_list, add_help=True)
    _process_args (largs, rargs, values)
    _process_args(largs : [string],
Process command-line arguments and populate ‘values’, consuming options and arguments from ‘rargs’. If ‘allow_interspersed_args’ is false, stop at the first non-option argument. If true, accumulate any interspersed non-option arguments in ‘largs’.

_process_long_opt (rargs, values)
SCons-specific processing of long options.
This is copied directly from the normal optparse._process_long_opt() method, except that, if configured to do so, we catch the exception thrown when an unknown option is encountered and just stick it back on the “leftover” arguments for later (re-)processing. This is because we may see the option definition later, while processing SConscript files.

_process_short_opts (rargs, values)
_share_option_mappings (parser)
add_local_option (*args, **kw)
   Adds a local option to the parser.
   This is initiated by an AddOption() call to add a user-defined command-line option. We add the option to a separate option group for the local options, creating the group if necessary.

add_option (Option)
add_option (opt_str,..., kwarg=val,...) → None
add_option_group (*args, **kwargs)
add_options (option_list)
check_values (values: Values, args: [string])
   -> (values : Values, args : [string])
   Check that the supplied option values and leftover arguments are valid. Returns the option values and leftover arguments (possibly adjusted, possibly completely new – whatever you like). Default implementation just returns the passed-in values; subclasses may override as desired.

destroy ()
Declare that you are done with this OptionParser. This cleans up reference cycles so the OptionParser (and all objects referenced by it) can be garbage-collected promptly. After calling destroy(), the OptionParser is unusable.

disable_interspersed_args ()
Set parsing to stop on the first non-option. Use this if you have a command processor which runs another command that has options of its own and you want to make sure these options don’t get confused.

enable_interspersed_args ()
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior. See also disable_interspersed_args() and the class documentation description of the attribute allow_interspersed_args.

error (msg)
   overridden OptionValueError exception handler
exit (status=0, msg=None)
expend_prog_name (s)
format_description (formatter)
format_epilog (formatter)
format_help (formatter=None)
format_local_option_help (formatter=None, file=None)
   Return the help for the project-level (“local”) options.
   New in version 4.6.0.
format_option_help (formatter=None)
get_default_values ()
get_description ()
get_option (opt_str)
get_option_group (opt_str)
get_prog_name ()
get_usage ()
get_version ()
has_option (opt_str)
parsed_args (args=None, values=None)
**parse_args**

`parse_args(args : [string] = sys.argv[1:],
values : Values = None)
-> (values : Values, args : [string])`

Parse the command-line options found in `args` (default: `sys.argv[1:]`). Any errors result in a call to `error()`, which by default prints the usage message to stderr and calls `sys.exit()` with an error message. On success returns a pair (values, args) where `values` is a `Values` instance (with all your option values) and `args` is the list of arguments left over after parsing options.

- `preserve_unknown_options = False`
- `print_help (file: file = stdout)`
  Print an extended help message, listing all options and any help text provided with them, to `file` (default stdout).
- `print_local_option_help (file=None)`
  Print help for just project-defined options.
  Writes to `file` (default stdout).
  New in version 4.6.0.
- `print_usage (file: file = stdout)`
  Print the usage message for the current program (self.usage) to `file` (default stdout). Any occurrence of the string “%prog” in self.usage is replaced with the name of the current program (basename of sys.argv[0]). Does nothing if self.usage is empty or not defined.
- `print_version (file: file = stdout)`
  Print the version message for this program (self.version) to `file` (default stdout). As with print_usage(), any occurrence of “%prog” in self.version is replaced by the current program’s name. Does nothing if self.version is empty or undefined.
- `raise_exception_on_error = False`
- `remove_option (opt_str)`
- `reparse_local_options () → None`
  Re-parse the leftover command-line options.
  Parse options stored in self.largs, so that any value overridden on the command line is immediately available if the user turns around and does a GetOption() right away.
  We mimic the processing of the single args in the original OptionParser _process_args(), but here we allow exact matches for long-opts only (no partial argument names!). Otherwise there could be problems in add_local_option() below. When called from there, we try to reparse the command-line arguments that

1. haven’t been processed so far (self.largs), but
2. are possibly not added to the list of options yet.
So, when we only have a value for “–myargument” so far, a command-line argument of “–myarg=test” would set it, per the behaviour of _match_long_opt(), which allows for partial matches of the option name, as long as the common prefix appears to be unique. This would lead to further confusion, because we might want to add another option “–myarg” later on (see issue #2929).
- `set_conflict_handler (handler)`
- `set_default (dest, value)`
- `set_defaults (**kwargs)`
- `set_description (description)`
- `set_process_default_values (process)`
- `set_usage (usage)`
- `standard_option_list = []`

**class**

`SCons.Script.SConsOptions.SConsValues (defaults)`

- **Bases**: Values
- **Description**: Holder class for uniform access to SCons options, regardless of whether they can be set on the command line or in the SConscript files (using the SetOption() function).

A SCons option value can originate three different ways:

1. set on the command line;
2. set in an SConscript file;
3. the default setting (from the the op.add_option() calls in the Parser() function, below).
The command line always overrides a value set in a SConscript file, which in turn always overrides default settings. Because we want to support user-specified options in the SConscript file itself, though, we may not know about all of the options when the command line is first parsed, so we can’t make all the necessary precedence decisions at the time the option is configured.

The solution implemented in this class is to keep these different sets of settings separate (command line, SConscript file, and default) and to override the __getattr__() method to check them in turn. This should allow the rest of the code to just fetch values as attributes of an instance of this class, without having to worry about where they came from.

Note that not all command line options are settable from SConscript files, and the ones that are must be explicitly added to the “settable” list in this class, and optionally validated and coerced in the set_option() method.

__getattr__(attr)
Fetches an options value, checking first for explicit settings from the command line (which are direct attributes), then the SConscript file settings, then the default values.

__update(dict, mode)
__update_careful(dict)
Update the option values from an arbitrary dictionary, but only use keys from dict that already have a corresponding attribute in self. Any keys in dict without a corresponding attribute are silently ignored.

__update_loose(dict)
Update the option values from an arbitrary dictionary, using all keys from the dictionary regardless of whether they have a corresponding attribute in self or not.

ensure_value(attr, value)
read_file(filename, mode='careful')
read_module(modname, mode='careful')
set_option(name, value)
Sets an option from an SConscript file.

Raises: UserError – invalid or malformed option (“error in your script”)


SCons.Script.SConsOptions.diskcheck_convert(value)

SCons.Script.SConscript module

This module defines the Python API provided to SConscript files.

SCons.Script.SConscript.BuildDefaultGlobals ()
Create a dictionary containing all the default globals for SConstruct and SConscript files.

SCons.Script.SConscript.Configure (*args, **kw)
class SCons.Script.SConscript.DefaultEnvironmentCall (method_name, subst: int = 0)
   Bases: object
   A class that implements “global function” calls of Environment methods by fetching the specified method from the DefaultEnvironment’s class. Note that this uses an intermediate proxy class instead of calling the DefaultEnvironment method directly so that the proxy can override the subst() method and thereby prevent expansion of construction variables (since from the user’s point of view this was called as a global function, with no associated construction environment).

class SCons.Script.SConscript.Frame (fs, exports, sconscript)
   Bases: object
   A frame on the SConstruct/SConscript call stack

SCons.Script.SConscript.Return (*vars, **kw)
class SCons.Script.SConscript.SConsEnvironment (platform=None, tools=None, toolpath=None, variables=None, parse_flags=None, **kw)
   Bases: Base
   An Environment subclass that contains all of the methods that are particular to the wrapper SCons interface and which aren’t (or shouldn’t be) part of the build engine itself.
   Note that not all of the methods of this class have corresponding global functions, there are some private methods.
   Action (*args, **kw)
   AddMethod (function, name=None) → None

SCons.Script package

282
SCons.Script package

Adds the specified function as a method of this construction environment with the specified name. If the name is omitted, the default name is the name of the function itself.

AddPostAction (files, action)
AddPreAction (files, action)
Alias (target, source=[], action=None, **kw)
AlwaysBuild (*targets)
Append (**kw) → None
  Append values to construction variables in an Environment.
  The variable is created if it is not already present.
AppendENVPath (name, newpath, envname: str = 'ENV', sep=':', delete_existing: bool = False) → None
  Append path elements to the path name in the envname dictionary for this environment. Will only add any particular path once, and will normpath and normcase all paths to help assure this. This can also handle the case where the env variable is a list instead of a string.
  If delete_existing is False, a newpath element already in the path will not be moved to the end (it will be left where it is).
AppendUnique (delete_existing: bool = False, **kw) → None
  Append values to existing construction variables in an Environment, if they’re not already there. If delete_existing is True, removes existing values first, so values move to end.
Builder (**kw)
CacheDir (path, custom_class=None) → None
Clean (targets, files) → None
Clone (tools=[], toolpath=None, parse_flags=None, **kw)
  Return a copy of a construction Environment.
  The copy is like a Python “deep copy”—that is, independent copies are made recursively of each objects—except that a reference is copied when an object is not deep-copyable (like a function). There are no references to any mutable objects in the original Environment.
Command (target, source, action, **kw)
  Builds the supplied target files from the supplied source files using the supplied action. Action may be any type that the Builder constructor will accept for an action.
Configure (*args, **kw)
Decider (function)
Default (*targets) → None
Depends (target, dependency)
  Explicitly specify that ‘target’s depend on ‘dependency’.
Detect (progs)
  Return the first available program from one or more possibilities.
Parameters:  progs (str or list) — one or more command names to check for
Dictionary (*args)
  Return construction variables from an environment.
Parameters:  *args (optional) — variable names to look up
  Returns:  If args omitted, the dictionary of all construction variables. If one arg, the corresponding value is returned. If more than one arg, a list of values is returned.
  Raises:  KeyError — if any of args is not in the construction environment.
Dir (name, *args, **kw)
Dump (key=None, format: str = 'pretty')
  Return construction variables serialized to a string.
Parameters:  • key (optional) — if None, format the whole dict of variables. Else format the value of key
  (Default value = None)
  • format (str, optional) — specify the format to serialize to. “pretty” generates a pretty-printed string, “json” a JSON-formatted string. (Default value = “pretty”)
static EnsurePythonVersion (major, minor) → None
Exit abnormally if the Python version is not late enough.

```python
static EnsureSConsVersion (major, minor, revision: int = 0) → None
```

Exit abnormally if the SCons version is not late enough.

```python
Entry (name, *args, **kw)
Environment (**kw)
Execute (action, *args, **kw)
```

Directly execute an action through an Environment

```python
static Exit (value: int = 0) → None
Export (*vars, **kw) → None
File (name, *args, **kw)
FindFile (file, dirs)
FindInstalledFiles ()

returns the list of all targets of the Install and InstallAs Builder.
```

FindIxes (paths, prefix, suffix)

Search a list of paths for something that matches the prefix and suffix.

```python
Parameters:
  - paths – the list of paths or nodes.
  - prefix – construction variable for the prefix.
  - suffix – construction variable for the suffix.

Returns: the matched path or None
```

```python
FindSourceFiles (node: str = '.') → list
Return a list of all source files.
```

```python
Flatten (sequence)
GetBuildPath (files)
static GetLaunchDir ()
GetOption (name)
Glob (pattern, ondisk: bool = True, source: bool = False, strings: bool = False, exclude=None)
Help (text, append: bool = False, keep_local: bool = False) → None

Update the help text.
The previous help text has text appended to it, except on the first call. On first call, the values of append and keep_local are considered to determine what is appended to.

```python
Parameters:
  - text – string to add to the help text.
  - append – on first call, if true, keep the existing help text (default False).
  - keep_local – on first call, if true and append is also true, keep only the help text from AddOption calls.

Changed in version 4.6.0: The keep_local parameter was added.
```

```python
Ignore (target, dependency)
Ignore a dependency.
Import (*vars)
Literal (string)
Local (*targets)
MergeFlags (args, unique: bool = True) → None
Merge flags into construction variables.
Merges the flags from args into this construction environment. If args is not a dict, it is first converted to one with flags distributed into appropriate construction variables. See ParseFlags().
```

As a side effect, if unique is true, a new object is created for each modified construction variable by the loop at the end. This is silently expected by the Override() parse_flags functionality, which does not want to share the list (or whatever) with the environment being overridden.

