### Contents

**SCons API Documentation**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCons package</td>
<td>1</td>
</tr>
<tr>
<td>Module contents</td>
<td>1</td>
</tr>
<tr>
<td>Subpackages</td>
<td></td>
</tr>
<tr>
<td>SCons.Node package</td>
<td>1</td>
</tr>
<tr>
<td>Submodules</td>
<td>1</td>
</tr>
<tr>
<td>SCons.Node.Alias module</td>
<td>1</td>
</tr>
<tr>
<td>SCons.Node.FS module</td>
<td>7</td>
</tr>
<tr>
<td>SCons.Node.Python module</td>
<td>51</td>
</tr>
<tr>
<td>Module contents</td>
<td>57</td>
</tr>
<tr>
<td>SCons.Platform package</td>
<td>65</td>
</tr>
<tr>
<td>Submodules</td>
<td>65</td>
</tr>
<tr>
<td>SCons.Platform.aix module</td>
<td>65</td>
</tr>
<tr>
<td>SCons.Platform.cygwin module</td>
<td>65</td>
</tr>
<tr>
<td>SCons.Platform.darwin module</td>
<td>65</td>
</tr>
<tr>
<td>SCons.Platform.hpux module</td>
<td>65</td>
</tr>
<tr>
<td>SCons.Platform.irix module</td>
<td>65</td>
</tr>
<tr>
<td>SCons.Platform.mingw module</td>
<td>66</td>
</tr>
<tr>
<td>SCons.Platform.os2 module</td>
<td>66</td>
</tr>
<tr>
<td>SCons.Platform.posix module</td>
<td>66</td>
</tr>
<tr>
<td>SCons.Platform.sunos module</td>
<td>66</td>
</tr>
<tr>
<td>SCons.Platform.virtualenv module</td>
<td>66</td>
</tr>
<tr>
<td>SCons.Platform.win32 module</td>
<td>67</td>
</tr>
<tr>
<td>Module contents</td>
<td>67</td>
</tr>
<tr>
<td>SCons.Scanner package</td>
<td>68</td>
</tr>
<tr>
<td>Submodules</td>
<td>68</td>
</tr>
<tr>
<td>SCons.Scanner.C module</td>
<td>68</td>
</tr>
<tr>
<td>SCons.Scanner.D module</td>
<td>71</td>
</tr>
<tr>
<td>SCons.Scanner.Dir module</td>
<td>72</td>
</tr>
<tr>
<td>SCons.Scanner.Fortran module</td>
<td>72</td>
</tr>
<tr>
<td>SCons.Scanner.IDL module</td>
<td>73</td>
</tr>
<tr>
<td>SCons.Scanner.Java module</td>
<td>73</td>
</tr>
<tr>
<td>SCons.Scanner.LaTeX module</td>
<td>73</td>
</tr>
<tr>
<td>SCons.Scanner.Prog module</td>
<td>75</td>
</tr>
<tr>
<td>SCons.Scanner.RC module</td>
<td>75</td>
</tr>
<tr>
<td>SCons.Scanner.SWIG module</td>
<td>75</td>
</tr>
<tr>
<td>Module contents</td>
<td>75</td>
</tr>
<tr>
<td>SCons.Script package</td>
<td>79</td>
</tr>
<tr>
<td>Submodules</td>
<td>79</td>
</tr>
<tr>
<td>SCons.Script.Interactive module</td>
<td>79</td>
</tr>
<tr>
<td>SCons.Script.Main module</td>
<td>80</td>
</tr>
<tr>
<td>SCons.Script.SConsOptions module</td>
<td>87</td>
</tr>
<tr>
<td>SCons.Script.SConscript module</td>
<td>92</td>
</tr>
<tr>
<td>Module contents</td>
<td>98</td>
</tr>
<tr>
<td>SCons.Taskmaster package</td>
<td>99</td>
</tr>
<tr>
<td>Submodules</td>
<td>99</td>
</tr>
<tr>
<td>SCons.Taskmaster.Job module</td>
<td>99</td>
</tr>
<tr>
<td>Module contents</td>
<td>103</td>
</tr>
<tr>
<td>SCons.Tool package</td>
<td>109</td>
</tr>
<tr>
<td>Module contents</td>
<td>109</td>
</tr>
<tr>
<td>SCons.Variables package</td>
<td>111</td>
</tr>
<tr>
<td>Submodules</td>
<td>111</td>
</tr>
<tr>
<td>SCons.Variables.BoolVariable module</td>
<td>111</td>
</tr>
<tr>
<td>SCons.Variables.EnumVariable module</td>
<td>111</td>
</tr>
<tr>
<td>SCons.Variables.ListVariable module</td>
<td>112</td>
</tr>
<tr>
<td>SCons.Variables.PackageVariable module</td>
<td>113</td>
</tr>
<tr>
<td>SCons.Variables.PathVariable module</td>
<td>113</td>
</tr>
<tr>
<td>Module contents</td>
<td>114</td>
</tr>
<tr>
<td>SCons.compat package</td>
<td>116</td>
</tr>
<tr>
<td>Module contents</td>
<td>116</td>
</tr>
<tr>
<td>Submodules</td>
<td>116</td>
</tr>
<tr>
<td>SCons.Action module</td>
<td>116</td>
</tr>
<tr>
<td>SCons.Builder module</td>
<td>122</td>
</tr>
<tr>
<td>SCons.CacheDir module</td>
<td>127</td>
</tr>
</tbody>
</table>
Index

SCons.Conftest module 127  
SCons.Debug module 131  
SCons.Defaults module 131  
SCons.Environment module 133  
SCons.Errors module 145  
SCons.Executor module 146  
SCons.Memoize module 150  
SCons.PathList module 151  
SCons.SConf module 152  
SCons.SConsign module 157  
SCons.Subst module 158  
SCons.Util module 162  
SCons.Warnings module 169  
SCons.cpp module 173  
SCons.dblite module 175  
SCons.exitfuncs module 179  
SCons.compat package 179  
SCons.Node package 179  
Module contents 179  
Submodules 179  
SCons.Node.Alias module 179  
SCons.Node.FS module 186  
SCons.Node.Python module 230  
Module contents 236  
SCons.Platform package 244  
Submodules 244  
SCons.Platform.aix module 244  
SCons.Platform.cygwin module 244  
SCons.Platform.darwin module 244  
SCons.Platform.hpux module 244  
SCons.Platform.irix module 244  
SCons.Platform.mingw module 244  
SCons.Platform.os2 module 245  
SCons.Platform.posix module 245  
SCons.Platform.sunos module 245  
SCons.Platform.virtualenv module 245  
SCons.Platform.win32 module 245  
Module contents 246  
SCons.Scanner package 247  
Submodules 247  
SCons.Scanner.C module 247  
SCons.Scanner.D module 250  
SCons.Scanner.Dir module 251  
SCons.Scanner.Fortran module 251  
SCons.Scanner.IDL module 252  
SCons.Scanner.Java module 252  
SCons.Scanner.latex module 252  
SCons.Scanner.Prog module 254  
SCons.Scanner.RC module 254  
SCons.Scanner.SWIG module 254  
SCons.Script package 258  
SCons.Script.Interactive module 258  
SCons.Script.Main module 259  
SCons.Script.SConsOptions module 266  
SCons.Script.SConscript module 271  
SCons.Taskmaster package 278  
SCons.Taskmaster.Job module 278  
SCons.Taskmaster.Job module 278  
SCons.Taskmaster.Job module 278  
SCons.Taskmaster.Job module 278  
SCons.Taskmaster.Job module 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCons.Tasks task 278  
SCon...
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 tool. Missing information is due to shortcomings in the docstrings in the code, which are by no means complete (contributions welcomed!).

The target audience is developers working on SCons itself: what is “Public API” is not clearly delineated here. The interfaces available for use in SCons configuration scripts, 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)
class Attrs
    Bases: object
    shared
BuildInfo
Decider(function)
GetTag(key)
    Return a user-defined tag.
NodeInfo
Tag(key, value)
    Add a user-defined tag.
_add_child(collection, set, child)
    Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.
_children_get()
_children_reset()
_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)
  Adds prerequisites.
add_source (source)
  Adds sources.
add_to_implicit (deps)
add_to_waiting_parents (node)
  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)
add_wkid (wkid)
  Add a node to the list of kids waiting to be evaluated.
all_children (scan=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
bin
build ()
A “builder” for aliases.
builder
builder_set (builder)
built ()
Called just after this node is successfully built.
cached
changed (node=None, allowcache=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()
cached_since_last_build
check_attributes (name)
  Simple API to check if the node.attributes for name has been set.
children (scan=1)
  Return a list of the node’s direct children, minus those that are ignored by this node.
children_are_up_to_date ()
  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 ()
  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 ()
convert ()
del_binfo ()
    Delete the build info from this node.
depends
depends_set
disambiguate (must_exist=None)
env
env_set (env, safe=0)
executor
executor_cleanup ()
    Let the executor clean up any cached information.
exists ()
    Does this 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=1)
    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 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()

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.

has_explicit_builder()

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()

Returns true if this node is an conftest node

is_derived()

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

isLiteral()

Always pass the string representation of a Node to the command interpreter literally.

is_sconscript()

Returns true if this node is an sconscript

is_under(dir)

is_up_to_date()

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 ()
  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 ()
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.
new_binfo ()
new_ninfo ()
ninfo
nocache
noclean
postprocess ()
  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 ()
  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 ()
  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 ()
  Remove cached executor; forces recompute when needed.
retrieve_from_cache ()
  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.

do thread unsafe stuff in built().

rexists ()

Does this node exist locally or in a repository?

scan ()

Scan this node’s dependents for implicit dependencies.

scanner_key ()

sconsign ()

An Alias is not recorded in .sconsign files

select_scanner (scanscaner)

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=1)

Set the Node’s always_build value.

set_executor (executor)

Set the action executor for this node.

set_explicit (is_explicit)

set_nocache (nocache=1)

Set the Node’s nocache value.

set_noclean (noclean=1)

Set the Node’s noclean value.

set_precious (precious=1)

Set the Node’s precious value.

set_pseudo (pseudo=True)

Set the Node’s precious value.

set_specific_source (source)

set_state (state)

side_effect

side_effects

sources

sources_set

state

store_info

str_for_display ()

target_peers

visited ()

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: SCons.Node.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)

Restore the attributes from a pickled state.

bact

bactsig

bdepends

bdependssigs

bimplicit

bimplicitssigs
bsources
bsourceigs
current_version_id = 2
merge (other)
    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: collections.UserDict
    Alias (name, **kw)
    _abc_impl = <_abc._abc_data object>
clear () → None. Remove all items from D.
copy ()
clasmethod fromkeys (iterable, value=None)
ge (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
    __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)
    Restore the attributes from a pickled state.
convert (node, val)
csig
current_version_id = 2
field_list = ['csig']
format (field_list=None, names=0)
merge (other)
    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)

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

class Attrs
    Bases: object
    shared
BuildInfo
    alias of SCons.Node.BuildInfoBase

Decider (function)
GetTag (key)
    Return a user-defined tag.

NodeInfo
    alias of SCons.Node.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)
    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’, ’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

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

_abspath

_add_child (collection, set, child)
    Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.

_children_get ()

_children_reset ()

_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=True, source=False, strings=False)

_labspath

_local

_memo

_path

_path_elements

_proxy

_save_str ()

_specific_sources
add_dependency(depend)
   Adds dependencies.
add_ignore(depend)
   Adds dependencies to ignore.
add_prerequisite(prerequisite)
   Adds prerequisites
add_source(source)
   Adds sources.
add_to_implicit(deps)
add_to_waiting_parents(node)
   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)
add_wkid(wkid)
   Add a node to the list of kids waiting to be evaluated
all_children(scan=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)
built()
   Called just after this node is successfully built.
cached
changed(node=None, allowcache=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=1)
   Return a list of the node’s direct children, minus those that are ignored by this node.
children_are_up_to_date()
   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 ()
    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 ()
cwd
del_binfo ()
    Delete the build info from this node.
depends
depends_set
dir
disambiguate (must_exist=None)
duplicate
env
env_set (env, safe=0)
extecutor
executor_cleanup ()
    Let the executor clean up any cached information.
exists ()
    Does this 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 ()
get_csig ()
get_dir ()
get_env ()
get_env_scanner (env, kw={})
get_executor (create=1)
    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_relpather
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()
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()
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 ()
  Returns true if this node is an conftest node

is-derived ()
  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 ()
  Always pass the string representation of a Node to the command interpreter literally.

is_sconscript ()
  Returns true if this node is an sconscript

is_under (dir)

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

isdir ()
isfile ()
islink ()
linked
lstat ()

make_ready ()
  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 ()
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.

name

new_binfo ()
new_ninfo ()
ninfo
nocache
noclean

postprocess ()
  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 ()
Try to push a node into a cache

ref_count
release_target_info()
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.
remove()
reset_executor()
Remove cached executor; forces recompute when needed.
retrieve_from_cache()
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()
A Node.FS.Base object’s string representation is its path name.
sbuilder
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=1)
Set the Node’s always_build value.
set_executor(executor)
Set the action executor for this node.
set_explicit(is_explicit)
set_local()
set_nocache(nocache=1)
Set the Node’s nocache value.
set_noclean(noclean=1)
Set the Node’s noclean value.
set_precious(precious=1)
Set the Node’s precious value.
set_pseudo(pseudo=True)
Set the Node’s precious value.
set_specific_source(source)
set_src_builder(builder)
Set the source code builder for this node.
set_state(state)
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()
    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: SCons.Node.FS.Base
    A class for directories in a file system.
class Attrs
    Bases: object
    shared
BuildInfo
    alias of SCons.Node.FS.DirBuildInfo
Decider(function)
Dir(name, create=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 SCons.Node.FS.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)
    Add a user-defined tag.
_Rfindalldirs_key(pathlist)
__clearRepositoryCache(duplicate=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)

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

_abspath

_add_child (collection, set, child)
  Adds 'child' to 'collection', first checking 'set' to see if it's already present.