```python
Parameters:
  - args – flags to merge
  - unique – merge flags rather than appending (default: True). When merging, path variables are retained from the front, other construction variables from the end.
```

284
NoCache (*targets)
Tags a target so that it will not be cached

NoClean (*targets)
Tags a target so that it will not be cleaned by -c

Override (overrides)
Produce a modified environment whose variables are overridden by the overrides dictionaries. “overrides” is a
dictionary that will override the variables of this environment.
This function is much more efficient than Clone() or creating a new Environment because it doesn’t copy the
construction environment dictionary, it just wraps the underlying construction environment, and doesn’t even create
a wrapper object if there are no overrides.

ParseConfig (command, function=None, unique: bool = True)
Parse the result of running a command to update construction vars.
Use function to parse the output of running command in order to modify the current environment.

Parameters:
- command – a string or a list of strings representing a command and its arguments.
- function – called to process the result of command, which will be passed as args. If
  function is omitted or None, MergeFlags() is used. Takes 3 args
  (env, args, unique)
- unique – whether no duplicate values are allowed (default true)

ParseDepends (filename, must_exist=None, only_one: bool = False)
Parse a mkdep-style file for explicit dependencies. This is completely abusable, and should be unnecessary in the
“normal” case of proper SCons configuration, but it may help make the transition from a Make hierarchy easier for
some people to swallow. It can also be genuinely useful when using a tool that can write a .d file, but for which
writing a scanner would be too complicated.

ParseFlags (*flags) → dict
Return a dict of parsed flags.
Parse flags and return a dict with the flags distributed into the appropriate construction variable names. The flags
are treated as a typical set of command-line flags for a GNU-style toolchain, such as might have been generated
by one of the {foo}-config scripts, and used to populate the entries based on knowledge embedded in this method -
the choices are not expected to be portable to other toolchains.
If one of the flags strings begins with a bang (exclamation mark), it is assumed to be a command and the rest of
the string is executed; the result of that evaluation is then added to the dict.

Platform (platform)

Precious (*targets)
Mark targets as precious: do not delete before building.

Prepend (**kw) → None
Prepend values to construction variables in an Environment.

PrependENVPath (name, newpath, envname: str = ‘ENV’, sep=‘:, delete_existing: bool = True) →
None
Prepend path elements to the path name in the envname dictionary for this environment. Will only add any
particular path once, and will normpath and normcase all paths to help assure this. This can also handle the case
where the env variable is a list instead of a string.
If delete_existing is False, a newpath component already in the path will not be moved to the front (it will be left
where it is).

PrependUnique (delete_existing: bool = False, **kw) → None
Prepend values to existing construction variables in an Environment, if they’re not already there. If delete_existing
is True, removes existing values first, so values move to front.

Pseudo (*targets)
Mark targets as pseudo: must not exist.

PyPackageDir (module_name)

RemoveMethod (function) → None
Removes the specified function’s MethodWrapper from the added_methods list, so we don’t re-bind it when
making a clone.

Replace (**kw) → None
Replace existing construction variables in an Environment with new construction variables and/or values.

ReplaceIxes(path, old_prefix, old_suffix, new_prefix, new_suffix)
Replace old_prefix with new_prefix and old_suffix with new_suffix.

env - Environment used to interpolate variables. path - the path that will be modified. old_prefix - construction variable for the old prefix. old_suffix - construction variable for the old suffix. new_prefix - construction variable for the new prefix. new_suffix - construction variable for the new suffix.

Repository(*dirs, **kw) → None
Specify Repository directories to search.

Requires(target, prerequisite)
Specify that prerequisite must be built before target.
Creates an order-only relationship, not a full dependency. prerequisite must exist before target can be built, but a change to prerequisite does not trigger a rebuild of target.

SConscript(*ls, **kw)
Execute SCons configuration files.

Parameters: *ls (str or list) – configuration file(s) to execute.

Keyword Arguments:
• dirs (list) – execute SConscript in each listed directory.
• name (str) – execute script ‘name’ (used only with ‘dirs’).
• exports (list or dict) – locally export variables the called script(s) can import.
• variant_dir (str) – mirror sources needed for the build in a variant directory to allow building in it.
• duplicate (bool) – physically duplicate sources instead of just adjusting paths of derived files (used only with ‘variant_dir’) (default is True).
• must_exist (bool) – fail if a requested script is missing (default is False, default is deprecated).

Returns: list of variables returned by the called script

Raises: UserError – a script is not found and such exceptions are enabled.

static SConscriptChdir(flag: bool) → None
SConsignFile(name='.sconsign', dbm_module=None) → None
Scanner(*args, **kw)
SetDefault(**kw) → None
SetOption(name, value) → None
SideEffect(side_effect, target)
Tell scons that side_effects are built as side effects of building targets.

Split(arg)
This function converts a string or list into a list of strings or Nodes. This makes things easier for users by allowing files to be specified as a white-space separated list to be split.

The input rules are:

• A single string containing names separated by spaces. These will be split apart at the spaces.
• A single Node instance
• A list containing either strings or Node instances. Any strings in the list are not split at spaces.

In all cases, the function returns a list of Nodes and strings.

Tool(tool, toolpath=None, **kwargs) → Tool
Find and run tool module tool.
Changed in version 4.2: returns the tool module rather than None.

Value(value, built_value=None, name=None)
Return a Value (Python expression) node.
Changed in version 4.0: the name parameter was added.

VariantDir(variant_dir, src_dir, duplicate: int = 1) → None
WhereIs(prog, path=None, pathext=None, reject=None)
Find prog in the path.

canonicalize (path)

Allow Dirs and strings beginning with # for top-relative.

Note this uses the current env's fs (in self).

changed_build (dependency, target, prev_ni, repo_node=None) → bool

changed_content (dependency, target, prev_ni, repo_node=None) → bool

changed_timestamp_match (dependency, target, prev_ni, repo_node=None) → bool

changed_timestamp_newer (dependency, target, prev_ni, repo_node=None) → bool

changed_timestamp_then_content (dependency, target, prev_ni, repo_node=None) → bool

find_toolpath_dir (tp)

get_SConscript_filenames (ls, kw)

Convert the parameters passed to SConscript() calls into a list of files and export variables. If the parameters are invalid, throws SCons.Errors.UserError. Returns a tuple (l, e) where l is a list of SConscript filenames and e is a list of exports.

static _get_major_minor_revision (version_string)

Split a version string into major, minor and (optionally) revision parts.

This is complicated by the fact that a version string can be something like 3.2b1.

gsm ()

_init_special () → None

Initial the dispatch tables for special handling of special construction variables.

update (other) → None

Private method to update an environment's consvar dict directly.

Bypasses the normal checks that occur when users try to set items.

update_onlynew (other) → None

Private method to add new items to an environment's consvar dict.

Only adds items from other whose keys do not already appear in the existing dict; values from other are not used for replacement. Bypasses the normal checks that occur when users try to set items.

arg2nodes (args, node_factory=<class 'SCons.Environment._Null'>, lookup_list=<class 'SCons.Environment._Null'>, **kw)

Converts args to a list of nodes.

Parameters:

• just (args - filename strings or nodes to convert; nodes are) – added to the list without further processing.

• not (node_factory - optional factory to create the nodes; if) – specified, will use this environment's `fs.File method.

• to (lookup_list - optional list of lookup functions to call) – attempt to find the file referenced by each args.

• add. (kw - keyword arguments that represent additional nodes to) –

backtick (command) → str

Emulate command substitution.

Provides behavior conceptually like POSIX Shell notation for running a command in backquotes (backticks) by running command and returning the resulting output string.

This is not really a public API any longer, it is provided for the use of ParseFlags() (which supports it using a syntax of !command) and ParseConfig().

Raises: OSError – if the external command returned non-zero exit status.

get (key, default=None)

Emulates the get() method of dictionaries.

get_CacheDir ()

get_builder (name)

Fetch the builder with the specified name from the environment.

get_factory (factory, default: str = 'File')

Return a factory function for creating Nodes for this construction environment.

get_scanner (skey)

Find the appropriate scanner given a key (usually a file suffix).
SCons.Script package

gvars ()

items ()
    Emulates the items() method of dictionaries.
keys ()
    Emulates the keys() method of dictionaries.
ivars ()

scanner_map_delete (kw=None) → None
    Delete the cached scanner map (if we need to).
setdefault (key, default=None)
    Emulates the setdefault() method of dictionaries.

subst (string, raw: int = 0, target=None, source=None, conv=None, executor: Executor | None = None, overrides: dict | None = None)
    Recursively interpolates construction variables from the Environment into the specified string, returning the
    expanded result. Construction variables are specified by a $ prefix in the string and begin with an initial underscore
    or alphabetic character followed by any number of underscores or alphanumeric characters. The construction
    variable names may be surrounded by curly braces to separate the name from trailing characters.

subst_kw (kw, raw: int = 0, target=None, source=None)

subst_list (string, raw: int = 0, target=None, source=None, conv=None, executor: Executor | None = None, overrides: dict | None = None)
    Calls through to SCons.Subst.scons_subst_list().
    See the documentation for that function.

subst_path (path, target=None, source=None)
    Substitute a path list.
    Turns EntryProxies into Nodes, leaving Nodes (and other objects) as-is.

subst_target_source (string, raw: int = 0, target=None, source=None, conv=None, executor: Executor | None = None, overrides: dict | None = None)
    Recursively interpolates construction variables from the Environment into the specified string, returning the
    expanded result. Construction variables are specified by a $ prefix in the string and begin with an initial underscore
    or alphabetic character followed by any number of underscores or alphanumeric characters. The construction
    variable names may be surrounded by curly braces to separate the name from trailing characters.

validate_CacheDir_class (custom_class=None)
    Validate the passed custom CacheDir class, or if no args are passed, validate the custom CacheDir class from the
    environment.

values ()
    Emulates the values() method of dictionaries.

exception SCons.Script.SConscript.SConscriptReturn
    Bases: Exception
    add_note ()
        Exception.add_note(note) – add a note to the exception
    args
        with_traceback ()
            Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.
SCons.Script.SConscript.SConscript_exception (file=<_io.TextIOWrapper name='<stderr>' mode='w'
    encoding='utf-8'>) → None
    Print an exception stack trace just for the SConscript file(s). This will show users who have Python errors where the
    problem is, without cluttering the output with all of the internal calls leading up to where we exec the SConscript.
SCons.Script.SConscript._SConscript (fs, *files, **kw)
SCons.Script.SConscript.annotate (node)
    Annotate a node with the stack frame describing the SConscript file and line number that created it.
SCons.Script.SConscript.compute_exports (exports)
    Compute a dictionary of exports given one of the parameters to the Export() function or the exports argument to
    SConscript().
SCons.Script.SConscript.get_DefaultEnvironmentProxy ()
SCons.Script.SConscript.get_calling_namespaces ()
    Return the locals and globals for the function that called into this module in the current call stack.
SCons.Script.SConscript.handle_missing_SConscript (f: str, must_exist: bool = True) → None
Take appropriate action on missing file in SConscript() call. Print a warning or raise an exception on missing file, unless missing is explicitly allowed by the must_exist parameter or by a global flag.

**Parameters:**
- **f** – path to missing configuration file
- **must_exist** – if true (the default), fail. If false do nothing, allowing a build to declare it's okay to be missing.

**Raises:** UserError – if must_exist is true or if global SCons.Script._no_missing_sconscript is true.

### Module contents

The main() function used by the scons script.

Architecturally, this is the scons script, and will likely only be called from the external “scons” wrapper. Consequently, anything here should not be, or be considered, part of the build engine. If it's something that we expect other software to want to use, it should go in some other module. If it’s specific to the “scons” script invocation, it goes here.

**SCons.Script.HelpFunction (text, append: bool = False, keep_local: bool = False) → None**

The implementation of the the Help method.

See Help().

Changed in version 4.6.0: The keep_local parameter was added.

**class SCons.Script.TargetList (initlist=None)**

Bases: UserList

- **_abc_impl = <_abc._abc_data object>**
- **_add_Default (list) → None**
- **_clear () → None**
- **_do_nothing (**args, **kw) → None**
- **append (item)**

**S.append(value) – append value to the end of the sequence**

**clear () → None -- remove all items from S**

**copy ()**

**count (value) → integer -- return number of occurrences of value**

**extend (other)**

**S.extend(iterable) – extend sequence by appending elements from the iterable**

**index (value[, start[, stop]]) → integer -- return first index of value.**

**Raises ValueError if the value is not present.**

**Supporting start and stop arguments is optional, but recommended.**

**insert (i, item)**

**S.insert(index, value) – insert value before index**

**pop ([, index]) → item -- remove and return item at index (default last).**

**Raise IndexError if list is empty or index is out of range.**

**remove (item)**

**S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.**

**reverse ()**

**S.reverse() – reverse IN PLACE**

**sort (**args, **kwds)**

**SCons.Script.Variables (files=None, args={})**

**SCons.Script._Add_Arguments (alist) → None**

**SCons.Script._Add_Targets (tlist) → None**

**SCons.Script._Get_Default_Targets (d, fs)**

**SCons.Script._Set_Default_Targets (env, tlist) → None**

**SCons.Script._Set_Default_Targets_Has_Been_Called (d, fs)**

**SCons.Script._Set_Default_Targets_Has_Not_Been_Called (d, fs)**

**SCons.Script.set_missing_sconscript_error (flag: bool = True) → bool**

Set behavior on missing file in SConscript() call.

**Returns:** previous value
SCons.Taskmaster package

Submodules

SCons.Taskmaster.Job module

Serial and Parallel classes to execute build tasks.

The Jobs class provides a higher level interface to start, stop, and wait on jobs.

class SCons.Taskmaster.Job.InterruptState

Bases: object

set () → None

class SCons.Taskmaster.Job.Jobs (num, taskmaster)

Bases: object

An instance of this class initializes N jobs, and provides methods for starting, stopping, and waiting on all N jobs.

_setup_sig_handler () → None

Setup an interrupt handler so that SCons can shutdown cleanly in various conditions:

a. SIGINT: Keyboard interrupt
b. SIGTERM: kill or system shutdown
c. SIGHUP: Controlling shell exiting

We handle all of these cases by stopping the taskmaster. It turns out that it’s very difficult to stop the build process by throwing asynchronously an exception such as KeyboardInterrupt. For example, the python Condition variables (threading.Condition) and queues do not seem to be asynchronous-exception-safe. It would require adding a whole bunch of try/finally block and except KeyboardInterrupt all over the place.

Note also that we have to be careful to handle the case when SCons forks before executing another process. In that case, we want the child to exit immediately.

run (postfunc=<function Jobs.<lambda>>) → None

Run the jobs.

postfunc() will be invoked after the jobs has run. It will be invoked even if the jobs are interrupted by a keyboard interrupt (well, in fact by a signal such as either SIGINT, SIGTERM or SIGHUP). The execution of postfunc() is protected against keyboard interrupts and is guaranteed to run to completion.

were_interrupted ()

Returns whether the jobs were interrupted by a signal.

class SCons.Taskmaster.Job.LegacyParallel (taskmaster, num, stack_size)

Bases: object

This class is used to execute tasks in parallel, and is somewhat less efficient than Serial, but is appropriate for parallel builds.

This class is thread safe.

start ()

Start the job. This will begin pulling tasks from the taskmaster and executing them, and return when there are no more tasks. If a task fails to execute (i.e. execute() raises an exception), then the job will stop.

class SCons.Taskmaster.Job.NewParallel (taskmaster, num, stack_size)

Bases: object

class FakeCondition (lock)

Bases: object

notify ()

notify_all ()

wait ()

class FakeLock

Bases: object

lock ()

unlock ()
class State (value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)
    Bases: Enum
    COMPLETED = 3
    READY = 0
    SEARCHING = 1
    STALLED = 2

    classmethod __contains__(member)
        Return True if member is a member of this enum raises TypeError if member is not an enum member
        note: in 3.12 TypeError will no longer be raised, and True will also be returned if member is the value of a
        member in this enum

    classmethod __getitem__(name)
        Return the member matching name.

    classmethod __iter__()
        Return members in definition order.

    classmethod __len__()
        Return the number of members (no aliases)

class Worker (owner)
    Bases: Thread
    _bootstrap()
    _bootstrap_inner()
    _delete()
        Remove current thread from the dict of currently running threads.
    _initialized = False
    _reset_internal_locks(is_alive)
    _set_ident()
    _set_native_id()
    _set_tstate_lock()
        Set a lock object which will be released by the interpreter when the underlying thread state (see pystate.h) gets
        deleted.
    _stop()
    _wait_for_tstate_lock(block=True, timeout=-1)

    property daemon
        A boolean value indicating whether this thread is a daemon thread.
        This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the
        creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread
default to daemon = False.
        The entire Python program exits when only daemon threads are left.

    getName()
        Return a string used for identification purposes only.
        This method is deprecated, use the name attribute instead.

    property ident
        Thread identifier of this thread or None if it has not been started.
        This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits
        and another thread is created. The identifier is available even after the thread has exited.

    isDaemon()
        Return whether this thread is a daemon.
        This method is deprecated, use the daemon attribute instead.

    is_alive()
        Return whether the thread is alive.
        This method returns True just before the run() method starts until just after the run() method terminates. See also
        the module function enumerate().

    join(timeout=None)
        Wait until the thread terminates.
        This blocks the calling thread until the thread whose join() method is called terminates – either normally or
        through an unhandled exception or until the optional timeout occurs.
When the timeout argument is present and not None, it should be a floating point number specifying a timeout for
the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join()
to decide whether a timeout happened – if the thread is still alive, the join() call timed out.
When the timeout argument is not present or None, the operation will block until the thread terminates.
A thread can be join()ed many times.
join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is
also an error to join() a thread before it has been started and attempts to do so raises the same exception.

**property** name
A string used for identification purposes only.
It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

**property** native_id
Native integral thread ID of this thread, or None if it has not been started.
This is a non-negative integer. See the get_native_id() function. This represents the Thread ID as reported by
the kernel.

run () → None
Method representing the thread’s activity.
You may override this method in a subclass. The standard run() method invokes the callable object passed to
the object’s constructor as the target argument, if any, with sequential and keyword arguments taken from the
args and kwargs arguments, respectively.

setDaemon (daemonic)
Set whether this thread is a daemon.
This method is deprecated, use the .daemon property instead.

setName (name)
Set the name string for this thread.
This method is deprecated, use the name attribute instead.

start ()
Start the thread’s activity.
It must be called at most once per thread object. It arranges for the object’s run() method to be invoked in a
separate thread of control.
This method will raise a RuntimeError if called more than once on the same thread object.

SCons.Taskmaster package

292
class SCons.Taskmaster.Job.Worker(requestQueue, resultsQueue, interrupted)

Bases: Thread

A worker thread waits on a task to be posted to its request queue, dequeues the task, executes it, and posts a tuple including the task and a boolean indicating whether the task executed successfully.

_bootstrap()
_bootstrap_inner()
_delete()

Remove current thread from the dict of currently running threads.

_initialized = False
_reset_internal_locks(is_alive)
_set_ident()
_set_native_id()
_set_tstate_lock()

Set a lock object which will be released by the interpreter when the underlying thread state (see pystate.h) gets deleted.

_stop()
_wait_for_tstate_lock(block=True, timeout=-1)

property daemon

A boolean value indicating whether this thread is a daemon thread.
This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread: the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

getName()

Return a string used for identification purposes only.
This method is deprecated, use the name attribute instead.

property ident

Thread identifier of this thread or None if it has not been started.
This is a nonzero integer. See the get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isDaemon()

Return whether this thread is a daemon.
This method is deprecated, use the daemon attribute instead.

is_alive()

Return whether the thread is alive.
This method returns True just before the run() method starts until just after the run() method terminates. See also the module function enumerate().

join(timeout=None)

Wait until the thread terminates.
This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.
When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.
When the timeout argument is not present or None, the operation will block until the thread terminates.
A thread can be joined many times.
join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

property name

A string used for identification purposes only.
It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

property native_id

Native integral thread ID of this thread, or None if it has not been started.
This is a non-negative integer. See the get_native_id() function. This represents the Thread ID as reported by the kernel.

run()
Method representing the thread’s activity.
You may override this method in a subclass. The standard run() method invokes the callable object passed to the
object’s constructor as the target argument, if any, with sequential and keyword arguments taken from the args and
kwargs arguments, respectively.

setDaemon(daemonic)
Set whether this thread is a daemon.
This method is deprecated, use the .daemon property instead.

setName(name)
Set the name string for this thread.
This method is deprecated, use the name attribute instead.

start()
Start the thread’s activity.
It must be called at most once per thread object. It arranges for the object’s run() method to be invoked in a
separate thread of control.
This method will raise a RuntimeError if called more than once on the same thread object.

Module contents

Generic Taskmaster module for the SCons build engine.

This module contains the primary interface(s) between a wrapping user interface and the SCons build engine. There
are two key classes here:

Taskmaster
This is the main engine for walking the dependency graph and calling things to decide what does or doesn’t need
to be built.

Task
This is the base class for allowing a wrapping interface to decide what does or doesn’t actually need to be done.
The intention is for a wrapping interface to subclass this as appropriate for different types of behavior it may need.

The canonical example is the SCons native Python interface, which has Task subclasses that handle its specific
behavior, like printing “foo’ is up to date” when a top-level target doesn’t need to be built, and handling the -c
option by removing targets as its “build” action. There is also a separate subclass for suppressing this output when
the -q option is used.

The Taskmaster instantiates a Task object for each (set of) target(s) that it decides need to be evaluated and/or
built.

class SCons.Taskmaster.AlwaysTask(tm, targets, top, node)
Bases: Task

LOGGER = None
_abc_impl = <_abc._abc_data object>

_exception_raise() → None
Raises a pending exception that was recorded while getting a Task ready for execution.

_no_exception_to_raise() → None

display(message) → None
Hook to allow the calling interface to display a message.
This hook gets called as part of preparing a task for execution (that is, a Node to be built). As part of figuring out
what Node should be built next, the actual target list may be altered, along with a message describing the
alteration. The calling interface can subclass Task and provide a concrete implementation of this method to see
those messages.

exc_clear() → None
Clears any recorded exception.
This also changes the “exception_raise” attribute to point to the appropriate do-nothing method.

exc_info() → None
Returns info about a recorded exception.

except_set(exception=None) → None
Records an exception to be raised at the appropriate time.
This also changes the “exception_raise” attribute to point to the method that will, in fact
execute ()
    Called to execute the task.
    This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe
    stuff in prepare(), executed() or failed().
executed () → None
    Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s
    callback methods.
    This may have been a do-nothing operation (to preserve build order), so we must check the node’s state before
deciding whether it was “built”, in which case we call the appropriate Node method. In any event, we always call
“visited()”, which will handle any post-visit actions that must take place regardless of whether or not the target was
an actual built target or a source Node.
executed_with_callbacks () → None
    Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s
    callback methods.
    This may have been a do-nothing operation (to preserve build order), so we must check the node’s state before
deciding whether it was “built”, in which case we call the appropriate Node method. In any event, we always call
“visited()”, which will handle any post-visit actions that must take place regardless of whether or not the target was
an actual built target or a source Node.
executed_without_callbacks () → None
    Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s
    callback methods.
fail_continue () → None
    Explicit continue-the-build failure.
    This sets failure status on the target nodes and all of their dependent parent nodes.
    Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on
up-to-date nodes when using Configure().
fail_stop () → None
    Explicit stop-the-build failure.
    This sets failure status on the target nodes and all of their dependent parent nodes.
    Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on
up-to-date nodes when using Configure().
failed () → None
    Default action when a task fails: stop the build.
    Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on
up-to-date nodes when using Configure().
get_target ()
    Fetch the target being built or updated by this task.
make_ready ()
    Marks all targets in a task ready for execution if any target is not current.
    This is the default behavior for building only what’s necessary.
make_ready_all () → None
    Marks all targets in a task ready for execution.
    This is used when the interface needs every target Node to be visited—the canonical example being the “scons -c”
option.
make_ready_current ()
    Marks all targets in a task ready for execution if any target is not current.
    This is the default behavior for building only what’s necessary.
needs_execute () → bool
    Always returns True (indicating this Task should always be executed).
    Subclasses that need this behavior (as opposed to the default of only executing Nodes that are out of date w.r.t.
their dependencies) can use this as follows:

    class MyTaskSubclass(SCons.Taskmaster.Task):
        needs_execute = SCons.Taskmaster.AlwaysTask.needs_execute
postprocess () → None
    Post-processes a task after it’s been executed.
This examines all the targets just built (or not, we don’t care if the build was successful, or even if there was no build because everything was up-to-date) to see if they have any waiting parent Nodes, or Nodes waiting on a common side effect, that can be put back on the candidates list.

prepare() → None
Called just before the task is executed.
This is mainly intended to give the target Nodes a chance to unlink underlying files and make all necessary directories before the Action is actually called to build the targets.

trace_message(node, description=’node’) → None

class SCons.Taskmaster.OutOfDateTask(tm, targets, top, node)
Bases: Task
LOGGER = None
_abc_impl = <_abc._abc_data object>

_exception_raise()  
 Raises a pending exception that was recorded while getting a Task ready for execution.

_no_exception_to_raise() → None

display(message) → None
Hook to allow the calling interface to display a message.
This hook gets called as part of preparing a task for execution (that is, a Node to be built). As part of figuring out what Node should be built next, the actual target list may be altered, along with a message describing the alteration. The calling interface can subclass Task and provide a concrete implementation of this method to see those messages.

exc_clear() → None
Clears any recorded exception. 
This also changes the “exception_raise” attribute to point to the appropriate do-nothing method.

exc_info()  
Returns info about a recorded exception.

exception_set(exception=None) → None
Records an exception to be raised at the appropriate time.
This also changes the “exception_raise” attribute to point to the method that will, in fact

execute()  
Called to execute the task.
This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in prepare(), executed() or failed().

executed() → None
Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s callback methods.
This may have been a do-nothing operation (to preserve build order), so we must check the node’s state before deciding whether it was “built”, in which case we call the appropriate Node method. In any event, we always call “visited()”, which will handle any post-visit actions that must take place regardless of whether or not the target was an actual built target or a source Node.

done_with_callbacks() → None
Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s callback methods.

executed_without_callbacks() → None
Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s callback methods.

fail_continue() → None
Explicit continue-the-build failure.
This sets failure status on the target nodes and all of their dependent parent nodes.

Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

fail_stop() → None
Explicit stop-the-build failure.
This sets failure status on the target nodes and all of their dependent parent nodes.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on
up-to-date nodes when using Configure().

failed () → None
Default action when a task fails: stop the build.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on
up-to-date nodes when using Configure().

get_target ()
Fetch the target being built or updated by this task.

make_ready ()
Marks all targets in a task ready for execution if any target is not current.
This is the default behavior for building only what’s necessary.

make_ready_all () → None
Marks all targets in a task ready for execution.
This is used when the interface needs every target Node to be visited— the canonical example being the “scons -c”
option.

make_ready_current ()
Marks all targets in a task ready for execution if any target is not current.
This is the default behavior for building only what’s necessary.

needs_execute ()
Returns True (indicating this Task should be executed) if this Task’s target state indicates it needs executing,
which has already been determined by an earlier up-to-date check.

postprocess () → None
Post-processes a task after it’s been executed.
This examines all the targets just built (or not, we don’t care if the build was successful, or even if there was no
build because everything was up-to-date) to see if they have any waiting parent Nodes, or Nodes waiting on a
common side effect, that can be put back on the candidates list.

prepare () → None
Called just before the task is executed.
This is mainly intended to give the target Nodes a chance to unlink underlying files and make all necessary
directories before the Action is actually called to build the targets.

trace_message (node, description: str = 'node') → None

class SCons.Taskmaster.Stats
Bases: object
A simple class for holding statistics about the disposition of a Node by the Taskmaster. If we’re collecting statistics,
each Node processed by the Taskmaster gets one of these attached, in which case the Taskmaster records its
decision each time it processes the Node. (Ideally, that’s just once per Node.)

class SCons.Taskmaster.Task (tm, targets, top, node)
Bases: ABC
SCons build engine abstract task class.
This controls the interaction of the actual building of node and the rest of the engine.
This is expected to handle all of the normally-customizable aspects of controlling a build, so any given application
should be able to do what it wants by sub-classing this class and overriding methods as appropriate. If an application
needs to customize something by sub-classing Taskmaster (or some other build engine class), we should first try to
migrate that functionality into this class.
Note that it’s generally a good idea for sub-classes to call these methods explicitly to update state, etc., rather than
roll their own interaction with Taskmaster from scratch.

LOGGER = None
_abc_impl = <_abc._abc_data object>
_exception_raise ()
Raises a pending exception that was recorded while getting a Task ready for execution.
_no_exception_to_raise () → None
display (message) → None
Hook to allow the calling interface to display a message.
This hook gets called as part of preparing a task for execution (that is, a Node to be built). As part of figuring out what Node should be built next, the actual target list may be altered, along with a message describing the alteration. The calling interface can subclass Task and provide a concrete implementation of this method to see those messages.

`exc_clear () → None`
Cleans any recorded exception.
This also changes the “exception_raise” attribute to point to the appropriate do-nothing method.

`exc_info ()`
Returns info about a recorded exception.

`exception_set (exception=None) → None`
Records an exception to be raised at the appropriate time.
This also changes the “exception_raise” attribute to point to the method that will, in fact

`execute ()`
Called to execute the task.
This method is called from multiple threads in a parallel build, so only do thread safe stuff here. Do thread unsafe stuff in prepare(), executed() or failed().

`executed () → None`
Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s callback methods.
This may have been a do-nothing operation (to preserve build order), so we must check the node’s state before deciding whether it was “built”, in which case we call the appropriate Node method. In any event, we always call “visited()”, which will handle any post-visit actions that must take place regardless of whether or not the target was an actual built target or a source Node.

`executed_with_callbacks () → None`
Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s callback methods.
This may have been a do-nothing operation (to preserve build order), so we must check the node’s state before deciding whether it was “built”, in which case we call the appropriate Node method. In any event, we always call “visited()”, which will handle any post-visit actions that must take place regardless of whether or not the target was an actual built target or a source Node.

`executed_without_callbacks () → None`
Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s callback methods.

`fail_continue () → None`
Explicit continue-the-build failure.
This sets failure status on the target nodes and all of their dependent parent nodes.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

`fail_stop () → None`
Explicit stop-the-build failure.
This sets failure status on the target nodes and all of their dependent parent nodes.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

`failed () → None`
Default action when a task fails: stop the build.
Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

`get_target ()`
Fetch the target being built or updated by this task.

`make_ready ()`
Marks all targets in a task ready for execution if any target is not current.
This is the default behavior for building only what’s necessary.

`make_ready_all () → None`
Marks all targets in a task ready for execution.
This is used when the interface needs every target Node to be visited—the canonical example being the “scons -c” option.
make_ready_current ()
Marks all targets in a task ready for execution if any target is not current.
This is the default behavior for building only what's necessary.
abstract needs_execute ()
postprocess () \rightarrow \text{None}
Post-processes a task after it's been executed.
This examines all the targets just built (or not, we don't care if the build was successful, or even if there was no build because everything was up-to-date) to see if they have any waiting parent Nodes, or Nodes waiting on a common side effect, that can be put back on the candidates list.
prepare () \rightarrow \text{None}
Called just before the task is executed.
This is mainly intended to give the target Nodes a chance to unlink underlying files and make all necessary directories before the Action is actually called to build the targets.
trace_message (node, description: str = 'node') \rightarrow \text{None}

class SCons.Taskmaster.Taskmaster (targets=[], tasker=None, order=None, trace=None)
Bases: object
The Taskmaster for walking the dependency DAG.

_find_next_ready_node ()
Finds the next node that is ready to be built.
This is the main guts of the DAG walk. We loop through the list of candidates, looking for something that has no un-built children (i.e., that is a leaf Node or has dependencies that are all leaf Nodes or up-to-date). Candidate Nodes are re-scanned (both the target Node itself and its sources, which are always scanned in the context of a given target) to discover implicit dependencies. A Node that must wait for some children to be built will be put back on the candidates list after the children have finished building. A Node that has been put back on the candidates list in this way may have itself (or its sources) re-scanned, in order to handle generated header files (e.g.) and the implicit dependencies therein.
Note that this method does not do any signature calculation or up-to-date check itself. All of that is handled by the Task class. This is purely concerned with the dependency graph walk.

_validate_pending_children () \rightarrow \text{None}
Validate the content of the pending_children set. Assert if an internal error is found.
This function is used strictly for debugging the taskmaster by checking that no invariants are violated. It is not used in normal operation.
The pending_children set is used to detect cycles in the dependency graph. We call a “pending child” a child that is found in the “pending” state when checking the dependencies of its parent node.
A pending child can occur when the Taskmaster completes a loop through a cycle. For example, let’s imagine a graph made of three nodes (A, B and C) making a cycle. The evaluation starts at node A. The Taskmaster first considers whether node A’s child B is up-to-date. Then, recursively, node B needs to check whether node C is up-to-date. This leaves us with a dependency graph looking like:

```
Next candidate
          ^
         /|
+---------> Node D (NoState) -------+
         /
```

Now, when the Taskmaster examines the Node C’s child Node A, it finds that Node A is in the “pending” state. Therefore, Node A is a pending child of node C.
Pending children indicate that the Taskmaster has potentially loop back through a cycle. We say potentially because it could also occur when a DAG is evaluated in parallel. For example, consider the following graph:

```
Node A (Pending) --> Node B(Pending) --> Node C (Pending) --> ...
|                     |                     |
|                     |                     |
+---------------------+---------------------+
         /                 /                   /
Next candidate /
The Taskmaster first evaluates the nodes A, B, and C and starts building some children of node C. Assuming, that the maximum parallel level has not been reached, the Taskmaster will examine Node D. It will find that Node C is a pending child of Node D.

In summary, evaluating a graph with a cycle will always involve a pending child at one point. A pending child might indicate either a cycle or a diamond-shaped DAG. Only a fraction of the nodes ends-up being a “pending child” of another node. This keeps the pending children set small in practice.

We can differentiate between the two cases if we wait until the end of the build. At this point, all the pending children nodes due to a diamond-shaped DAG will have been properly built (or will have failed to build). But, the pending children involved in a cycle will still be in the pending state.

The taskmaster removes nodes from the pending_children set as soon as a pending_children node moves out of the pending state. This also helps to keep the pending_children set small.

cleanup()
- Check for dependency cycles.
configure_trace (trace=None) → None
- This handles the command line option --taskmastertrace= It can be: - : output to stdout <filename> : output to a file False/None : Do not trace
find_next_candidate()
- Returns the next candidate Node for (potential) evaluation.
- The candidate list (really a stack) initially consists of all of the top-level (command line) targets provided when the Taskmaster was initialized. While we walk the DAG, visiting Nodes, all the children that haven't finished processing get pushed on to the candidate list. Each child can then be popped and examined in turn for whether their children are all up-to-date, in which case a Task will be created for their actual evaluation and potential building.
- Here is where we also allow candidate Nodes to alter the list of Nodes that should be examined. This is used, for example, when invoking SCons in a source directory. A source directory Node can return its corresponding build directory Node, essentially saying, “Hey, you really need to build this thing over here instead.”

next_task()
- Returns the next task to be executed.
- This simply asks for the next Node to be evaluated, and then wraps it in the specific Task subclass with which we were initialized.

no_next_candidate()
- Stops Taskmaster processing by not returning a next candidate.
- Note that we have to clean-up the Taskmaster candidate list because the cycle detection depends on the fact all nodes have been processed somehow.

stop() → None
- Stops the current build completely.

tm_trace_node (node) → str
- Will perform clean-up about nodes that will never be built. Invokes a user defined function on all of these nodes (including all of their parents).

SCons.Taskmaster.dump_stats () → None
SCons.Taskmaster.find_cycle (stack, visited)

SCons.Tool package

Module contents
SCons tool selection.
- Looks for modules that define a callable object that can modify a construction environment as appropriate for a given tool (or tool chain).

Note that because this subsystem just selects a callable that can modify a construction environment, it's possible for people to define their own “tool specification” in an arbitrary callable function. No one needs to use or tie in to this subsystem in order to roll their own tool specifications.

SCons.Tool.CreateJarBuilder (env)
- The Jar builder expects a list of class files which it can package into a jar file.
The jar tool provides an interface for passing other types of Java files such as .java, directories or SWIG interfaces and will build them to class files in which it can package into the jar.