_children_get ()

_children_reset ()

_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=True, source=False, strings=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 ()
  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

addRepository (dir)

add_dependency (depend)
  Adds dependencies.

add_ignore (depend)
  Adds dependencies to ignore.

add_prerequisite (prerequisite)
  Adds prerequisites

add_source (source)
  Adds sources.

add_to_implicit (deps)
add_to_waiting_parents (node)
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)
add_wkid (wkid)
Add a node to the list of kids waiting to be evaluated

all_children (scan=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)
A null “builder” for directories.
builder
builder_set (builder)
built ()
Called just after this node is successfully built.
cached
cachedir_csig
cachesig
changed (node=None, allowcache=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=1)
Return a list of the node’s direct children, minus those that are ignored by this node.
children_are_up_to_date ()
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 ()
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 ()
contentsig
cwd
del_binfo ()
Delete the build info from this node.
depends
depends_set
dir
dir_on_disk (name)
dirname
SCons API Documentation

```python
disambiguate(must_exist=None)
diskcheck_match()
do_duplicate(src)
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 rentry_exists_on_disk
entry_abspath(name)
entry_path(name)
entry_tpath(name)
env
env_set(env, safe=0)
ex
executor
executor_cleanup()
    Let the executor clean up any cached information.
exists()
    Does this 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=1)
  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_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_tpath ()
gettext ()
getsize ()
glob (pathname, ondisk=True, source=False, strings=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()
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()
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()
Returns true if this node is an conftest node
is_derived()
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()
Always pass the string representation of a Node to the command interpreter literally.

is_sconscript() Returns true if this node is an sconscript

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

isdir()
isfile()islink()

link(srcdir, duplicate)
   Set this directory as the variant directory for the supplied source directory.

linked

lstat()

make_ready()
   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()
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.

ame

new_binfo()

new_ninfo()
ninfo

cache

clean

on_disk_entries

postprocess()

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()
   Try to push a node into a cache

dir()

dir

dir

dir

dir

rel_path(other)
   Return a path to “other” relative to this directory.

release_target_info()
   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.
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()
Remove cached executor; forces recompute when needed.
retrieve_from_cache()
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()
A Node.FS.Base object’s string representation is its path name.
sbuilder
scan()
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.
ssearched
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=1)
Set the Node’s always_build value.
set_executor(executor)
Set the action executor for this node.
set_explicit(is_explicit)
set_local()
set_nocache(nocache=1)
Set the Node’s nocache value.
set_noclean(noclean=1)
Set the Node’s noclean value.
set_precious(precious=1)
Set the Node’s precious value.
set_pseudo(pseudo=True)
Set the Node’s precious value.
set_specific_source(source)
set_src_builder(builder)
Set the source code builder for this node.
set_state(state)
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()
    Called just after this node has been visited (with or without a build).
waiting_parents
waiting_s_e
walk(func, arg)
    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: SCons.Node.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)
        Restore the attributes from a pickled state.
bact
bactsig
bdepends
bdependsigs
bimplicit
bimplicitigs
bsources
bsourcesigs
merge(other)
    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
    __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)
    Restore the attributes from a pickled state. The version is discarded.
    convert (node, val)
    current_version_id = 2
    format (field_list=None, names=0)
    fs = None
    merge (other)
    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)

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)
    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: SCons.Node.FS.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 SCons.Node.BuildInfoBase
        Decider (function)
        GetTag (key)
        Return a user-defined tag.
        NodeInfo
        alias of SCons.Node.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)
        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__()  
A Node.FS.Base object’s string representation is its path name.
_abspath
_add_child(collection, set, child)
  Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.
_children_get()
_children_reset()
_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=True, source=False, strings=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)
  Adds prerequisites
_add_source(source)
  Adds sources.
_add_to_implicit(deps)
_add_to_waiting_parents(node)
  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)
_add_wkid(wkid)
  Add a node to the list of kids waiting to be evaluated
_all_children(scan=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)
built ()

Called just after this node is successfully built.
cached
cachedir_csиг

cachesig
changed (node=None, allowcache=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=1)

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

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 ()

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 ()
contentsig
cwd
del_binfo ()

Delete the build info from this node.
depends
depends_set
dir
dirname
disambiguate (must_exist=None)
diskcheck_match ()
duplicate
entries
env
evil_set (env, safe=0)
exector
eexecutor_cleanups ()

Let the executor clean up any cached information.
exists ()
Does this 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=1)
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 ()
    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 ()
    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 ()
    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 ()
    Returns true if this node is an conftest node
is_derived ()
    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 ()
    Always pass the string representation of a Node to the command interpreter literally.
is_sconscript ()
    Returns true if this node is an sconscript
is_under (dir)
is_up_to_date ()
Default check for whether the Node is current: unknown Node subtypes are always out of date, so they will always get built.

```python
dir()
exists()
existslink()
existsfile()
lstat()
make_ready()
```

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.

```python
missing()
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.

```python
must_be_same(klass)
```

Called to make sure a Node is a Dir. Since we’re an Entry, we can morph into one.

```python
name
new_binfo()
ninfo
nocache
noclean
on_disk_entries
postprocess()
```

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

```python
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.

```python
prerequisites
pseudo
push_to_cache()
ref_count
rel_path(other)
release_target_info()
```

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()

```python
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.

retnode ()
    Remove cached executor; forces recompute when needed.

retrieve_from_cache ()
    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 ()
    We're a generic Entry, but the caller is actually looking for a File at this point, so morph into one.

root

rstr ()
    A Node.FS.Base object's string representation is its path name.

sbuilder

scan ()
    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=1)
    Set the Node's always_build value.

set_executor (executor)
    Set the action executor for this node.

set_explicit (is_explicit)

set_local ()

set_nocache (nocache=1)
    Set the Node's nocache value.

set_noclean (noclean=1)
    Set the Node's noclean value.

set_precious (precious=1)
    Set the Node's precious value.

set_pseudo (pseudo=True)
    Set the Node's precious value.

set_specific_source (source)

set_src_builder (builder)
    Set the source code builder for this node.

set_state (state)

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 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
variant_dirs
visited ()
  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: SCons.Util.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_repath ()
__get_rsrcdir ()
  Returns the directory containing the source node linked to this node via VariantDir(), or the directory of this node if not linked.
__get_rsrcnode ()
__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_sncnode ()
__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>, 'repath': <function EntryProxy.__get_repath>,
  'rsrcdir': <function EntryProxy.__get_rsrcdir>, 'rsrcpath': <function EntryProxy.__get_rsrcnode>, 'srcdir': <function EntryProxy.__get_srcdir>, 'srcpath': <function EntryProxy.__get_sncnode>, '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.

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: SCons.Node.FS.LocalFS

Dir (name, directory=None, create=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=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=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=True, source=True, strings=False, exclude=None, cwd=None)

    This is mainly a shim layer

PyPackageDir (modulename)

    Locate the directory of a given python module name

    For example scons might resolve to Windows: C:Python27Libsite-packages\scons-2.5.1 Linux: /usr/lib/scons

    This can be useful when we want to determine a toolpath based on a python module name

Repository (*dirs)

    Specify Repository directories to search.

VariantDir (variant_dir, src_dir, duplicate=True)

    Link the supplied variant directory to the source directory for purposes of building files.

_lookup (p, directory, fsclass, create=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=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)
exists (path)
get_max_drift ()
get_root (drive)

    Returns the root directory for the specified drive, creating it if necessary.

cgetcwd ()
getmtime (path)
getsize (path)
isdir (path)
isfile (path)
islink (path)
class SCons.Node.FS.File (name, directory, fs)

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

class Attr
   Bases: object
   shared

BuildInfo
   alias of SCons.Node.FS.FileBuildInfo

Decider (function)
   Dir (name, create=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 SCons.Node.FS.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)
     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__ ()

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

_abspath

_add_child (collection, set, child)

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 -)

Returns: dictionary of file->signature mappings

_children_get ()

_children_reset ()

_createDir ()

_func_exists

_func_get_contents

_func_is Derived

_func_exists

_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 -)

Returns: List of csigs for provided list of children

_get_scanner (env, initial_scanner, root_node_scanner, kw)

_get_str ()

_glob1 (pattern, ondisk=True, source=False, strings=False)

_labspath

_local

_memo

_morph ()

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)
   Adds prerequisites
add_source(source)
   Adds sources.
add_to_implicit(deps)
add_to_waiting_parents(node)
   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)
add_wkid(wkid)
   Add a node to the list of kids waiting to be evaluated
all_children(scan=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)
built()
   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=False)
   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)
changed_since_last_build
changed_state(target, prev_ni, repo_node=None)
changed_timestamp_match(target, prev_ni, repo_node=None)
   Return True if the timestamps don’t match or if there is no previous timestamp.
   Information about the node from the previous build:
   return:
changed_timestamp_newer(target, prev_ni, repo_node=None)
changed_timestamp_then_content(target, prev_ni, node=None)
   Used when decider 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*)

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=1)`
  Return a list of the node’s direct children, minus those that are ignored by this node.
- `children_are_up_to_date()`
  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()`
  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()`
- `contentsig`
- `convert_copy_attrs = ['bsources', 'bimplicit', 'bdepends', 'bact', 'bactsig', 'ninfo']`
- `convert_old_entry( old_entry )`
- `convert_sig_attrs = ['bsourcesigs', 'bimplicitSIGs', 'bdependsigs']`
- `cwd`
- `del_binfo()`
  Delete the build info from this node.
- `depends`
- `depends_set`
- `dir`
- `dirname`
- `disambiguate(must_exist=None)`
- `diskcheck_match()`
- `do_duplicate(src)`
- `duplicate entries env env_set(env, safe=0)`
- `executor`
- `executor_cleanup()`
  Let the executor clean up any cached information.
- `exists()`
  Does this 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
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.

get_content_hash () → str
Compute and return the hash for this file.

get_contents () → bytes
Return the contents of the file as bytes.

get_contents_sig ()
A helper method for get_cachedir_bsig.
It computes and returns the signature for this node’s contents.

get_csig () → str
Generate a node’s content signature.

get_dir ()

get_env ()

get_env_scanner (env, kw={})

get_executor (create=1)
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 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.

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_max_drift_csig () → Optional [str]
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_repath ()
    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 in text form.
    This 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.
get_timestamp () → int
get_tpath ()
getmtime ()
getsize ()
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.
has_explicit_builder ()
    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 ()
    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 ()
     Returns true if this node is a conftest node
 is_derived ()
     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 ()
     Always pass the string representation of a Node to the command interpreter literally.
 is_sconscript ()
     Returns true if this node is an sconscript
 is_under (dir)
 is_up_to_date ()
     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 ()
isfile ()
islink ()
linked
 lstat ()
 make_ready ()
     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 ()
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.
 name
 new_binfo ()
 new_ninfo ()
ninfo
 nocache
 noclean
 on_disk_entries
 postprocess ()
     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 ()
     Try to push the node into a cache
 ref_count
 rel_path (other)
 release_target_info ()
     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\(^\star\).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.
retry()
repositories
reset_executor()
  Remove cached executor; forces recompute when needed.
retrieve_from_cache()
  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?
rfile()
root
rstr()
  A Node.FS.Base object’s string representation is its path name.
sbuilder
scan()
  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=1)
  Set the Node’s always_build value.
set_executor(executor)
  Set the action executor for this node.
set_explicit(is_explicit)
set_local()
set_nocache(nocache=1)
  Set the Node’s nocache value.
set_noclean(noclean=1)
  Set the Node’s noclean value.
set_precious(precious=1)
  Set the Node’s precious value.
set_pseudo(pseudo=True)
  Set the Node’s precious value.
set_specific_source(source)
set_src_builder(builder)
  Set the source code builder for this node.
set_state (state)
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

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 ()
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: SCons.Node.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)
Restore the attributes from a pickled state.
bact
bactsig
bdepends
bdependsigs
bimplicit
bimplicitssigs
bsources
bsourcesigs
convert_from_sconsign (dir, name)

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 ()

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=0)
merge (other)

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 ()
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
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
   returns The node created from the found file.
class SCons.Node.FS.FileNodeInfo
__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)
   Restore the attributes from a pickled state.
convert (node, val)
csig
current_version_id = 2
field_list = ['csig', 'timestamp', 'size']
format (field_list=None, names=0)
fs = None
merge (other)
   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)
SCons API Documentation

SCons.Node.FS.LinkFunc (target, source, env)
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)
exists (path)
getmtime (path)
getsize (path)
isdir (path)
isfile (path)
islink (path)
link (src, dst)
listdir (path)
lstat (path)
makedirs (path, mode=511, exist_ok=False)
mkdir (path, mode=511)
on(path)
readlink (file)
rename (old, new)
scandir (path)
stat (path)
symlink (src, dst)
unlink (path)

SCons.Node.FS.LocalString (target, source, env)
SCons.Node.FS.MkdirFunc (target, source, env)
class SCons.Node.FS.RootDir (drive, fs)
Bases: SCons.Node.FS.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 SCons.Node.FS.DirBuildInfo

Decider (function)
Dir (name, create=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

42
alias of SCons.Node.FS.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)
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)
Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.

_children_get ()

_children_reset ()

_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=True, source=False, strings=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=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 ()
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)
add_dependency (depend)
    Adds dependencies.
add_ignore (depend)
    Adds dependencies to ignore.
add_prerequisite (prerequisite)
    Adds prerequisites
add_source (source)
    Adds sources.
add_to_implicit (deps)
add_to_waiting_parents (node)
    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)
add_wkid (wkid)
    Add a node to the list of kids waiting to be evaluated
all_children (scan=1)
    Return a list of all the node’s direct children.
alter_targets ()
    Return any corresponding targets in a variant directory.
always_build
attributes
build (**kw)
    A null “builder” for directories.
binfo
builder
builder_set (builder)
built ()
    Called just after this node is successfully built.
cashed
cachedir_csigit
changes (node=None, allowcache=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=1)
    Return a list of the node’s direct children, minus those that are ignored by this node.
children_are_up_to_date()
    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()
    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()

cwddel_binfo()
    Delete the build info from this node.
depends
depends_set
dir
dir_on_disk(name)
dirname
disambiguate(must_exist=None)
diskcheck_match()
do_duplicate(src)
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 rentry_exists_on_disk
entry_labspath(name)
entry_path(name)
entry_tpath(name)
env
eval_set(env, safe=0)
executor
executor_cleanup()
    Let the executor clean up any cached information.
exists()
    Does this 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=1)
    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 ()
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=True, source=False, strings=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 ()
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 ()
   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 ()
   Returns true if this node is a conftest node
is_derived ()
   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 ()
   Always pass the string representation of a Node to the command interpreter literally.

is_sconscript ()

is_under (dir)

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

isdir ()
isfile ()
islink ()

link (srcdir, duplicate)
   Set this directory as the variant directory for the supplied source directory.
linked
lstat ()

make_ready ()
   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 ()
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.

name
new_binfo ()
new_ninfo ()
ninfo
nocache
noclean
on_disk_entries
path
postprocess ()
   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 ()
  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 ()
  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 ()
  Remove cached executor; forces recompute when needed.
retrieve_from_cache ()
  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 ()
  A Node.FS.Base object’s string representation is its path name.
sbuilder
scan ()
  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=1)
Set the Node’s always_build value.
set_executor (executor)
Set the action executor for this node.
set_explicit (is_explicit)
set_local ()
set_nocache (nocache=1)
Set the Node’s nocache value.
set_noclean (noclean=1)
Set the Node’s noclean value.
set_precious (precious=1)
Set the Node’s precious value.
set_pseudo (pseudo=True)
Set the Node’s precious value.
set_specific_source (source)
set_src_builder (builder)
Set the source code builder for this node.
set_state (state)
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 ()
Called just after this node has been visited (with or without a build).
waiting_parents
waiting_s_e
walk (func, arg)
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**

```python
SCons.Node.FS.UnlinkFunc (target, source, env)
```