SCons.Tool.CreateJavaClassDirBuilder (env)
SCons.Tool.CreateJavaClassFileBuilder (env)
SCons.Tool.CreateJavaFileBuilder (env)
SCons.Tool.CreateJavaHBuilder (env)
SCons.Tool.FindAllTools (tools, env)
SCons.Tool.FindTool (tools, env)
SCons.Tool.Initializers (env) → None

class SCons.Tool.Tool (name, toolpath=None, **kwargs)
Bases: object
_tool_module ()
Try to load a tool module.
This will hunt in the toolpath for both a Python file (toolname.py) and a Python module (toolname directory), then try the regular import machinery, then fallback to try a zipfile.

class SCons.Tool.ToolInitializer (env, tools, names)
Bases: object
A class for delayed initialization of Tools modules.
Instances of this class associate a list of Tool modules with a list of Builder method names that will be added by those Tool modules. As part of instantiating this object for a particular construction environment, we also add the appropriate ToolInitializerMethod objects for the various Builder methods that we want to use to delay Tool searches until necessary.
apply_tools (env) → None
Searches the list of associated Tool modules for one that exists, and applies that to the construction environment.
remove_methods (env) → None
Removes the methods that were added by the tool initialization so we no longer copy and re-bind them when the construction environment gets cloned.

class SCons.Tool.ToolInitializerMethod (name, initializer)
Bases: object
This is added to a construction environment in place of a method(s) normally called for a Builder (env.Object, env.StaticObject, etc.). When called, it has its associated ToolInitializer object search the specified list of tools and apply the first one that exists to the construction environment. It then calls whatever builder was (presumably) added to the construction environment in place of this particular instance.
__call__ (env, *args, **kw)
get_builder (env)
Returns the appropriate real Builder for this method name after having the associated ToolInitializer object apply the appropriate Tool module.

SCons.Tool.createCFileBuilders (env)
This is a utility function that creates the CFile/CXXFile Builders in an Environment if they are not there already. If they are there already, we return the existing ones.
This is a separate function because soooo many Tools use this functionality.
The return is a 2-tuple of (CFile, CXXFile)

SCons.Tool.createLoadableModuleBuilder (env, loadable_module_suffix: str = '_LDMODULESUFFIX')
This is a utility function that creates the LoadableModule Builder in an Environment if it is not there already. If it is already there, we return the existing one.

Parameters: loadable_module_suffix – The suffix specified for the loadable module builder

SCons.Tool.createObjBuilders (env)
This is a utility function that creates the StaticObject and SharedObject Builders in an Environment if they are not there already. If they are there already, we return the existing ones.
This is a separate function because soooo many Tools use this functionality.
The return is a 2-tuple of (StaticObject, SharedObject)

SCons.Tool.createProgBuilder (env)
This is a utility function that creates the Program Builder in an Environment if it is not there already. If it is already there, we return the existing one.
SCons.Tool.createSharedLibBuilder (env, shlib_suffix: str = '$_SHLIBSUFFIX')
This is a utility function that creates the SharedLibrary Builder in an Environment if it is not there already.
If it is already there, we return the existing one.

**Parameters:**

- **shlib_suffix** – The suffix specified for the shared library builder

SCons.Tool.createStaticLibBuilder (env)
This is a utility function that creates the StaticLibrary Builder in an Environment if it is not there already.
If it is already there, we return the existing one.

SCons.Tool.find_program_path (env, key_program, default_paths=None, add_path: bool = False) → str | None
Find the location of a tool using various means.
Mainly for windows where tools aren’t all installed in /usr/bin, etc.

**Parameters:**

- **env** – Current Construction Environment.
- **key_program** – Tool to locate.
- **default_paths** – List of additional paths this tool might be found in.
- **add_path** – If true, add path found if it was from default_paths.

SCons.Tool.tool_list (platform, env)

SCons.Util package

Submodules

SCons utility functions

This package contains routines for use by other parts of SCons. Candidates for inclusion here are routines that do not need other parts of SCons (other than Util), and have a reasonable chance of being useful in multiple places, rather then being topical only to one module/package.

**class** SCons.Util.CLVar (initlist=None)
Bases: UserList
A container for command-line construction variables.

Forces the use of a list of strings intended as command-line arguments. Like collections.UserList, but the argument passed to the initializer will be processed by the Split() function, which includes special handling for string types: they will be split into a list of words, not coerced directly to a list. The same happens if a string is added to a CLVar, which allows doing the right thing with both Append()/Prepend() methods, as well as with pure Python addition, regardless of whether adding a list or a string to a construction variable.

Side effect: spaces will be stripped from individual string arguments. If you need spaces preserved, pass strings containing spaces inside a list argument.

```python
globals()['u'] = UserList("--some --opts and args")
globals()['len(u)'] = repr(u))
globals()['c'] = CLVar("--some --opts and args")
globals()['repr(c)']
globals()['len(c)']
globals()['c += "strips spaces "']
globals()['len(c)']
globals()['c += "does not split or strip"]'
```

_SCons.Util.CLVar_.impl
_SCons.Util.CLVar_.append (item)
S.append(value) – append value to the end of the sequence
SCons.Util package

clear () → None -- remove all items from S

copy ()

count (value) → integer -- return number of occurrences of value

extend (other)

S.extend(iterable) – extend sequence by appending elements from the iterable

index (value[, start[, stop]]) → integer -- return first index of value.

  Raises ValueError if the value is not present.

  Supporting start and stop arguments is optional, but recommended.

insert (i, item)

S.insert(index, value) – insert value before index

pop ([, index]) → item -- remove and return item at index (default last).

  Raise IndexError if list is empty or index is out of range.

remove (item)

S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

reverse ()

S.reverse() – reverse IN PLACE

sort (*args, **kwds)

class SCons.Util.Delegate (attribute)

  Bases: object

  A Python Descriptor class that delegates attribute fetches to an underlying wrapped subject of a Proxy. Typical use:

```python
    class Foo(Proxy):
        __str__ = Delegate('__str__')
```

class SCons.Util.DispatchingFormatter (formatters, default_formatter)

  Bases: Formatter

  Logging formatter which dispatches to various formatters.

  converter ()

  localtime([seconds]) -> (tm_year,tm_mon,tm_mday,tm_hour,tm_min,

  tm_sec,tm_wday,tm_yday,tm_isdst)

  Convert seconds since the Epoch to a time tuple expressing local time. When `seconds` is not passed in, convert
  the current time instead.

default_msec_format = '%s,%03d'
default_time_format = '%Y-%m-%d %H:%M:%S'

format (record)

  Format the specified record as text.

  The record’s attribute dictionary is used as the operand to a string formatting operation which yields the returned
  string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the
  record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call
to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using
formatException() and appended to the message.

formatException (ei)

  Format and return the specified exception information as a string.

  This default implementation just uses traceback.print_exception()

formatMessage (record)

formatStack (stack_info)

  This method is provided as an extension point for specialized formatting of stack information.

  The input data is a string as returned from a call to traceback.print_stack(), but with the last trailing newline
  removed.

  The base implementation just returns the value passed in.

formatTime (record, datefmt=None)

  Return the creation time of the specified LogRecord as formatted text.

  This method should be called from format() by a formatter which wants to make use of a formatted time. This
  method can be overridden in formatters to provide for any specific requirement, but the basic behaviour is as
  follows: if datefmt (a string) is specified, it is used with time.strftime() to format the creation time of the record.
Otherwise, an ISO8601-like (or RFC 3339-like) format is used. The resulting string is returned. This function uses a user-configurable function to convert the creation time to a tuple. By default, `time.localtime()` is used; to change this for a particular formatter instance, set the ‘converter’ attribute to a function with the same signature as `time.localtime()` or `time.gmtime()`. To change it for all formatters, for example if you want all logging times to be shown in GMT, set the ‘converter’ attribute in the Formatter class.

```python
usesTime ()
```

Check if the format uses the creation time of the record.

```python
class SCons.Util.DisplayEngine
    Bases: object
    A callable class used to display SCons messages.
    print_it = True
    set_mode (mode) → None
SCons.Util.IDX (n) → bool
    Generate in index into strings from the tree legends.
    These are always a choice between two, so bool works fine.
class SCons.Util.LogicalLines (fileobj)
    Bases: object
    Wrapper class for the logical_lines() function.
    Allows us to read all “logical” lines at once from a given file object.
    readlines ()
class SCons.Util.NodeList (initiallist=None)
    Bases: UserList
    A list of Nodes with special attribute retrieval.
    Unlike an ordinary list, access to a member’s attribute returns a NodeList containing the same attribute for each member. Although this can hold any object, it is intended for use when processing Nodes, where fetching an attribute of each member is very common, for example getting the content signature of each node. The term “attribute” here includes the string representation.
```

```python
>>> someList = NodeList(['  foo  ', '  bar  '])
>>> someList.strip()
['foo', 'bar']
```

`__getattr__ (name) → NodeList`

Returns a NodeList of `name` from each member.

`__getitem__ (index)`

Returns one item, forces a NodeList if `index` is a slice.

`_abc_impl` = <_abc._abc_data object>

```python
append (item)
    S.append(value) – append value to the end of the sequence
```

```python
clear () → None -- remove all items from S
```

```python
copy ()
```

```python
count (value) → integer -- return number of occurrences of value
```

```python
extend (other)
    S.extend(iterable) – extend sequence by appending elements from the iterable
```

```python
index (value[, start[, stop]]) → integer -- return first index of value.
    Raises ValueError if the value is not present.
    Supporting start and stop arguments is optional, but recommended.
```

```python
insert (i, item)
    S.insert(index, value) – insert value before index
```

```python
pop ([, index]) → item -- remove and return item at index (default last).
    Raise IndexError if list is empty or index is out of range.
```

```python
remove (item)
    S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.
```

```python
reverse ()
    S.reverse() – reverse IN PLACE
```

```python
sort (*args, **kwds)
```

304
A simple generic Proxy class, forwarding all calls to subject.
This means you can take an object, let’s call it ‘obj_a’, and wrap it in this Proxy class, with a statement like this:

```python
proxy_obj = Proxy(obj_a)
```

Then, if in the future, you do something like this:

```python
x = proxy_obj.var1
```

since the Proxy class does not have a var1 attribute (but presumably obj_a does), the request actually is equivalent to saying:

```python
x = obj_a.var1
```

Inherit from this class to create a Proxy.
With Python 3.5+ this does not work transparently for Proxy subclasses that use special dunder method names, because those names are now bound to the class, not the individual instances. You now need to know in advance which special method names you want to pass on to the underlying Proxy object, and specifically delegate their calls like this:

```python
class Foo(Proxy):
    __str__ = Delegate('__str__')
```

```python
__getattr__(name)
```

Retrieve an attribute from the wrapped object.

**Raises:** `AttributeError` – if attribute `name` doesn’t exist.

```python
get()
```

Retrieve the entire wrapped object

SCons.Util.RegError

alias of ‘_NoError’

SCons.Util.RegGetValue(root, key)

SCons.Util.RegOpenKeyEx(root, key)

class SCons.Util.Selector

Bases: dict

A callable dict for file suffix lookup.
Often used to associate actions or emitters with file types.
Depends on insertion order being preserved so that get_suffix() calls always return the first suffix added.
clear() → None. Remove all items from D.
copy() → a shallow copy of D
fromkeys(value=None, /)
Create a new dictionary with keys from iterable and values set to value.
get(key, default=None, /)
Return the value for key if key is in the dictionary, else default.
items() → a set-like object providing a view on D’s items
keys() → a set-like object providing a view on D’s keys
pop(k[, d]) → v, remove specified key and return the corresponding value.
If the key is not found, return the default if given; otherwise, raise a KeyError.
popitem()
Remove and return a (key, value) pair as a 2-tuple.
Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.
setdefault(key, default=None, /)
Insert key with a value of default if key is not in the dictionary.
Return the value for key if key is in the dictionary, else default.
update ([, ] **F) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys()
method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values () → an object providing a view on D's values
SCons.Util.Split (arg) → list
Returns a list of file names or other objects.
If arg is a string, it will be split on whitespace within the string. If arg is already a list, the list will be returned
untouched. If arg is any other type of object, it will be returned in a single-item list.

```python
>>> print(Split(" this is a string "))
[ 'this', 'is', 'a', 'string' ]
>>> print(Split(['"stringlist", " preserving ", " spaces "]))
[ 'stringlist', ' preserving ', ' spaces ' ]
```

class SCons.Util.Unbuffered (file)
Bases: object
A proxy that wraps a file object, flushing after every write.
Delegates everything else to the wrapped object.
write (arg) → None
writelines (arg) → None
class SCons.Util.UniqueList (initlist=None)
Bases: UserList
A list which maintains uniqueness.
Uniquing is lazy: rather than being enforced on list changes, it is fixed up on access by those methods which need to
act on a unique list to be correct. That means things like membership tests don’t have to eat the uniquing time.
__make_unique () → None
__abc_impl = < _abc_ _abc_data object >
append (item) → None
S.append(value) – append value to the end of the sequence
clear () → None -- remove all items from S
copy ()
count (value) → integer -- return number of occurrences of value
extend (other) → None
S.extend(iterable) – extend sequence by appending elements from the iterable
index (value[, start[, stop]]) → integer -- return first index of value.
Raises ValueError if the value is not present.
Supporting start and stop arguments is optional, but recommended.
insert (i, item) → None
S.insert(index, value) – insert value before index
pop ([, index]) → item -- remove and return item at index (default last).
Raise IndexError if the list is empty or index is out of range.
remove (item)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.
reverse () → None
S.reverse() – reverse IN PLACE
sort (*args, **kwd)
SCons.Util.WhereIs (file, path=None, pathext=None, reject=None) → str | None
Return the path to an executable that matches file.
Searches the given path for file, considering any filename extensions in pathext (on the Windows platform only), and
returns the full path to the matching command of the first match, or None if there are no matches. Will not select any
path name or names in the optional reject list.
If path is None (the default), os.environ[PATH] is used. On Windows, If pathext is None (the default),
oS.environ[PATHEXT] is used.
The construction environment method of the same name wraps a call to this function by filling in path from the execution environment if it is None (and for pathext on Windows, if necessary), so if called from there, this function will not backfill from os.environ.

**Note**
Finding things in os.environ may answer the question “does file exist on the system”, but not the question “can SCons use that executable”, unless the path element that yields the match is also in the the Execution Environment (e.g. env['ENV']['PATH']). Since this utility function has no environment reference, it cannot make that determination.
In Cygwin, this converts from a Cygwin path to a Windows path, without regard to whether path refers to an existing file system object. For other platforms, path is unchanged.

SCons.Util.logical_lines (physical_lines, joiner=<built-in method join of str object>)
SCons.Util.make_path_relative (path) \rightarrow str

Converts an absolute path name to a relative pathname.

SCons.Util.print_time ()

Hack to return a value from Main if can't import Main.

SCons.Util.print_tree (root, child_func, prune: bool = False, showtags: int = 0, margin: List[bool] = [False], visited: dict | None = None, lastChild: bool = False, singleLineDraw: bool = False) \rightarrow None

Print a tree of nodes.
This is like func:render_tree, except it prints lines directly instead of creating a string representation in memory, so that huge trees can be handled.

Parameters:
- root – the root node of the tree
- child_func – the function called to get the children of a node
- prune – don’t visit the same node twice
- showtags – print status information to the left of each node line The default is false (value 0). A value of 2 will also print a legend for the margin tags.
- margin – the format of the left margin to use for children of root. Each entry represents a column, where a true value will display a vertical bar and a false one a blank.
- visited – a dictionary of visited nodes in the current branch if prune is false, or in the whole tree if prune is true.
- lastChild – this is the last leaf of a branch
- singleLineDraw – use line-drawing characters rather than ASCII.

SCons.Util.render_tree (root, child_func, prune: bool = False, margin: List[bool] = [False], visited: dict | None = None) \rightarrow str

Render a tree of nodes into an ASCII tree view.

Parameters:
- root – the root node of the tree
- child_func – the function called to get the children of a node
- prune – don’t visit the same node twice
- margin – the format of the left margin to use for children of root. Each entry represents a column where a true value will display a vertical bar and a false one a blank.
- visited – a dictionary of visited nodes in the current branch if prune is false, or in the whole tree if prune is true.

SCons.Util.rightmost_separator (path, sep)
SCons.Util.sanitize_shell_env (execution_env: dict) \rightarrow dict

Sanitize all values in execution_env
The execution environment (typically comes from env['ENV']) is propagated to the shell, and may need to be cleaned first.