**class SCons.Node.FS._Null**

Bases: object

SCons.Node.FS._classEntry

alias of SCons.Node.FS.Entry

SCons.Node.FS._copy_func (fs, src, dest)

SCons.Node.FS._hardlink_func (fs, src, dst)

SCons.Node.FS._my_normcase (x)

SCons.Node.FS._my_splitdrive (p)

SCons.Node.FS._softlink_func (fs, src, dst)

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)

SCons.Node.FS.ignore_diskcheck_match (node, predicate, errorfmt)

SCons.Node.FS.initialize_do_splitdrive ()

SCons.Node.FS.invalidate_node_memos (targets)

Invalidates 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)

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)

SCons.Node.FS.set_duplicate (duplicate)

SCons.Node.FS invalidate_node_memos (targets)

SCons.Node.FS.get_MkdirBuilder ()

SCons.Node.FS.get_default_fs ()

SCons.Node.FS.has_glob_magic (s)

SCons.Node.FS.ignore_diskcheck_match (node, predicate, errorfmt)

SCons.Node.FS.initialize_do_splitdrive ()

SCons.Node.FS.invalidate_node_memos (targets)

Invalidates 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)

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)

SCons.Node.FS.set_duplicate (duplicate)

SCons.Node.FS invalidate_node_memos (targets)

SCons.Node.Fs.UnlinkFunc (target, source, env)

**class SCons.Node.FS._Null**

Bases: object

SCons.Node.FS._classEntry

alias of SCons.Node.FS.Entry

SCons.Node.FS._copy_func (fs, src, dest)

SCons.Node.FS._hardlink_func (fs, src, dst)

SCons.Node.FS._my_normcase (x)

SCons.Node.FS._my_splitdrive (p)

SCons.Node.FS._softlink_func (fs, src, dst)

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)

SCons.Node.FS.ignore_diskcheck_match (node, predicate, errorfmt)

SCons.Node.FS.initialize_do_splitdrive ()

SCons.Node.FS.invalidate_node_memos (targets)

Invalidates 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)

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)

SCons.Node.FS.set_duplicate (duplicate)

SCons.Node.Fs.UnlinkFunc (target, source, env)
shared
BuildInfo
Decider (function)
GetTag (key)
    Return a user-defined tag.
NodeInfo
Tag (key, value)
    Add a user-defined tag.
_add_child (collection, set, child)
    Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.
_children_get ()
_children_reset ()
_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)
    Adds prerequisites
add_source (source)
    Adds sources.
add_to_implicit (deps)
add_to_waiting_parents (node)
    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)
add_wkid (wkid)
    Add a node to the list of kids waiting to be evaluated
all_children (scan=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)
built ()
    Called just after this node is successfully built.
cached
changed (node=None, allowcache=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()

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

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

clean_up_to_date ()
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 ()
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 ()
del_binfo ()
Delete the build info from this node.

depends
depends_set
disambiguate (must_exist=None)

e env_set (env, safe=0)
exec
executor
executor_cleanup ()
Let the executor clean up any cached information.
exists ()
Does this 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.
SCons API Documentation

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_cachepath ()
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=1)
    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 ()
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 ()
    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 ()
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 ()
Returns true if this node is a conftest node.

is_derived ()
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 ()
Always pass the string representation of a Node to the command interpreter literally.

is_sconscript ()
Returns true if this node is an sconscript.

is_under (dir)

is_up_to_date ()
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 ()
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 ()
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.

new_binfo ()
new_ninfo ()
ninfo
nocache
noclean
postprocess ()
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 ()

Try to push a node into a cache

read ()

Return the value. If necessary, the value is built.

ref_count

release_target_info ()

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 ()

Remove cached executor; forces recompute when needed.

retrieve_from_cache ()

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 ()

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=1)

Set the Node’s always_build value.

set_executor (executor)

Set the action executor for this node.

set_explicit (is_explicit)

set_nocache (nocache=1)

Set the Node’s nocache value.

set_noclean (noclean=1)

Set the Node’s noclean value.

set_precious (precious=1)

Set the Node’s precious value.

set_pseudo (pseudo=True)

Set the Node’s precious value.

set_specific_source (source)

set_state (state)

side_effect

side_effects

sources
sources_set
state
store_info
str_for_display()
target_peers
visited()
Called just after this node has been visited (with or without a build).
waiting_parents
waiting_s_e
wkids
write(built_value)
Set the value of the node.
class SCons.Node.Python.ValueBuildInfo
Bases: SCons.Node.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)
Restore the attributes from a pickled state.
bact
bactsig
bdepends
bdependentsigs
bimplicit
bimplicitargs
bsources
bsourcesigs
current_version_id = 2
merge(other)
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.
__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)
Restore the attributes from a pickled state.
convert(node, val)
csig
current_version_id = 2
field_list = ['csig']
format(field_list=None, names=0)
merge(other)
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)
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)
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 NodeInfo 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)
    Restore the attributes from a pickled state.
bact
  bactsig
  bdepins
  bdependsigs
  bimplicit
  bimplicitsigs
  bsources
  bsourcesigs
  current_version_id = 2
  merge (other)
    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 SCons.Node.BuildInfoBase
  Decider (function)
  GetTag (key)
    Return a user-defined tag.
  NodeInfo
    alias of SCons.Node.NodeInfoBase
  Tag (key, value)
    Add a user-defined tag.
  _add_child (collection, set, child)
    Adds 'child' to 'collection', first checking 'set' to see if it’s already present.
  _children_get ()
  _children_reset ()
  _func_exists
  _func_get_contents
  _func_is_derived
func_exists
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)
  Adds prerequisites
add_source(source)
  Adds sources.
add_to_implicit(deps)
add_to_waiting_parents(node)
  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)
add_wkid(wkid)
  Add a node to the list of kids waiting to be evaluated
all_children(scan=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)
built()
  Called just after this node is successfully built.
cached
changed(node=None, allowcache=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=1)
  Return a list of the node’s direct children, minus those that are ignored by this node.
children_are_up_to_date ()
    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 ()
    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 ()

del_binfo ()
    Delete the build info from this node.

depends

depends_set

disambiguate (must_exist=None)

env
    env_set (env, safe=0)

executor
    executor_cleanup ()
        Let the executor clean up any cached information.

eexists ()
    Does this 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 ()

get_csig ()

get_env ()

get_env_scanner (env, kw={})

get_executor (create=1)
    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 ()

get_target_scanner ()

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.

has_explicit_builder ()
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 ()
Returns true if this node is a conftest node

is_derived ()
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 ()
Always pass the string representation of a Node to the command interpreter literally.

is_sconscript ()
Returns true if this node is an sconscript
is_up_to_date ()
    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 ()
    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 ()
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.
new_binfo ()
new_ninfo ()
ninfo
nocache
noclean
postprocess ()
    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 ()
    Try to push a node into a cache
ref_count
release_target_info ()
    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 ()
    Remove cached executor; forces recompute when needed.
retrieve_from_cache ()
    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 ()
   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=1)
   Set the Node’s always_build value.
set_executor (executor)
   Set the action executor for this node.
set_explicit (is_explicit)
set_nocache (nocache=1)
   Set the Node’s nocache value.
set_noclean (noclean=1)
   Set the Node’s noclean value.
set_precious (precious=1)
   Set the Node’s precious value.
set_pseudo (pseudo=True)
   Set the Node’s precious value.
set_specific_source (source)
set_state (state)
side_effect
side_effects
sources
sources_set
state
store_info
target_peers
visited ()
   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)
   Restore the attributes from a pickled state. The version is discarded.
convert (node, val)
current_version_id = 2
format (field_list=None, names=0)
merge (other)
   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)
class SCons.Node.NodeList (initlist=None)
   Bases: collections.UserList
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 ()
S.pop() → 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 ()