Parameters:
- execution_env – The shell environment variables to be propagated
- shell. (to the spawned) –

Returns: sanitized dictionary of env variables (similar to what you’d get from os.environ)

SCons.Util.semi_deepcopy (obj)
SCons.Util.semi_deepcopy_dict (obj, exclude=None) \rightarrow dict
SCons.Util.silent_intern (__string: Any) \rightarrow str
Intern a string without failing.
Perform sys.intern on the passed argument and return the result. If the input is ineligible for interning the original argument is returned and no exception is thrown.

SCons.Util.splitext(path) → tuple
Split path into a (root, ext) pair.
Same as os.path.splitext but faster.

SCons.Util.unique(seq)
Return a list of the elements in seq without duplicates, ignoring order.
For best speed, all sequence elements should be hashable. Then unique() will usually work in linear time.
If not possible, the sequence elements should enjoy a total ordering, and if list(s).sort() doesn’t raise TypeError it is assumed that they do enjoy a total ordering. Then unique() will usually work in O(N*log2(N)) time.
If that’s not possible either, the sequence elements must support equality-testing. Then unique() will usually work in quadratic time.

>>> mylist = unique([1, 2, 3, 1, 2, 3])
>>> print(sorted(mylist))
[1, 2, 3]

>>> mylist = unique("abcabc")
>>> print(sorted(mylist))
['a', 'b', 'c']

>>> mylist = unique(([1, 2], [2, 3], [1, 2]))
>>> print(sorted(mylist))
[[1, 2], [2, 3]]

SCons.Util.uniquer_hashables(seq)
SCons.Util.updrive(path) → str
Make the drive letter (if any) upper case.
This is useful because Windows is inconsistent on the case of the drive letter, which can cause inconsistencies when calculating command signatures.

SCons.Util.wait_for_process_to_die(pid) → None
Wait for specified process to die, or alternatively kill it NOTE: This function operates best with psutil pypi package
TODO: Add timeout which raises exception

SCons environment utility functions.
Routines for working with environments and construction variables that don’t need the specifics of the Environment class.

SCons.Util.envs.AddMethod(obj, function: Callable, name: str | None = None) → None
Add a method to an object.
Adds function to obj if obj is a class object. Adds function as a bound method if obj is an instance object. If obj looks like an environment instance, use MethodWrapper to add it. If name is supplied it is used as the name of function. Although this works for any class object, the intent as a public API is to be used on Environment, to be able to add a method to all construction environments; it is preferred to use env.AddMethod to add to an individual environment.

>>> class A:
...   ...

>>> a = A()

>>> def f(self, x, y):
...   self.z = x + y

>>> AddMethod(A, f, "add")
>>> a.add(2, 4)
>>> print(a.z)
>>> a.data = ['a', 'b', 'c', 'd', 'e', 'f']
>>> AddMethod(a, lambda self, i: self.data[i], "listIndex")
>>> print(a.listIndex(3))
d
SCons.Util.envs.AddPathIfNotExists (env_dict, key, path, sep: str = ':') → None
Add a path element to a construction variable.  
key is looked up in env_dict, and path is added to it if it is not already present. env_dict[key] is assumed to be in the format of a PATH variable: a list of paths separated by sep tokens.

>>> env = {'PATH': '/bin:/usr/bin:/usr/local/bin'}
>>> AddPathIfNotExists(env, 'PATH', '/opt/bin')
>>> print(env['PATH'])
/opt/bin:/bin:/usr/bin:/usr/local/bin

SCons.Util.envs.AppendPath (oldpath, newpath, sep=':', delete_existing: bool = True, canonicalize: Callable | None = None) → list | str
Append newpath path elements to oldpath.
Will only add any particular path once (leaving the last one it encounters and ignoring the rest, to preserve path order), and will os.path.normpath and os.path.normcase all paths to help assure this. This can also handle the case where oldpath is a list instead of a string, in which case a list will be returned instead of a string. For example:

>>> p = AppendPath("/foo/bar:/foo", "/biz/boom:/foo")
>>> print(p)
/foo/bar:/biz/boom:/foo

If delete_existing is False, then adding a path that exists will not move it to the end; it will stay where it is in the list.

>>> p = AppendPath("/foo/bar:/foo", "/biz/boom:/foo", delete_existing=False)
>>> print(p)
/foo/bar:/foo:/biz/boom

If canonicalize is not None, it is applied to each element of newpath before use.
class SCons.Util.envs.MethodWrapper (obj: Any, method: Callable, name: str | None = None) → object
Bases: object
A generic Wrapper class that associates a method with an object.  
As part of creating this MethodWrapper object an attribute with the specified name (by default, the name of the supplied method) is added to the underlying object. When that new “method” is called, our __call__() method adds the object as the first argument, simulating the Python behavior of supplying “self” on method calls.  
We hang on to the name by which the method was added to the underlying base class so that we can provide a method to “clone” ourselves onto a new underlying object being copied (without which we wouldn’t need to save that info).

call (new_object)
Returns an object that re-binds the underlying “method” to the specified new object.

SCons.Util.envs.PrependPath (oldpath, newpath, sep=':', delete_existing: bool = True, canonicalize: Callable | None = None) → list | str
Prepend newpath path elements to oldpath.
Will only add any particular path once (leaving the first one it encounters and ignoring the rest, to preserve path order), and will os.path.normpath and os.path.normcase all paths to help assure this. This can also handle the case where oldpath is a list instead of a string, in which case a list will be returned instead of a string. For example:
SCons.Util package

```python
>>> p = PrependPath("/foo/bar:/foo", "/biz/boom:/foo")
>>> print(p)
/biz/boom:/foo/bar:/foo
```

If `delete_existing` is `False`, then adding a path that exists will not move it to the beginning; it will stay where it is in the list.

```python
>>> p = PrependPath("/foo/bar:/foo", "/biz/boom:/foo", delete_existing=False)
>>> print(p)
/biz/boom:/foo/bar:/foo
```

If `canonicalize` is not `None`, it is applied to each element of `newpath` before use.

SCons file locking functions.

Simple-minded filesystem-based locking. Provides a context manager which acquires a lock (or at least, permission) on entry and releases it on exit.

Usage:

```python
from SCons.Util.filelock import FileLock

with FileLock("myfile.txt", writer=True) as lock:
    print(f"Lock on {lock.file} acquired.")
    # work with the file as it is now locked
```

class SCons.Util.filelock.FileLock (file: str, timeout: int | None = None, delay: float | None = 0.05, writer: bool = False)

Bases: object

Lock a file using a lockfile.

Basic locking for when multiple processes may hit an externally shared resource that cannot depend on locking within a single SCons process. SCons does not have a lot of those, but caches come to mind.

Cross-platform safe, does not use any OS-specific features. Provides context manager support, or can be called with `acquire_lock()` and `release_lock()`.

Lock can be a write lock, which is held until released, or a read lock, which releases immediately upon acquisition - we want to not read a file which somebody else may be writing, but not create the writers starvation problem of the classic readers/writers lock.

TODO: Should default timeout be `None` (non-blocking), or 0 (block forever),
or some arbitrary number?

Parameters:

- **file** – name of file to lock. Only used to build the lockfile name.
- **timeout** – optional time (sec) to give up trying. If `None`, quit now if we failed to get the lock (non-blocking). If 0, block forever (well, a long time).
- **delay** – optional delay between tries [default 0.05s]
- **writer** – if True, obtain the lock for safe writing. If False (default), just wait till the lock is available, give it back right away.

Raises: `SConsLockFailure` – if the operation “timed out”, including the non-blocking mode.

```python
__enter__ () -> FileLock

Context manager entry: acquire lock if not holding.

__exit__ (exc_type, exc_value, exc_tb) -> None

Context manager exit: release lock if holding.

__repr__ () -> str

Nicer display if someone repr’s the lock class.
```
acquire_lock () → None
Acquire the lock, if possible.
    If the lock is in use, check again every delay seconds. Continue until lock acquired or timeout expires.
release_lock () → None
Release the lock by deleting the lockfile.

exception SCons.Util.FileLock.SConsLockFailure
    Bases: Exception
    Lock failure exception.

add_note ()
    Exception.add_note(note) – add a note to the exception

with_traceback ()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

SCons hash utility routines.

Routines for working with content and signature hashes.
SCons.Util.hashes.MD5collect (signatures)
    Deprecated. Use hash_collect() instead.
SCons.Util.hashes.MD5filesignature (fname, chunksize: int = 65536)
    Deprecated. Use hash_file_signature() instead.
SCons.Util.hashes.MD5signature (s)
    Deprecated. Use hash_signature() instead.
SCons.Util.hashes._attempt_get_hash_function (hash_name, hashlib_used=<module 'hashlib' from '/opt/local/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/hashlib.py'>, sys_used=<module 'sys' (built-in)>)
    Wrapper used to try to initialize a hash function given.
    If successful, returns the name of the hash function back to the user.
    Otherwise returns None.
SCons.Util.hashes._attempt_init_of_python_3_9_hash_object (hash_function_object, sys_used=<module 'sys' (built-in)>)
    Initialize hash function with non-security indicator.
    In Python 3.9 and onwards, hashlib constructors accept a keyword argument usedforsecurity, which, if set to False,
    lets us continue to use algorithms that have been deprecated either by FIPS or by Python itself, as the MD5 algorithm
    SCons prefers is not being used for security purposes as much as a short, 32 char hash that is resistant to accidental
    collisions.
    In prior versions of python, hashlib returns a native function wrapper, which errors out when it’s queried for the
    optional parameter, so this function wraps that call.
    It can still throw a ValueError if the initialization fails due to FIPS compliance issues, but that is assumed to be the
    responsibility of the caller.
SCons.Util.hashes._get_hash_object (hash_format, hashlib_used=<module 'hashlib' from '/opt/local/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/hashlib.py'>, sys_used=<module 'sys' (built-in)>)
    Allocates a hash object using the requested hash format.

    Parameters:    hash_format – Hash format to use.
    Returns:        hashlib object.
SCons.Util.hashes._set_allowed_viable_default_hashes (hashlib_used, sys_used=<module 'sys' (built-in)>) → None
    Check if the default hash algorithms can be called.
    This util class is sometimes called prior to setting the user-selected hash algorithm, meaning that on FIPS-compliant
    systems the library would default-initialize MD5 and throw an exception in set_hash_format. A common case is using
    the SConf options, which can run prior to main, and thus ignore the options.hash_format variable.
    This function checks the DEFAULT_HASH_FORMATS and sets the ALLOWED_HASH_FORMATS to only the ones
    that can be called. In Python >= 3.9 this will always default to MD5 as in Python 3.9 there is an optional attribute
    "usedforsecurity" set for the method.
    Throws if no allowed hash formats are detected.
SCons.Util package

SCons.Util.hashes._show_md5_warning (function_name) → None

Shows a deprecation warning for various MD5 functions.

SCons.Util.hashes.get_current_hash_algorithm_used ()

Returns the current hash algorithm name used.
Where the python version >= 3.9, this is expected to return md5. If python’s version is <= 3.8, this returns md5 on non-FIPS-mode platforms, and sha1 or sha256 on FIPS-mode Linux platforms.
This function is primarily useful for testing, where one expects a value to be one of N distinct hashes, and therefore the test needs to know which hash to select.

SCons.Util.hashes.get_hash_format ()

Retrieves the hash format or None if not overridden.
A return value of None does not guarantee that MD5 is being used; instead, it means that the default precedence order documented in SCons.Util.set_hash_format() is respected.

SCons.Util.hashes.hash_collect (signatures, hash_format=None)

Collects a list of signatures into an aggregate signature.

Parameters:
  • signatures – a list of signatures
  • hash_format – Specify to override default hash format

Returns: the aggregate signature

SCons.Util.hashes.hash_file_signature (fname, chunksize: int = 65536, hash_format=None)

Generate the md5 signature of a file

Parameters:
  • fname – file to hash
  • chunksize – chunk size to read
  • hash_format – Specify to override default hash format

Returns: String of Hex digits representing the signature

SCons.Util.hashes.hash_signature (s, hash_format=None)

Generate hash signature of a string

Parameters:
  • s – either string or bytes. Normally should be bytes
  • hash_format – Specify to override default hash format

Returns: String of hex digits representing the signature

SCons.Util.hashes.set_hash_format (hash_format, hashlib_used=<module 'hashlib' from '/opt/local/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/hashlib.py'>, sys_used=<module 'sys' (built-in)>)

Sets the default hash format used by SCons.
If hash_format is None or an empty string, the default is determined by this function.
Currently the default behavior is to use the first available format of the following options: MD5, SHA1, SHA256.

Various SCons utility functions

Routines which check types and do type conversions.

class SCons.Util.sctypes.Null (*args, **kwargs)

Bases: object

Null objects always and reliably ‘do nothing’.

class SCons.Util.sctypes.NullSeq (*args, **kwargs)

Bases: Null

A Null object that can also be iterated over.

SCons.Util.sctypes.get_env_bool (env, name: str, default: bool = False) → bool

Convert a construction variable to bool.
If the value of name in dict-like object env is ‘true’, ‘yes’, ‘y’, ‘on’ (case insensitive) or anything convertible to int that yields non-zero, return True; if ‘false’, ‘no’, ‘n’, ‘off’ (case insensitive) or a number that converts to integer zero return False. Otherwise, or if name is not found, return the value of default.
### Parameters:
- `env` – construction environment, or any dict-like object.
- `name` – name of the variable.
- `default` – value to return if `name` not in `env` or cannot be converted (default: False).

SCons.Util.sctypes.get_environment_var (varstr) → str | None
Return undecorated construction variable string.
Determine if `varstr` looks like a reference to a single environment variable, like "$FOO" or "$\{FOO\}". If so, return that variable with no decorations, like "FOO". If not, return None.

SCons.Util.sctypes.get_os_env_bool (name: str, default: bool = False) → bool
Convert an external environment variable to boolean.
Like `get_env_bool()`, but uses os.environ as the lookup dict.

SCons.Util.sctypes.is_Dict (obj, isinstance=<built-in function isinstance>, DictTypes=(<class 'dict'>, <class 'collections.UserDict'>)) → bool
Check if object is a dict.

SCons.Util.sctypes.is_List (obj, isinstance=<built-in function isinstance>, ListTypes=(<class 'list'>, <class 'collections.UserList'>, <class 'collections.deque'>)) → bool
Check if object is a list.

SCons.Util.sctypes.is_Scalar (obj, isinstance=<built-in function isinstance>, StringTypes=(<class 'str'>, <class 'collections.UserString'>), Iterable=<class 'collections.abc.Iterable'>) → bool
Check if object is a scalar: not a container or iterable.

SCons.Util.sctypes.is_Sequence (obj, isinstance=<built-in function isinstance>, SequenceTypes=(<class 'list'>, <class 'tuple'>, <class 'collections.deque'>, <class 'collections.UserList'>, <class 'collections.abc.MappingView'>)) → bool
Check if object is a sequence.

SCons.Util.sctypes.is_String (obj, isinstance=<built-in function isinstance>, StringTypes=(<class 'str'>, <class 'collections.UserString'>)) → bool
Check if object is a string.

SCons.Util.sctypes.is_Tuple (obj, isinstance=<built-in function isinstance>, tuple=<class 'tuple'>) → bool
Check if object is a tuple.

SCons.Util.sctypes.to_String (obj, isinstance=<built-in function isinstance>, str=<class 'str'>, UserString=<class 'collections.UserString'>, BaseStringTypes=<class 'str'>) → str
Return a string version of obj.
Use this for data likely to be well-behaved. Use `to_Text()` for unknown file data that needs to be decoded.

SCons.Util.sctypes.to_String_for_signature (obj, to_String_for_subst=<function to_String_for_subst>, AttributeError=<class 'AttributeError'>) → str
Return a string version of obj for signature usage.
Like `to_String_for_subst()` but has special handling for scons objects that have a for_signature() method, and for dicts.

SCons.Util.sctypes.to_String_for_subst (obj, isinstance=<built-in function isinstance>, str=<class 'str'>, BaseStringTypes=<class 'str'>, SequenceTypes=(<class 'list'>, <class 'tuple'>, <class 'collections.deque'>, <class 'collections.UserList'>, <class 'collections.abc.MappingView'>), UserString=<class 'collections.UserString'>) → str
Return a string version of obj for subst usage.

SCons.Util.sctypes.to_Text (data: bytes) → str
Return bytes data converted to text.
Useful for whole-file reads where the data needs some interpretation, particularly for Scanners. Attempts to figure out what the encoding of the text is based upon the BOM bytes, and then decodes the contents so that it’s a valid python string.

SCons.Util.sctypes.to_bytes (s) → bytes
Convert object to bytes.

SCons.Util.sctypes.to_str (s) → str
Convert object to string.

SCons statistics routines.
SCons.Variables package

This package provides a way to gather various statistics during an SCons run and dump that info in several formats. Additionally, it probably makes sense to do stderr/stdout output of those statistics here as well.

There are basically two types of stats:

1. Timer (start/stop/time) for specific event. These events can be hierarchical. So you can record the children events of some parent. Think program compile could contain the total Program builder time, which could include linking, and stripping the executable.

2. Counter. Counting the number of events and/or objects created. This would likely only be reported at the end of a given SCons run, though it might be useful to query during a run.

   ```python
   class SCons.Util.stats.CountStats
       Bases: Stats
       _abc_impl = _abc.abc_data object
       do_append (label)
       do_nothing (*args, **kw)
       do_print ()
       enable (outfp)
   
   class SCons.Util.stats.MemStats
       Bases: Stats
       _abc_impl = _abc.abc_data object
       do_append (label)
       do_nothing (*args, **kw)
       do_print ()
       enable (outfp)
   
   class SCons.Util.stats.Stats
       Bases: ABC
       _abc_impl = _abc.abc_data object
       do_append (label)
       do_nothing (*args, **kw)
       do_print ()
       enable (outfp)
   
   class SCons.Util.stats.TimeStats
       Bases: Stats
       _abc_impl = _abc.abc_data object
       add_command (command, start_time, finish_time)
       do_append (label)
       do_nothing (*args, **kw)
       do_print ()
       enable (outfp)
       total_times (build_time, sconscript_time, scons_exec_time, command_exec_time)
   
   SCons.Util.stats.add_stat_type (name, stat_object)
       Add a statistic type to the global collection
   
   SCons.Util.stats.write_scons_stats_file ()
       Actually write the JSON file with debug information. Depending which of: count, time, action-timestamps, memory their information will be written.
   
SCons.Variables package

Submodules

SCons.Variables.BoolVariable module

Variable type for true/false Variables.

Usage example:
opts = Variables()
opts.Add(BoolVariable('embedded', 'build for an embedded system', False))
...
if env['embedded']:
...

Return a tuple describing a boolean SCons Variable.
The input parameters describe a boolean option. Returns a tuple including the correct converter and validator. The help text will have (yes|no) automatically appended to show the valid values. The result is usable as input to Add().

SCons.Variables.BoolVariable._text2bool (val: str) → bool
Convert boolean-like string to boolean.
If val looks like it expresses a bool-like value, based on the TRUE_STRINGS and FALSE_STRINGS tuples, return the appropriate value.
This is usable as a converter function for SCons Variables.

Raises: ValueError – if val cannot be converted to boolean.

SCons.Variables.BoolVariable._validator (key, val, env) → None
Validate that the value of key in env is a boolean.
Parmaeter val is not used in the check.
Usable as a validator function for SCons Variables.

Raises:
- KeyError – if key is not set in env
- UserError – if the value of key is not True or False.

SCons.Variables.EnumVariable module

Variable type for enumeration Variables.
Enumeration variables allow selection of one from a specified set of values.
Usage example:

opts = Variables()
opts.Add(
    EnumVariable(
        'debug',
        help='debug output and symbols',
        default='no',
        allowed_values=('yes', 'no', 'full'),
        map={},
        ignorecase=2,
    )
)
...
if env['debug'] == 'full':
...

SCons.Variables.EnumVariable(EnumVariable (key, help, default, allowed_values, map={}, ignorecase: int = 0) → Tuple[str, str, str, Callable, Callable]
Return a tuple describing an enumeration SCons Variable.
The input parameters describe an option with only certain values allowed. Returns A tuple including an appropriate converter and validator. The result is usable as input to Add().
key and default are passed directly on to Add().
help is the descriptive part of the help text, and will have the allowed values automatically appended.
allowed_values is a list of strings, which are the allowed values for this option.
The map-dictionary may be used for converting the input value into canonical values (e.g. for aliases). The value of ignorecase defines the behaviour of the validator:

- 0: the validator/converter are case-sensitive.
- 1: the validator/converter are case-insensitive.
- 2: the validator/converter is case-insensitive and the converted value will always be lower-case.

The validator tests whether the value is in the list of allowed values. The converter converts input values according to the given map-dictionary (unmapped input values are returned unchanged).

SCons.Variables.ListVariable module

Variable type for list Variables.

A ‘list’ option may either be ‘all’, ‘none’ or a list of names separated by comma. After the option has been processed, the option value holds either the named list elements, all list elements or no list elements at all.

Usage example:

```python
list_of_libs = Split('x11 gl qt ical')

opts = Variables()
opts.Add(
    ListVariable(
        'shared',
        help='libraries to build as shared libraries',
        default='all',
        elems=list_of_libs,
    )
)
...
for lib in list_of_libs:
    if lib in env['shared']:
        env.SharedObject(...)
    else:
        env.Object(...)
```

SCons.Variables.ListVariable._converter (val, allowedElems, mapdict) → _ListVariable

Return a tuple describing a list SCons Variable.

The input parameters describe a ‘list’ option. Returns a tuple including the correct converter and validator. The result is usable for input to Add().

help will have text appended indicating the legal values (not including any extra names from map).

map can be used to map alternative names to the ones in names - that is, a form of alias.

A ‘list’ option may either be ‘all’, ‘none’ or a list of names (separated by commas).

SCons.Variables.ListVariable.ListVariable (key, help, default, names, map={}) → Tuple [str, str, str, None, Callable]

Return a tuple describing a list SCons Variable.

The input parameters describe a ‘list’ option. Returns a tuple including the correct converter and validator. The result is usable for input to Add().

help will have text appended indicating the legal values (not including any extra names from map).

map can be used to map alternative names to the ones in names - that is, a form of alias.

A ‘list’ option may either be ‘all’, ‘none’ or a list of names (separated by commas).

SCons.Variables.ListVariable._converter (val, allowedElems, mapdict) → _ListVariable

SCons.Variables.PackageVariable module

Variable type for package Variables.

To be used whenever a ‘package’ may be enabled/disabled and the package path may be specified.

Given these options
SCons.Variables package

```python
x11=no   (disables X11 support)
x11=yes  (will search for the package installation dir)
x11=/usr/local/X11 (will check this path for existence)
```

Can be used as a replacement for autoconf's `--with-xxx=yyy`

```python
opts = Variables()
 opts.Add(
   PackageVariable(
     key='x11',
     help='use X11 installed here (yes = search some places)',
     default='yes'
   )
 )
...
if env['x11'] == True:
  dir = ...  # search X11 in some standard places ...
  env['x11'] = dir
if env['x11']:
  ...  # build with x11 ...
```

Return a tuple describing a package list SCons Variable.
The input parameters describe a ‘package list’ option. Returns a tuple including the correct converter and validator appended. The result is usable as input to Add().

A ‘package list’ option may either be ‘all’, ‘none’ or a pathname string. This information is appended to help.

SCons.Variables.PathVariable module
Variable type for path Variables.
To be used whenever a user-specified path override setting should be allowed.

**Arguments to PathVariable are:**

- `key` - name of this option on the command line (e.g. “prefix”)
- `help` - help string for option
- `default` - default value for this option
- `validator` - [optional] validator for option value. Predefined are:
  - `PathAccept` - accepts any path setting; no validation
  - `PathsDir` - path must be an existing directory
  - `PathsDirCreate` - path must be a dir; will create
  - `PathFile` - path must be a file
  - `PathExists` - path must exist (any type) [default]

The `validator` is a function that is called and which should return True or False to indicate if the path is valid. The arguments to the validator function are: `(key, val, env)`. `key` is the name of the option, `val` is the path specified for the option, and `env` is the environment to which the Options have been added.

Usage example:
opts = Variables()
opts.Add(
    PathVariable(
        'qtdir',
        help='where the root of Qt is installed',
        default=qtdir,
        validator=PathIsDir,
    )
)
opts.Add(
    PathVariable(
        'qt_includes',
        help='where the Qt includes are installed',
        default='$qtdir/includes',
        validator=PathIsDirCreate,
    )
)
opts.Add(
    PathVariable(
        'qt_libraries',
        help='where the Qt library is installed',
        default='$qtdir/lib',
    )
)

Module contents

Adds user-friendly customizable variables to an SCons build.

```python
class SCons.Variables.Variables (files=None, args=None, is_global: bool = True)

Bases: object

Holds all the options, updates the environment with the variables, and renders the help text.
If is_global is true, this is a singleton, create only once.
```

**Parameters:**

- `files (optional)` – List of option configuration files to load (backward compatibility). If a single string is passed it is automatically placed in a file list (Default value = None)
- `args (optional)` – dictionary to override values set from files. (Default value = None)
- `is_global (optional)` – global instance? (Default value = True)

### Add

`Add (key, *args, **kwargs) → None`

Adds an option.

**Parameters:**

- `key` – the name of the variable, or a 5-tuple (or list). If a tuple, and there are no additional arguments, the tuple is unpacked into the four named kwargs from below. If a tuple and there are additional arguments, the first word of the tuple is taken as the key, and the remainder as aliases.
- `*args` – optional positional arguments, corresponding to the four named kwargs below.
**Keyword Arguments:**
- `help` – help text for the options (Default value = "")
- `default` – default value for option (Default value = None)
- `validator` – function called to validate the option's value (Default value = None)
- `converter` – function to be called to convert the option's value before putting it in the environment. (Default value = None)
- `**kwargs` – arbitrary keyword arguments used by the variable itself.

AddVariables (*optlist*) → None
Adds a list of options.
Each list element is a tuple/list of arguments to be passed on to the underlying method for adding options.
Example:

```python
opt.AddVariables(
    ('debug', '', 0),
    ('CC', 'The C compiler'),
    ('VALIDATE', 'An option for testing validation', 'notset', validator, None),
)
```

FormatVariableHelpText (env, key, help, default, actual, aliases=None) → str
Generates the help text for the options.

**Parameters:**
- `env` – an environment that is used to get the current values of the options.
- `sort` – Either a comparison function used for sorting (must take two arguments and return -1, 0 or 1) or a boolean to indicate if it should be sorted.

Save (filename, env) → None
Save the options to a file.
Saves all the options which have non-default settings to the given file as Python expressions. This file can then be used to load the options for a subsequent run. This can be used to create an option cache file.

**Parameters:**
- `filename` – Name of the file to save into
- `env` – the environment get the option values from

UnknownVariables () → dict
Returns unknown variables.
Identifies options that were not known, declared options in this object.

Update (env, args=None) → None
Updates an environment with the option variables.

**Parameters:**
- `env` – the environment to update.
- `args` *(optional)* – a dictionary of keys and values to update in `env`. If omitted, uses the variables from the commandline.

Indices and Tables
- `genindex`
Indices and Tables

- modindex
- search
Index

_index__ (SCons.cpp.DumbPreProcessor method)
(SCons.cpp.FunctionEvaluator method)
(SCons.cpp.PreProcessor method)
(SCons.Scanner.C.SConsCPPConditionalScanner method)
(SCons.Scanner.C.SConsCPPScanner method)
(SCons.Scanner.Classic method)
(SCons.Scanner.ClassicCPP method)
(SCons.Scanner.Current method)
(SCons.Scanner.D.D method)
(SCons.Scanner.Fortran.F90Scanner method)
(SCons.Scanner.LaTeX.LaTeX method)
(SCons.Scanner.ScannerBase method)
(SCons.Tool.ToolInitializerMethod method)

__call__() (SCons.cpp.DumbPreProcessor method)
(SCons.cpp.FunctionEvaluator method)
(SCons.cpp.PreProcessor method)
(SCons.Scanner.C.SConsCPPConditionalScanner method)
(SCons.Scanner.C.SConsCPPScanner method)
(SCons.Scanner.Classic method)
(SCons.Scanner.ClassicCPP method)
(SCons.Scanner.Current method)
(SCons.Scanner.D.D method)
(SCons.Scanner.Fortran.F90Scanner method)
(SCons.Scanner.LaTeX.LaTeX method)
(SCons.Scanner.ScannerBase method)
(SCons.Tool.ToolInitializerMethod method)

__clearRepositoryCache__() (SCons.Node.FS.Dir method)


dmap_cache (SCons.Node.FS.File attribute)
__dmap_sig_cache (SCons.Node.FS.File attribute)

__enter__() (SCons.Util.filelock.FileLock method)
__exit__() (SCons.Util.filelock.FileLock method)

get_abspath() (SCons.Node.FS.EntryProxy method)
get_base_path() (SCons.Node.FS.EntryProxy method)

get_dir() (SCons.Node.FS.EntryProxy method)
get_file() (SCons.Node.FS.EntryProxy method)
get_filebase() (SCons.Node.FS.EntryProxy method)
get_posix_path() (SCons.Node.FS.EntryProxy method)

get_relpath() (SCons.Node.FS.EntryProxy method)
get_rsrcdir() (SCons.Node.FS.EntryProxy method)
get_rsrcnode() (SCons.Node.FS.EntryProxy method)
get_suffix() (SCons.Node.FS.EntryProxy method)

get_windows_path() (SCons.Node.FS.EntryProxy method)

__getattr__() (SCons.Builder.CompositeBuilder method)
(SCons.Node.FS.Base method)
(SCons.Node.FS.Dir method)
(SCons.Node.FS.Entry method)
(SCons.Node.FS.File method)
(SCons.Node.FS.RootDir method)
(SCons.Script.SConsOptions.SConsValues method)
(SCons.Util.NodeList method)
(SCons.Util.Proxy method)


(SCons.Node.BuildInfoBase method)
(SCons.Node.FS.DirBuildInfo method)
(SCons.Node.FS.DirNodeInfo method)
(SCons.Node.FS.FileBuildInfo method)
(SCons.Node.FS.FileNodeInfo method)
(SCons.Node.NodeInfoBase method)
(SCons.SConf.SConfBuildInfo method)


__lib_either_version_flag__() (in module SCons.Defaults)
__libversionflags__() (in module SCons.Defaults)

__lt__() (SCons.Node.FS.Base method)
(SCons.Node.FS.Dir method)
(SCons.Node.FS.Entry method)
(SCons.Node.FS.File method)
(SCons.Node.FS.RootDir method)

__make_unique__() (SCons.Util.UniqueList method)

__repr__() (SCons.Util.filelock.FileLock method)

__resetDuplicate__() (SCons.Node.FS.Dir method)

(SCons.Node.FS.RootDir method)
(SCons.Node.FS.Dir method)
(SCons.Node.FS.Entry method)
(SCons.Node.FS.File method)
(SCons.Node.FS.RootDir method)
add_to_waiting_s_e() (SCons.Node.Alias.Alias method)
(SCons.Node.FS.RootDir method)
(SCons.Node.FS.Dir method)
(SCons.Node.FS.Entry method)
(SCons.Node.FS.File method)
(SCons.Node.FS.RootDir method)
(SCons.Node.FS.RootDir method)
(SCons.Node.FS.Dir method)
(SCons.Node.FS.Entry method)
(SCons.Node.FS.File method)
(SCons.Node.FS.RootDir method)
AddBatchExecutor() (in module SCons.Executor)
AddMethod() (in module SCons.Util.envs)
(SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Environment.SubstitutionEnvironment method)
(SCons.Script.SConscript.SConsEnvironment method)
AddOption() (in module SCons.Script.Main)
AddPathIfNotExists() (in module SCons.Util.envs)
AddPostAction() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Script.SConscript.SConsEnvironment method)
AddPreAction() (SCons.Environment.Base method)
CleanTask (class in SCons.Script.Main)
cleanup() (SCons.Executor.Executor method)
   (SCons.Executor.Null method)
   (SCons.Taskmaster.Job.ThreadPool method)
   (SCons.Taskmaster.Taskmaster method)
Cleanup_CPP_Expressions() (in module SCons.cpp)
clear() (SCons.Builder.CallableSelector method)
   (SCons.Builder.DictCmdGenerator method)
   (SCons.Builder.DictEmitter method)
   (SCons.Builder.ListEmitter method)
   (SCons.Builder.OverrideWarner method)
   (SCons.Environment.BuilderDict method)
   (SCons.Executor.TSList method)
   (SCons.Node.Alias.AliasNameSpace method)
   (SCons.Node.FS.Base method)
   (SCons.Node.FS.Dir method)
   (SCons.Node.FS.Entry method)
   (SCons.Node.FS.File method)
   (SCons.Node.FS.RootDir method)
   (SCons.Script.TargetList method)
   (SCons.Subst.ListSubber method)
   (SCons.Subst.Targets_or_Sources method)
   (SCons.Util.CLVar method)
   (SCons.Util.NodeList method)
   (SCons.Util.Selector method)
   (SCons.Util.UniqueList method)
   (SCons.Node.FS.Base method)
   (SCons.Node.FS.Dir method)
   (SCons.Node.FS.Entry method)
   (SCons.Node.FS.File method)
   (SCons.Node.FS.RootDir method)
Clone() (SCons.Environment.Base method)
clone() (SCons.Environment.BuilderWrapper method)
Clone() (SCons.Environment.OverrideEnvironment method)
   (SCons.Script.SConscript.SConscriptEnvironment method)
close() (SCons.Util.envs.MethodWrapper method)
close_strip() (SCons.Subst.ListSubber method)
CLVar (class in SCons.Util)
cmdloop()
   (SCons.Script.Interactive.SConsInteractiveCmd method)
CmdStringBuilder (class in SCons.Subst)
cmp() (in module SCons.Util)
collect_node_states() (SCons.SConf.SConfBuildTask method)
columnize()
   (SCons.Script.Interactive.SConsInteractiveCmd method)
Command() (SCons.Environment.Base method)
   (SCons.Script.SConscript.SConscriptEnvironment method)
CommandAction (class in SCons.Action)
CommandGeneratorAction (class in SCons.Action)
CompileProg() (SCons.SConf.CheckContext method)
CompileSharedObject() (SCons.SConf.CheckContext method)
complete() (SCons.Script.Interactive.SConsInteractiveCmd method)
complete_help() (SCons.Script.Interactive.SConsInteractiveCmd method)
COMPLETED
completdefault() (SCons.Script.Interactive.SConsInteractiveCmd method)
completnames() (SCons.Script.Interactive.SConsInteractiveCmd method)
CompositeBuilder (class in SCons.Builder)
compute_exports() (in module SCons.Script.SConscript)
Configure() (in module SCons.Script.SConscript)
   (SCons.Environment.Base method)
   (SCons.Environment.OverrideEnvironment method)
   (SCons.Script.SConscript.SConscriptEnvironment method)
configure_trace() (SCons.Taskmaster.Taskmaster method)
ConfigureCacheError
ConfigureDryRunError
CONST_ACTIONS
(SCons.Script.SConsOptions.SConsOption attribute)
containsAll() (in module SCons.Util)
containsAny() (in module SCons.Util)
containsOnly() (in module SCons.Util)
containsigs (SCons.Node.FS.Dir attribute)
  (SCons.Node.FS.Entry attribute)
  (SCons.Node.FS.File attribute)
  (SCons.Node.FS.RootDir attribute)
  (SCons.Node.FS.DirNodeInfo method)
  (SCons.Node.NodeInfoBase method)
convert_copy_attrs (SCons.Node.FS.File attribute)
convert_from_sconsign() (SCons.Node.FS.FileBuildInfo method)
  (SCons.Conf.SConfBuildInfo method)
  (SCons.SConsign.SConsignEntry method)
convert_old_entry() (SCons.Node.FS.File method)
convert_sigattrs (SCons.Node.FS.File method)
convert_to_BeBuildError() (in module SCons.Errors)
convert_to_sconsign() (SCons.Node.FS.FileBuildInfo method)
  (SCons.Conf.SConfBuildInfo method)
  (SCons.SConsign.SConsignEntry method)
copy() (SCons.Node.FS.FS method)
  (SCons.Node.FS.LocalFS method)
copy2() (SCons.Node.FS.FS method)
copy_from_cache() (SCons.CacheDir.CacheDir class method)
copy_func() (in module SCons.Defaults)
copy_non_reserved_keywords() (in module SCons.Environment)
copy_strfunc() (in module SCons.Defaults)
copy_to_cache() (SCons.CacheDir.CacheDir class method)
corrupt_dblite_warning() (in module SCons.SConsign)
corruptSConsignWarning
count (SCons.Script.Main.Progressor attribute)
count() (SCons.Builder.ListProgressor attribute)
  (SCons.Executor.TSList method)
  (SCons.Memoize.CountDict method)
  (SCons.Memoize.CountValue method)
  (SCons.Script.TargetList method)
  (SCons.Subst.CmdStringHolder method)
  (SCons.Subst.ListSubber method)
  (SCons.Subst.Targets_or_Sources method)
  (SCons.Util.CLVar method)
  (SCons.Util.NodeList method)
  (SCons.Util.Selector method)
  (SCons.Util.UniqueList method)
CountDict (class in SCons.Memoize)
CountDictCall() (in module SCons.Memoize)
Counter (class in SCons.Memoize)
countLoggedInstances() (in module SCons.Debug)
do_help() (SCons.Script.Interactive.SConsInteractiveCmd method)

do_if() (SCons.cpp.DumbPreProcessor method)
  (SCons.cpp.PreProcessor method)
  (SCons.Scanner.C.SConsCPPConditionalScanner method)
  (SCons.Scanner.C.SConsCPPScanner method)

do_ifdef() (SCons.cpp.DumbPreProcessor method)
  (SCons.cpp.PreProcessor method)
  (SCons.Scanner.C.SConsCPPConditionalScanner method)
  (SCons.Scanner.C.SConsCPPScanner method)

do_ifndef() (SCons.cpp.DumbPreProcessor method)
  (SCons.cpp.PreProcessor method)
  (SCons.Scanner.C.SConsCPPConditionalScanner method)
  (SCons.Scanner.C.SConsCPPScanner method)

do_include() (SCons.cpp.DumbPreProcessor method)
  (SCons.cpp.PreProcessor method)
  (SCons.Scanner.C.SConsCPPConditionalScanner method)
  (SCons.Scanner.C.SConsCPPScanner method)

do_include_next() (SCons.cpp.DumbPreProcessor method)
  (SCons.cpp.PreProcessor method)
  (SCons.Scanner.C.SConsCPPConditionalScanner method)
  (SCons.Scanner.C.SConsCPPScanner method)

do_nothing() (in module SCons.Node)
  (SCons.cpp.DumbPreProcessor method)
  (SCons.cpp.PreProcessor method)
  (SCons.Scanner.C.SConsCPPConditionalScanner method)
  (SCons.Scanner.C.SConsCPPScanner method)
  (SCons.Util.stats.CountStats method)
  (SCons.Util.stats.MemStats method)
  (SCons.Util.stats.Stats method)
  (SCons.Util.stats.TimeStats method)

do_nothing_node() (in module SCons.Node)

do_print() (SCons.Util.stats.CountStats method)
  (SCons.Util.stats.MemStats method)
  (SCons.Util.stats.Stats method)
  (SCons.Util.stats.TimeStats method)

do_shell() (SCons.Script.Interactive.SConsInteractiveCmd method)

do_undef() (SCons.cpp.DumbPreProcessor method)
  (SCons.cpp.PreProcessor method)
  (SCons.Scanner.C.SConsCPPConditionalScanner method)
  (SCons.Scanner.C.SConsCPPScanner method)

do_version() (SCons.Script.Interactive.SConsInteractiveCmd method)

doc_header (SCons.Script.Interactive.SConsInteractiveCmd attribute)

doc_leader (SCons.Script.Interactive.SConsInteractiveCmd attribute)

DScanner() (in module SCons.Scanner.D)

DumbPreProcessor (class in SCons.cpp)

Dump() (in module SCons.Memoize)
  (SCons.Environment.Base method)
  (SCons.Environment.OverrideEnvironment method)
  (SCons.Script.SConscript.SConsEnvironment method)

dump_caller_counts() (in module SCons.Debug)

dump_stats() (in module SCons.Debug)

dumpLoggedInstances() (in module SCons.Taskmaster)

duplicate (SCons.Node.FS.Base attribute)
  (SCons.Node.FS.Dir attribute)
FindSourceFiles() (SCons.Environment.Base method)
  (SCons.Environment.OverrideEnvironment method)
  (SCons.Script.SConscript.SConsEnvironment method)
FindTool() (in module SCons.Tool)
Finish() (SCons.SConf.SConfBase method)
flatten() (in module SCons.Util)
Flatten() (SCons.Environment.Base method)
  (SCons.Environment.OverrideEnvironment method)
  (SCons.Script.SConscript.SConsEnvironment method)
flatten_sequence() (in module SCons.Util)
flush() (SCons.SConf.Streamer method)
fmt (SCons.Variables.Variables attribute)
  (SCons.Node.FS.Base method)
  (SCons.Node.FS.Dir method)
  (SCons.Node.FS.Entry method)
  (SCons.Node.FS.File method)
  (SCons.Node.FS.RootDir method)
  (SCons.Node.FS.Value method)
  (SCons.Subst.Literal method)
  (SCons.Subst.SpecialAttrWrapper method)
ForDirectory (in module SCons.SConsign)
  (SCons.Node.FS.DirNodeInfo method)
  (SCons.Node.FS.FileBuildInfo method)
  (SCons.Node.FS.FileInfo method)
  (SCons.Node.NodeInfoBase method)
  (SCons.SConf.SConfBuildInfo method)
  (SCons.Subst.CmdStringHolder method)
  (SCons.Util.DispatchingFormatter method)
format_description() (SCons.Script.SConsOptions.SConsIndentedHelpFormatter method)
  (SCons.Script.SConsOptions.SConsOptionParser method)
format_epilog() (SCons.Script.SConsOptions.SConsIndentedHelpFormatter method)
  (SCons.Script.SConsOptions.SConsOptionParser method)
format_heading() (SCons.Script.SConsOptions.SConsIndentedHelpFormatter method)
format_help() (SCons.Script.SConsOptions.SConsOptionGroup method)
  (SCons.Script.SConsOptions.SConsOptionParser method)
format_local_option_help() (SCons.Script.SConsOptions.SConsOptionParser method)
format_map() (SCons.Subst.CmdStringHolder method)
format_option() (SCons.Script.SConsOptions.SConsIndentedHelpFormatter method)
format_option_help() (SCons.Script.SConsOptions.SConsOptionGroup method)
  (SCons.Script.SConsOptions.SConsOptionParser method)
format_option_strings() (SCons.Script.SConsOptions.SConsIndentedHelpFormatter method)
format_usage() (SCons.Script.SConsOptions.SConsIndentedHelpFormatter method)
formatException() (SCons.Util.DispatchingFormatter method)
formatMessage() (SCons.Util.DispatchingFormatter method)
formatStack() (SCons.Util.DispatchingFormatter method)
formatTime() (SCons.Util.DispatchingFormatter method)
FormatVariableHelpText() (SCons.Variables.Variables method)
FortranCxxMixWarning
FortranScan() (in module SCons.Scanner.Fortran)
Frame (class in SCons.Script.SConscript)
fromkeys() (SCons.Builder.CallableSelector method)
  (SCons.Builder.DictCmdGenerator method)
  (SCons.Builder.DictEmitter method)
  (SCons.Builder.OverrideWarner class method)
  (SCons.Environment.BuilderDict class method)
  (SCons.Node.Alias.AliasNameSpace class method)
  (SCons.Util.Selector method)
FS (class in SCons.Node.FS)
get_root() (SCons.Node.FS.FS method)

get_scanner() (SCons.Environment.Base method)
  (SCons.Environment.OverrideEnvironment method)
  (SCons.Script.SConscript.SConsEnvironment method)

get_size() (SCons.Node.FS.File method)

get_skeys() (SCons.Scanner.Classic method)
  (SCons.Scanner.ClassicCPP method)
  (SCons.Scanner.Current method)
  (SCons.Scanner.D.D method)
  (SCons.Scanner.Fortran.F90Scanner method)
  (SCons.Scanner.LaTeX.LaTeX method)
  (SCons.Scanner.ScannerBase method)
  (SCons.Scanner.Selector method)

  (SCons.Node.FS.Base method)
  (SCons.Node.FS.Dir method)
  (SCons.Node.FS.Entry method)
  (SCons.Node.FS.File method)
  (SCons.Node.FS.RootDir method)

get_sources() (SCons.Executor.Executor method)

get_src_builders() (SCons.Builder.BuilderBase method)

get_src_suffix() (SCons.Builder.BuilderBase method)

  (SCons.Node.FS.Base method)
  (SCons.Node.FS.Dir method)
  (SCons.Node.FS.Entry method)
  (SCons.Node.FS.File method)
  (SCons.Node.FS.RootDir method)

  (SCons.Node.FS.Base method)
  (SCons.Node.FS.Dir method)
  (SCons.Node.FS.Entry method)
  (SCons.Node.FS.File method)