SCons.Node.changed_since_last_build_alias (node, target, prev_ni, repo_node=None)
SCons.Node.changed_since_last_build_entry (node, target, prev_ni, repo_node=None)
SCons.Node.changed_since_last_build_node (node, target, prev_ni, repo_node=None)
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)
SCons.Node.changed_since_last_build_state_changed (node, target, prev_ni, repo_node=None)
SCons.Node.classname (obj)
SCons.Node.decide_source (node, target, prev_ni, repo_node=None)
SCons.Node.decide_target (node, target, prev_ni, repo_node=None)
SCons.Node.do_nothing (node, parent)
SCons.Node.do_nothing_node (node)
SCons.Node.exists_always (node)
SCons.Node.exists_base (node)
SCons.Node.exists_entry (node)
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)
SCons.Node.exists_none (node)
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)
SCons.Node.is_derived_node (node)

Returns true if this node is derived (i.e. built).
SCons.Node.is_derived_none (node)
SCons.Node.rexists_base (node)
SCons.Node.rexists_node (node)
SCons.Node.rexists_none (node)
SCons.Node.store_info_file (node)
SCons.Node.store_info_pass (node)
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)
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.irix.generate (env)

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)

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)

SCons.Platform.posix.exec_popen3 (l, env, stdout, stderr)

SCons.Platform.posix.exec_subprocess (l, env)

SCons.Platform.posix.generate (env)

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)

SCons.Platform.virtualenv module

“Platform” support for a Python virtualenv.

SCons.Platform.virtualenv.ImportVirtualenv (env)

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.Virtualenv ()

Returns path to the virtualenv home if scons is executing within a virtualenv or None, if not.

SCons.Platform.virtualenv._enable_virtualenv_default ()

SCons.Platform.virtualenv._ignore_virtualenv_default ()

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

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

SCons.Platform.virtualenv._inject_venv_variables (env)

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

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 API Documentation

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

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 PROCESSOR_ARCHITEW6432 or PROCESSOR_ARCHITECTURE environment variables).
SCons.Platform.win32.get_program_files_dir ()
SCons.Platform.win32.get_system_root ()
SCons.Platform.win32.piped_spawn (sh, escape, cmd, env, stdout, stderr)
SCons.Platform.win32.spawn (sh, escape, cmd, 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:

```
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:
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:

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.Scanner package

Submodules

SCons.Scanner.C module

Dependency scanner for C/C++ code.
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: SCons.cpp.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)
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)
do_define(t)
  Default handling of a #define line.
do_elif(t)
  Default handling of a #elif line.
do_else(t)
  Default handling of a #else line.
do_endif(t)
  Default handling of a #endif line.
do_if(t)
  Default handling of a #if line.
do_ifdef(t)
  Default handling of a #ifdef line.
do ifndef(t)
  Default handling of a #ifndef line.
do_import(t)
  Default handling of a #import line.
do_include(t)
  Default handling of a #include line.
do include_next(t)
  Default handling of a #include line.
do nothing(t)
  Null method for when we explicitly want the action for a specific preprocessor directive to do nothing.
do undef(t)
  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)

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)

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 ()
   Pops the previous dispatch table off the stack and makes it the current one.
save ()
   Pushes the current dispatch table on the stack and re-initializes the current dispatch table to the default.
scons_current_file (t)
start_handling_includes (t=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)
   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.
   recurse_nodes (nodes)
   select (node)
class SCons.Scanner.C.SConsCPPScanner (*args, **kwargs)
   Bases: SCons.cpp.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)
      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)
   do_define (t)
      Default handling of a #define line.
   do_elif (t)
      Default handling of a #elif line.
   do_else (t)
      Default handling of a #else line.
   do_endif (t)
      Default handling of a #endif line.
   do_if (t)
      Default handling of a #if line.
   do ifndef (t)
      Default handling of a #ifndef line.
   do ifndef (t)
      Default handling of a #ifndef line.
   do import (t)
      Default handling of a #import line.
   do include (t)
Default handling of a #include line.
do_include_next (t)
Default handling of a #include line.
do_nothing (t)
Null method for when we explicitly want the action for a specific preprocessor directive to do nothing.
do_undef (t)
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.
final_result (fname)
find_include_file (t)
Finds the #include file for a given preprocessor tuple.
initialize_result (fname)
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)
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 ()
Pops the previous dispatch table off the stack and makes it the current one.
save ()
Pushes the current dispatch table on the stack and re-initializes the current dispatch table to the default.
scons_current_file (t)
start_handling_includes (t=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)
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 block 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.
recursion_nodes (nodes)
select (node)
SCons.Scanner.C.dictify_CPPDEFINES (env) → dict
Returns CPPDEFINES converted to a dict.
SCons.Scanner.D module
Scanner for the Digital Mars "D" programming language.
SCons API Documentation

Coded by Andy Friesen, 17 Nov 2003

class SCons.Scanner.D.D
    Bases: SCons.Scanner.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.

    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)
    add_skey (skey)
    Add a skey to the list of skeys

    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)

SCons.Scanner.Dir.scan_in_memory (node, env, path=())
    “Scans” a Node.FS.Dir for its in-memory entries.

SCons.Scanner.Dir.scan_on_disk (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: SCons.Scanner.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.
- **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)
add_skey(skey)

    Add a skey to the list of skeys

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

### SCons.Scanner.Fortran module

Dependency scanner for Fortran USE & INCLUDE statements.

```python
SCons.Scanner.Fortran.FortranScan(path_variable='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.

```python
SCons.Scanner.IDL.IDLScan ()
    Return a prototype Scanner instance for scanning IDL source files
```

### SCons.Scanner.Java module

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

```python
SCons.Scanner.Java._collect_classes(classlist, dirname, files)
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
```

73
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: SCons.Scanner.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
$ 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 "inputlisting" 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[\texttt{class}|\texttt{style}]{}/

FIXME: also look for the argument of \texttt{bibliographystyle}{}/

```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
_latex_names(include_type, filename)
static _recurse_all_nodes(nodes)
static _recurse_no_nodes(nodes)
add_scanner(skey, scanner)
add_skey(skey)

Add a skey to the list of skews

canonical_text(text)

Standardize an input TeX-file contents.
```

**Currently:**

```
env_variables = [TEXINPUTS, BIBINPUTS, BSTINPUTS, INDEXSTYLE]
find_include(include, source_dir, path)
geet_skeys(env=None)
keyword_paths = ['addbibresource': 'BIBINPUTS', 'addglobalbib': 'BIBINPUTS', 'addsectionbib': 'BIBINPUTS', 'bibliography': 'BIBINPUTS', 'bibliographystyle': 'BSTINPUTS', 'include': 'TEXINPUTS', 'includegraphics': 'TEXINPUTS', 'input': 'TEXINPUTS', 'inputlisting': 'TEXINPUTS', 'makeindex': 'INDEXSTYLE', 'usepackage': 'TEXINPUTS']
path(env, dir=None, target=None, source=None)
scan(node, subdir=')
```
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)
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 SCons.Scanner.LaTeX._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 SCons.Scanner.ScannerBase
class SCons.Scanner.Classic (name, suffixes, path_variable, regex, *args, **kwargs)
   Bases: SCons.Scanner.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.

```python
static _recurse_all_nodes (nodes)
static _recurse_no_nodes (nodes)
add_scanner (skey, scanner)
add_skey (skey)
    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)
```

**class SCons.Scanner.ClassicCPP (name, suffixes, path_variable, regex, *args, **kwargs)**

Bases: SCons.Scanner.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.

```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)
add_skey (skey)
    Add a skey to the list of skeys
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)
```

**class SCons.Scanner.Current (**args, **kwargs)**

Bases: SCons.Scanner.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).

```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
def _recurse_all_nodes(nodes):
    static

def _recurse_no_nodes(nodes):
    static

add_scanner(skey, scanner)
add_skey(skey)
get_skeys(env=None)
path(env, dir=None, target=None, source=None)
select(node)
```

```python
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='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. May also return a callable which is called with no args and returns the tuple (supporting Bindable class). 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)
add_skey (skey)

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: SCons.Scanner.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.)
SCons API Documentation

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

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

SCons.Script package

Submodules

SCons.Script.Interactive module

SCons interactive mode.

class SCons.Script.Interactive.SConsInteractiveCmd (**kw)
    Bases: cmd.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)
    _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)
    completedefault (*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)
        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)
    do_build (argv)
        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)
        exit Exit SCons interactive mode.
    do_help (argv)
        help [COMMAND] Prints help for the specified COMMAND. ‘h’ and ‘?’ are synonyms.
    do_shell (argv)
        shell [COMMANDLINE] Execute COMMANDLINE in a subshell. ‘sh’ and ‘!’ are synonyms.
    do_version (argv)
        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.
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.
prenocmd (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)

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: SCons.Taskmaster.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 ()
   display (message)
      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=2)
**exc_clear ()**
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)**
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 ()**
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 ()**
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 ()**
Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s callback methods.

**fail_continue ()**
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 ()**
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 ()**
Make a task ready for execution.

**make_ready_all ()**
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 ()
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='node')

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

LOGGER = None
_abc_impl = <_abc._abc_data object>
_clean_targets (remove=True)
_exception_raise ()
  Raises a pending exception that was recorded while getting a Task ready for execution.
_get_files_to_clean ()
_no_exception_to_raise ()
display (message)
  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 ()
  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)
  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 ()
  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 ()
  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 ()
  Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s callback methods.
fail_continue ()
  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()
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().

fs_delete(path, pathstr, remove=True)

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

make_ready()
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()
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()
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()
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.

remove()
show()

trace_message(node, description='node')

class SCons.Script.Main.CountStats
do_append(label)
do_nothing(*args, **kw)
do_print()
enable(outfp)

class SCons.Script.Main.FakeOptionParser
Bases: object
A do-nothing option parser, used for the initial OptionsParser variable.
During normal SCons operation, the OptionsParser is created right away by the main() function. Certain tests 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.
SCons API Documentation

```python
class FakeOptionValues
    Bases: object
    add_local_option (*args, **kw)
values = <SCons.Script.Main.FakeOptionParser.FakeOptionValues object>
SCons.Script.Main.GetBuildFailures ()
SCons.Script.Main.GetOption (name)
class SCons.Script.Main.MemStats
    do_append (label)
    do_nothing (*args, **kw)
    do_print ()
    enable (outfp)
SCons.Script.Main.PrintHelp (file=None)
SCons.Script.Main.Progress (*args, **kw)
class SCons.Script.Main.Progressor (obj, interval=1, file=None, overwrite=False)
    Bases: object
    count = 0
    erase_previous ()
    prev = ''
    replace_string (node)
    spinner (node)
    string (node)
    target_string = '$TARGET'
    write (s)
class SCons.Script.Main.QuestionTask (tm, targets, top, node)
    Bases: SCons.Taskmaster.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 ()
    display (message)
        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 ()
        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)
        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 ()
        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
```

84
“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()
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()
Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s callback methods.

fail_continue()
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()
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()
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()
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()
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()
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='node')

exception SCons.Script.Main.SConsPrintHelpException
Bases: Exception

args

with_traceback()

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

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

class SCons.Script.Main.Stats

Bases: object

do_nothing (*args, **kw)

enable (outfp)

class SCons.Script.Main.TreePrinter (derived=False, prune=False, status=False, sLineDraw=False)

Bases: object

display (t)

get_all_children (node)

get_derived_children (node)

SCons.Script.Main.ValidateOptions (throw_exception=False) → None

Validate options passed to SCons on the command line.

If you call this after you set all your command line options with AddOption(), it will verify that all command line options are valid. So 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 (bool) – (Optional) Should this function raise an error if there's an invalid option on the command line, or issue a message and exit with error status.

Raises:

   SConsBadOptionError – If throw_exception is True and there are invalid options on command line.

New in version 4.5.0.

SCons.Script.Main._SConstruct_exists (dirname='', repositories=[], filelist=None)

This function checks that an SConstruct file exists in a directory. If so, it returns the path of the file. By default, it checks the current directory.

SCons.Script.Main._build_targets (fs, options, targets, target_top)

SCons.Script.Main._create_path (plist)

SCons.Script.Main._exec_main (parser, values)

SCons.Script.Main._load_all_site_scons_dirs (topdir, verbose=False)

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 ()

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)

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)

Handle syntax errors. Print out a message and show where the error occurred.

SCons.Script.Main._scons_user_error (e)

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)

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)

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 API Documentation

SCons.Script.Main.main ()
SCons.Script.Main.path_string (label, module)
SCons.Script.Main.python_version_deprecated (version=sys.version_info(major=3, minor=10, micro=10, releaselevel='final', serial=0))
SCons.Script.Main.python_version_string ()
SCons.Script.Main.python_version_unsupported (version=sys.version_info(major=3, minor=10, micro=10, releaselevel='final', serial=0))
SCons.Script.Main.revert_io ()
SCons.Script.Main.test_load_all_site_scons_dirs (d)
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:

```
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: optparse.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

**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: optparse.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.

derdent ()
expand_default (option)
format_description (description)
format_epilog (epilog)
format_heading (heading)
   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.
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 metavarars 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)
Formats the usage message.

indent ()

set_long_opt_delimiter (delim)

set_short_opt_delimiter (delim)

store_option_strings (parser)

class SCons.Script.SConsOptions.SConsOption (*opts, **attrs)
Bases: optparse.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_kwargs', '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 Option._check_callback>, <function SConsOption._check_nargsOptional>]
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>}
SCons API Documentation

check_value (opt, value)
convert_value (opt, value)
get_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: optparse.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: optparse.OptionParser
  _add_help_option ()
  _add_version_option ()
  _check_conflict (option)
  _create_option_list ()
  _create_option_mappings ()
  _get_all_options ()
  _get_args ()
  _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.

```python
_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.

```python
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.

derror(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_option_help(formatter=None)
get_default_values()
get_description()
get_option(optarg)
get_option_group(optarg)
get_prog_name()
get_usage()
get_version()
has_option(optarg)
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_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 ()

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: optparse.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 are must be explicitly added to the “settable” list in this class, and optionally validated and coerced in the set_option() method.

__getattr__(attr)

Fetched 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.

event_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=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: SCons.Environment.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)

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)

Append values to construction variables in an Environment.

The variable is created if it is not already present.

AppendENVPath (name, newpath, envname='ENV', sep=':', deleteExisting=False)

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=False, **kw``)

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``)

**Clean** (``targets, files``)

**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``)

**Depends** (``target, dependency``)

Explicitly specify that ‘target’ depends 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='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``)

Exit abnormally if the Python version is not late enough.