  (SCons.Node.FS.RootDir method)

  (SCons.Node.FS.Base method)
  (SCons.Node.FS.Dir method)
  (SCons.Node.FS.Entry method)
  (SCons.Node.FS.File method)
  (SCons.Node.FS.RootDir method)

get_suffix() (SCons.Builder.BuilderBase method)
  (SCons.Node.FS.Base method)
  (SCons.Node.FS.Dir method)
  (SCons.Node.FS.Entry method)
  (SCons.Node.FS.File method)
  (SCons.Node.FS.RootDir method)

  (SCons.Node.FS.Base method)
  (SCons.Node.FS.Dir method)
  (SCons.Node.FS.Entry method)
  (SCons.Node.FS.File method)
  (SCons.Node.FS.RootDir method)

get_system_root() (in module SCons.Platform.win32)

get_target() (SCons.SConf.SConfBuildTask method)
  (SCons.Script.Main.BuildTask method)
  (SCons.Script.Main.CleanTask method)
(SCons.Node.FS.Base method)
(SCons.Node.FS.Dir method)
(SCons.Node.FS.Entry method)
(SCons.Node.FS.File method)
(SCons.Node.FS.RootDir method)
getvalue() (SCons.SConf.Streamer method)
Glob() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
glob() (SCons.Node.FS.Dir method)
Glob() (SCons.Node.FS.FS method)
glob() (SCons.Node.FS.RootDir method)
Glob() (SCons.Script.SConscript.SConsEnvironment method)
gvars() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Environment.SubstitutionEnvironment method)
(SCons.Script.SConscript.SConsEnvironment method)
handle_missing_SConscript() (in module SCons.Script.SConscript)
has_builder() (SCons.Node.Alias.Alias method)
(SCons.Node.FS.Base method)
(SCons.Node.FS.Dir method)
(SCons.Node.FS.Entry method)
(SCons.Node.FS.File method)
(SCons.Node.FS.RootDir method)
has_explicit_builder() (SCons.Node.Alias.Alias method)
(SCons.Node.FS.Base method)
(SCons.Node.FS.Dir method)
(SCons.Node.FS.Entry method)
(SCons.Node.FS.File method)
(SCons.Node.FS.RootDir method)
has_glob_magic() (in module SCons.Node.FS)
has_option() (SCons.Script.SConsOptions.SConsOptionGroup method)
(SCons.Script.SConsOptions.SConsOptionParser method)
has_src_builder() (SCons.Node.FS.File method)
hash_chunkszie (SCons.Node.FS.File attribute)
hash_collect() (in module SCons.Util.hashes)
hash_file_signature() (in module SCons.Util.hashes)
hash_signature() (in module SCons.Util.hashes)
Help() (SCons.Script.SConscript.SConsEnvironment method)
HelpFunction() (in module SCons.Script)
hit_ratio (SCons.CacheDir.CacheDir property)
I
(SCons.Taskmaster.Job.Worker property)
identchars (SCons.Script.Interactive.SConsInteractiveCmd attribute)
IDLScan() (in module SCons.Scanner.IDL)
IDX() (in module SCons.Util)
(SCons.Node.FS.Base attribute)
(SCons.Node.FS.Dir attribute)
(SCons.Node.FS.Entry attribute)
(SCons.Node.FS.File attribute)
(SCons.Node.FS.RootDir attribute)
(SCons.Node.Node attribute)
Ignore() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Script.SConscript.SConsEnvironment method)
ignore_cycle() (in module SCons.Node)
ignore_diskcheck_match() (in module SCons.Node.FS)
(SCons.Node.FS.Base attribute)
Precious() (SCons.Environment.Base method)
  (SCons.Environment.OverrideEnvironment method)
  (SCons.Script.SConscript.SConsEnvironment method)
precmd() (SCons.Script.Interactive.SConsInteractiveCmd method)
preloop() (SCons.Script.Interactive.SConsInteractiveCmd method)
prepare() (SCons.Executor.Executor method)
  (SCons.Executor.Null method)
  (SCons.Node.FS.Base method)
  (SCons.Node.FS.Dir method)
  (SCons.Node.FS.Entry method)
  (SCons.Node.FS.File method)
  (SCons.Node.FS.RootDir method)
prepare_dependencies() (SCons.Node.FS.FileBuildInfo method)
  (SCons.SConf.SConfBuildTask method)
  (SCons.Script.Main.BuildTask method)
  (SCons.Script.Main.CleanTask method)
  (SCons.Script.Main.QuestionTask method)
  (SCons.Taskmaster.AlwaysTask method)
  (SCons.Taskmaster.OutOfDateTask method)
  (SCons.Taskmaster.Task method)
Prepend() (SCons.Environment.Base method)
  (SCons.Environment.OverrideEnvironment method)
  (SCons.Script.SConscript.SConsEnvironment method)
PrependENVPath() (SCons.Environment.Base method)
PrependLIBS() (SCons.SConf.CheckContext method)
PrependPath() (in module SCons.Util.envs)
PrependUnique() (SCons.Environment.Base method)
PreProcessor (class in SCons.cpp)
  (SCons.Node.FS.Base attribute)
  (SCons.Node.FS.Dir attribute)
  (SCons.Node.FS.Entry attribute)
  (SCons.Node.FS.File attribute)
  (SCons.Node.FS.RootDir attribute)
  (SCons.Node.Node attribute)
preserve_unknown_options (SCons.Script.SConf.SConfBuildInfo method)
presub_lines() (SCons.Action._ActionAction method)
  (SCons.Action.ActionBase method)
  (SCons.Action.CommandGeneratorAction method)
  (SCons.Action.FunctionAction method)
  (SCons.Action.LazyAction method)
  (SCons.Action.ListAction method)
prev (SCons.Script.Main.Progressor attribute)
print_cmd_line() (SCons.Action._ActionAction method)
  (SCons.Action.FunctionAction method)
  (SCons.Action.LazyAction method)
print_help() (SCons.Script.SConf.SConfBuildInfo method)
print_it (SCons.Util.DisplayEngine attribute)
print_local_option_help() (SCons.Script.SConf.SConfBuildInfo method)
print_time() (in module SCons.Util)
print_topics() (in module SCons.Util)
print_usage() (SCons.Script.Interactive.SConsInteractiveCmd method)
print_tree() (in module SCons.Util)
print_version()
  (SCons.Script.SConsOptions.SConsOptionParser method)
PrintHelp() (in module SCons.Script.Main)
  (SCons.Action.LazyAction method)
  (SCons.Script.SConsOptions.SConsOption method)
process_contents() (SCons.cpp.DumbPreProcessor method)
  (SCons.cpp.PreProcessor method)
  (SCons.Scanner.C.SConsCPPConditionalScanner method)
  (SCons.Scanner.C.SConsCPPScanner method)
process_file() (SCons.cpp.DumbPreProcessor method)
  (SCons.cpp.PreProcessor method)
  (SCons.Scanner.C.SConsCPPConditionalScanner method)
  (SCons.Scanner.C.SConsCPPScanner method)
process_warn_strings() (in module SCons.Warnings)
processDefines() (in module SCons.Defaults)
ProgramScanner() (in module SCons.Scanner.Prog)
Progress() (in module SCons.Script.Main)
Progressor (class in SCons.Script.Main)
prompt (SCons.Script.Interactive.SConsInteractiveCmd attribute)
Proxy (class in SCons.Util)
  (SCons.Node.FS.Base attribute)
  (SCons.Node.FS.Dir attribute)
  (SCons.Node.FS.Entry attribute)
  (SCons.Node.FS.File attribute)
  (SCons.Node.FS.RootDir attribute)
  (SCons.Node.Node attribute)
  (SCons.Node.Node attribute)
  (SCons.Node.FS.Base method)
  (SCons.Node.FS.Dir method)
  (SCons.Node.FS.Entry method)
  (SCons.Node.FS.File method)
  (SCons.Node.FS.RootDir method)
  (SCons.Node.FS.Base method)
  (SCons.Node.FS.Dir method)
  (SCons.Node.FS.Entry method)
  (SCons.Node.FS.File method)
  (SCons.Node.FS.RootDir method)
  (SCons.Node.FS.Base attribute)
  (SCons.Node.FS.Dir attribute)
  (SCons.Node.FS.Entry attribute)
  (SCons.Node.FS.File attribute)
  (SCons.Node.FS.RootDir attribute)
  (SCons.Node.Node attribute)
python_version_deprecated() (in module SCons.Script.Main)
python_version_string() (in module SCons.Script.Main)
python_version_unsupported() (in module SCons.Script.Main)
PythonVersionWarning
Q
QuestionTask (class in SCons.Script.Main)
quote_spaces() (in module SCons.Subst)
R
raise_exception() (in module SCons.Subst)
raise_exception_on_error
  (SCons.Script.SConsOptions.SConsOptionParser attribute)
RCScan() (in module SCons.Scanner.RC)
rdir() (SCons.Node.FS.Dir method)
  (SCons.Node.FS.RootDir method)
RDirs() (SCons.Node.FS.Base method)
  (SCons.Node.FS.Dir method)
  (SCons.Node.FS.Entry method)
  (SCons.Node.FS.File method)
  (SCons.Node.FS.RootDir method)
read() (SCons.Node.FS.File attribute)
  (SCons.Node.FS.Base method)
  (SCons.Node.FS.Dir method)
  (SCons.Node.FS.Entry method)
  (SCons.Node.FS.File method)
  (SCons.Node.FS.RootDir method)
read_file() (SCons.cpp.DumbPreProcessor method)
  (SCons.cpp.PreProcessor method)
Serial (class in SCons.Taskmaster.Job)
set_action_list() (SCons.Executor.Executor method)
  (SCons.Executor.Null method)
  (SCons.Node.FS.Base method)
  (SCons.Node.FS.Dir method)
  (SCons.Node.FS.Entry method)
  (SCons.Node.FS.File method)
  (SCons.Node.FS.RootDir method)
set_build_result() (SCons.SConf.SConfBuildInfo method)
set_conflict_handler() (SCons.Script.SConsOptions.SConsOptionGroup method)
  (SCons.Script.SConsOptions.SConsOptionParser method)
set_default() (SCons.Script.SConsOptions.SConsOptionParser method)
set_defaults() (SCons.Script.SConsOptions.SConsOptionParser method)
set_description() (SCons.Script.SConsOptions.SConsOptionGroup method)
  (SCons.Script.SConsOptions.SConsOptionParser method)
set_diskcheck() (in module SCons.Node.FS)
set_duplicate() (in module SCons.Node.FS)
set_entry() (SCons.SConsign.Base method)
  (SCons.SConsign.DB method)
  (SCons.SConsign.Dir method)
  (SCons.SConsign.DirFile method)
  (SCons.Node.FS.Base method)
  (SCons.Node.FS.Dir method)
  (SCons.Node.FS.Entry method)
  (SCons.Node.FS.File method)
  (SCons.Node.FS.RootDir method)
set_hash_format() (in module SCons.Util.hashes)
set_local() (SCons.Node.FS.Base method)
  (SCons.Node.FS.Dir method)
  (SCons.Node.FS.Entry method)
  (SCons.Node.FS.File method)
  (SCons.Node.FS.RootDir method)
set_long_opt_delimiter() (SCons.Script.SConsOptions.SConsIndentedHelpFormatter method)
set_max_drift() (SCons.Node.FS.FS method)
set_missing_sconscript_error() (in module SCons.Script)
set_mode() (SCons.Util.DisplayEngine method)
  (SCons.Node.FS.Base method)
  (SCons.Node.FS.Dir method)
  (SCons.Node.FS.Entry method)
  (SCons.Node.FS.File method)
  (SCons.Node.FS.RootDir method)
  (SCons.Node.FS.Base method)
  (SCons.Node.FS.Dir method)
  (SCons.Node.FS.Entry method)
  (SCons.Node.FS.File method)
  (SCons.Node.FS.RootDir method)
set_option() (SCons.Script.SConsOptions.SConsValues method)
set_parser() (SCons.Script.SConsOptions.SConsIndentedHelpFormatter method)
(SCons.Node.FS.Base method)
(SCons.Node.FS.Dir method)
(SCons.Node.FS.Entry method)
(SCons.Node.FS.File method)
(SCons.Node.FS.RootDir method)

set_process_default_values()
(SCons.Script.SConsOptions.SConsOptionParser method)

(SCons.Node.FS.Base method)
(SCons.Node.FS.Dir method)
(SCons.Node.FS.Entry method)
(SCons.Node.FS.File method)
(SCons.Node.FS.RootDir method)

set_SConstruct_dir() (SCons.Node.FS.FS method)

set_short_opt_delimiter() (SCons.Script.SConsOptions.SConsIndentedHelpFormatter method)

(SCons.Node.FS.Base method)
(SCons.Node.FS.Dir method)
(SCons.Node.FS.Entry method)
(SCons.Node.FS.File method)
(SCons.Node.FS.RootDir method)

(SCons.Node.FS.Base method)
(SCons.Node.FS.Dir method)
(SCons.Node.FS.Entry method)

set_suffix() (SCons.Builder.BuilderBase method)

set_title()
(SCons.Script.SConsOptions.SConsOptionGroup method)

set_usage()
(SCons.Script.SConsOptions.SConsOptionParser method)

SetAllowableExceptions() (in module SCons.Subst)
SetBuildType() (in module SCons.SConf)
SetCacheMode() (in module SCons.SConf)


getDefault() (SCons.Builder.CallableSelector method)

getDefault() (SCons.Builder.DictCmdGenerator method)

getDefault() (SCons.Builder.DictEmitter method)

getDefault() (SCons.Builder.OverrideWarner method)

SetDefault() (SCons.Environment.Base method)

getDefault() (SCons.Environment.Base method)

SetDefault() (SCons.Environment.BuilderDict method)

SetDefault() (SCons.Environment.OverrideEnvironment method)

getDefault() (SCons.Environment.OverrideEnvironment method)

getDefault() (SCons.Environment.SubstitutionEnvironment method)

getDefault() (SCons.Node.Alias.AliasNameSpace method)

SetDefault() (SCons.Script.SConscript.SConsEnvironment method)

getDefault() (SCons.Script.SConscript.SConsEnvironment method)

getDefault() (SCons.Script.SConscript.SConsEnvironment method)

getDefault() (SCons.Util.Selector method)

SetLIBS() (SCons.SConf.CheckContext method)

setName() (SCons.Taskmaster.Job.NewParallel.Worker method)

getDefault() (SCons.Taskmaster.Job.Worker method)

SetOption() (in module SCons.Script.Main)

getDefault() (SCons.Taskmaster.Job.Worker method)
splitlines() (SCons.Subst.CmdStringHolder method)

src_builder() (SCons.Node.FS.Base method)
  (SCons.Node.FS.Dir method)
  (SCons.Node.FS.Entry method)
  (SCons.Node.FS.File method)
  (SCons.Node.FS.RootDir method)

src_builder_sources() (SCons.Builder.BuilderBase method)

src_suffixes() (SCons.Builder.BuilderBase method)
  (SCons.Builder.DictCmdGenerator method)

srcdir (SCons.Node.FS.Dir attribute)
  (SCons.Node.FS.Entry attribute)
  (SCons.Node.FS.File attribute)
  (SCons.Node.FS.RootDir attribute)

srcdir_duplicate() (SCons.Node.FS.Dir method)
  (SCons.Node.FS.RootDir method)

srcdir_find_file() (SCons.Node.FS.Dir method)
  (SCons.Node.FS.RootDir method)

srcdir_list() (SCons.Node.FS.Dir method)
  (SCons.Node.FS.RootDir method)

srcnode() (SCons.Node.FS.Base method)
  (SCons.Node.FS.Dir method)
  (SCons.Node.FS.Entry method)
  (SCons.Node.FS.File method)
  (SCons.Node.FS.RootDir method)

StackSizeWarning


standard_option_list
  (SCons.Script.SConsOptions.SConsOptionParser attribute)

start() (SCons.Taskmaster.Job.LegacyParallel method)
  (SCons.Taskmaster.Job.Worker method)

start_handling_includes()
  (SCons.cpp.DumbPreProcessor method)
  (SCons.cpp.PreProcessor method)

AppStateWarning

STORE_ACTIONS (SCons.Script.SConsOptions.SConsOption attribute)

  (SCons.Node.FS.Base attribute)
  (SCons.Node.FS.Dir attribute)
  (SCons.Node.FS.Entry attribute)
  (SCons.Node.FS.File attribute)
  (SCons.Node.FS.RootDir attribute)
  (SCons.Node.Node attribute)
ValueWithMemo() (in module SCons.Node.Python)
Variable_Method_Caller (class in SCons.Defaults)
Variables (class in SCons.Variables)
Variables() (in module SCons.Script)
variant_dir_target_climb() (SCons.Node.FS.FS method)
variant_dirs (SCons.Node.FS.Dir attribute)
(SCons.Node.FS.Entry attribute)
(SCons.Node.FS.File attribute)
(SCons.Node.FS.RootDir attribute)
VariantDir() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Node.FS.File attribute)
(SCons.Node.FS.RootDir attribute)
version_string() (in module SCons.Script.Main)
Virtualenv() (in module SCons.Platform.virtualenv)
(SCons.Node.FS.Base method)
(SCons.Node.FS.Dir method)
(SCons.Node.FS.Entry method)
(SCons.Node.FS.File method)
(SCons.Node.FS.RootDir method)
VisualCMissingWarning
VisualStudioMissingWarning
VisualVersionMismatch

wait()
wait_for_process_to_die() (in module SCons.Util)
(SCons.Node.FS.Base attribute)
(SCons.Node.FS.Dir attribute)
(SCons.Node.FS.Entry attribute)
(SCons.Node.FS.Base attribute)
(SCons.Node.FS.Dir attribute)
(SCons.Node.FS.Entry attribute)
(SCons.Node.FS.File attribute)
(SCons.Node.FS.RootDir attribute)
(SCons.Node.Node attribute)

Worker (class in SCons.Taskmaster.Job)
write() (in module SCons.SConsign)
(SCons.SConfStreamer method)
(SCons.SConsign.DB method)
(SCons.SConsign.DirFile method)
(SCons.Script.Main.Progressor method)
(SCons.Util.Unbuffered method)
write_scons_stats_file() (in module SCons.Util.stats)
write_lines() (SCons.SConf.Streamer method)
(SCons.Util.Unbuffered method)

Z
zfill() (SCons.Subst.CmdStringHolder method)
Python Module Index

SCons
SCons.Action
SCons.Builder
SCons.CacheDir
SCons.compat
SCons.Conftest
SCons.cpp
SCons.dblite
SCons.Debug
SCons.Defaults
SCons.Environment
SCons.Errors
SCons.Executor
SCons.exitfuncs
SCons.Memoize
SCons.Node
SCons.Node.Alias
SCons.Node.FS
SCons.Node.Python
SCons.PathList
SCons.Platform
SCons.Platform.aix
SCons.Platform.cygwin
SCons.Platform.darwin
SCons.Platform.hpux
SCons.Platform.irix
SCons.Platform.mingw
SCons.Platform.os2
SCons.Platform.posix
SCons.Platform.sunos
SCons.Platform.virtualenv
SCons.Platform.win32
SCons.Scanner
SCons.Scanner.C
SCons.Scanner.D
SCons.Scanner.Dir
SCons.Scanner.Fortran
SCons.Scanner.IDL
SCons.Scanner.Java
SCons.Scanner.LaTeX
SCons.Scanner.Prog
SCons.Scanner.RC
SCons.Scanner.SWIG
SCons.SConf
SCons.SConsign
SCons.Script
SCons.Script.Interactive
SCons.Script.Main
SCons.Script.SConscript
SCons.Script.SConsOptions
SCons.Subst
SCons.Taskmaster
SCons.Taskmaster.Job
SCons.Tool
SCons.Util
SCons.Util.envs
SCons.Util.filelock
SCons.Util.hashes
SCons.Util.sctypes
SCons.Util.stats
SCons.Variables
SCons.Variables.BoolVariable
SCons.Variables.EnumVariable
SCons.Variables.ListVariable
SCons.Variables.PackageVariable
SCons.Variables.PathVariable
SCons.Variables.PathVariable
SCons.Warnings