**static** **EnsureSConsVersion** (``major, minor, revision=0``)

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=0``)

**Export** (``*vars, **kw``)

**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=')') → list
Return a list of all source files.

Flatten (sequence)
GetBuildPath (files)
static GetLaunchDir ()
GetOption (name)
Glob (pattern, ondisk=True, source=False, strings=False, exclude=None)
Help (text, append=False)
Ignore (target, dependency)
Ignore a dependency.
Import (*vars)
Literal (string)
Local (*targets)
MergeFlags (args, unique=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().

**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=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=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)
Prepend (**kw)
  Prepend values to construction variables in an Environment.
  The variable is created if it is not already present.
PrependENVPath (name, newpath, envname='ENV', sep=':', delete_existing=True)
  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=False, **kw)
  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)
PyPackageDir (modulename)
RemoveMethod (function)
  Removes the specified function’s MethodWrapper from the added_methods list, so we don’t re-bind it when making a clone.
Replace (**kw)
  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)
Requires (target, prerequisite)
  Specify that ‘prerequisite’ must be built before ‘target’, (but ‘target’ does not actually depend on ‘prerequisite’ and need not be rebuilt if it changes).
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)
SCons API Documentation

Scanner (*args, **kw)
SetDefault (**kw)
SetOption (name, value)
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) → SCons.Tool.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=1)

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)
_changed_content (dependency, target, prev_ni, repo_node=None)
_changed_source (dependency, target, prev_ni, repo_node=None)
_changed_timestamp_match (dependency, target, prev_ni, repo_node=None)
_changed_timestamp_newer (dependency, target, prev_ni, repo_node=None)
_changed_timestamp_then_content (dependency, target, prev_ni, repo_node=None)
_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 ()
Initial the dispatch tables for special handling of special construction variables.

_update (other)
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)
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)
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
def get(key, default=None):
    Emulates the `get()` method of dictionaries.

def get_CacheDir():
    Fetch the builder with the specified name from the environment.

get_factory(factory, default='File')
    Return a factory function for creating Nodes for this construction environment.

def get_scanner(skey):
    Find the appropriate scanner given a key (usually a file suffix).

def get_src_sig_type():

def get_tgt_sig_type():

def gvars():
    Emulates the `items()` method of dictionaries.

def keys():
    Emulates the `keys()` method of dictionaries.

def lvars():

def scanner_map_delete(kw=None):
    Delete the cached scanner map (if we need to).

def setdefault(key, default=None):
    Emulates the `setdefault()` method of dictionaries.

def subst(string, raw=0, target=None, source=None, conv=None, executor=None, overrides=False):
    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=None, raw=0, target=None, source=None)

def subst_list(string, raw=0, target=None, source=None, conv=None, executor=None, overrides=False):
    Calls through to `SCons.Subst.scons_subst_list()`. See the documentation for that function.

def subst_path(path, target=None, source=None)
    Substitute a path list, turning EntryProxies into Nodes and leaving Nodes (and other objects) as-is.

subst_target_source(string, raw=0, target=None, source=None, conv=None, executor=None, overrides=False):
    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.

def values():
    Emulates the `values()` method of dictionaries.

exception SCons.Script.SConscript.SConscriptReturn
    Bases: Exception

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

SCons.Script.SConscript.SConscript_exception(file=<_io.TextIOWrapper name='<stderr>' mode='w', encoding='utf-8'>)
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, must_exist=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 value. On first warning, print a deprecation message.

Parameters:
- f (str) – path of missing configuration file
- must_exist (bool) – if true, fail. If false, but not None, allow the file to be missing. The default is None, which means issue the warning. The default is deprecated.

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=False)
class SCons.Script.TargetList (initlist=None)

Bases: collections.UserList

_add_Default (list)
_clear ()
_do_nothing (*args, **kw)
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 API Documentation

SCons.Script._Add_Targets(tlist)
SCons.Script._Get_Default_Targets(d, fs)
SCons.Script._Set_Default_Targets(env, tlist)
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_sconsscript_error(flag=1)

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 ()
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.
  _reset_sig_handler ()
    Restore the signal handlers to their previous state (before the call to _setup_sig_handler()).
  _setup_sig_handler ()
    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>>)
    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 State** *(value)*
   Bases: enum.Enum
   An enumeration.
   COMPLETED = 3
   READY = 0
   SEARCHING = 1
   STALLED = 2
**class Worker** *(owner)*
   Bases: threading.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 ()
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.

_adjust_stack_size ()
_restore_stack_size (prev_size)
_setup_logging ()
_start_workers ()
_work ()
start ()
trace_message (message)

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 ()
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)
pull (task)
Put task into request queue.

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

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.

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.

defDaemon(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: SCons.Taskmaster.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()
    display(message)
    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()
    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)
    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()
    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 ()
    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 ()
    Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s callback methods.

fail_continue ()
    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 ()
    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 ()
    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 ()
    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 ()
    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 ()
    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='node')

class SCons.Taskmaster.OutOfDateTask (tm, targets, top, node)
    Bases: SCons.Taskmaster.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()
display(message)
  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()
  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)
  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().
exected()
  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()
  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()
  Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s callback methods.
fail_continue()
  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()
  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()
**make_ready_all()**
Marks all targets in a task ready for execution.
This is the default behavior for building only what's necessary.

**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()**
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='node')**

**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.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()**

**display(message)**
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()**
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)**
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 ()
   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 ()
   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 ()
   Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s
   callback methods.
fail_continue ()
   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 ()
   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().
fail ()
   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 ()
   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 ()
   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.

```python
trace_message (node, description='node')
```

```python
class SCons.Taskmaster.Taskmaster (targets=[], tasker=None, order=None, trace=None)
```

**Bases:** object

The Taskmaster for walking the dependency DAG.

```python
__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.

```python
__validate_pending_children ()
```

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) --> ...    
|                     ^                                       |
|                   +---------------------+                  |
|                                         /             
|                                      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)
    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()
    Stops the current build completely.

SCons.Taskmaster.dump_stats()
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)

class SCons.Tool.Tool(name, toolpath=None, **kwargs)
    Bases: object
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)
Searches the list of associated Tool modules for one that exists, and applies that to the construction environment.

**remove_methods** (env)
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=’$_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=’$_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=False) → Optional[std]
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.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.Variables.EnumVariable.EnumVariable (key, help, default, allowed_values, map={}, ignorecase=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.ListVariable (key, help, default, names, map={}) → Tuple[str, str, str, None, Callable]
SCons API Documentation

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, 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) ->
SCons.Variables.ListVariable._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

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.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
SCons API Documentation

- **default** - default value for this option
- **validator** - [optional] validator for option value. Predefined are:
  - **PathAccept** - accepts any path setting; no validation
  - **PathIsDir** - path must be an existing directory
  - **PathIsDirCreate** - path must be a dir; will create
  - **PathIsFile** - 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=True)
Bases: object
Hold 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 (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.
- **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:

```
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 help text for a variable.

**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.

**do_add** (key, help='', default=None, validator=None, converter=None, **kwargs) → None

**aliasfmt** = '{default}
**fmt** = '{default}
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)
   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: object

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.

batch_key (env, target, source)
genstring (target, source, env)
get_contents (target, source, env)
get_targets (env, executor=None)

Returns the type of targets ($TARGETS, $CHANGED_TARGETS) used by this action.

get_varlist (target, source, env, executor=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)
strfunction (target, source, env)
SCons API Documentation

```
subst (s, target, source, env)
subst_args (target, source, env)
subst_kw (target, source, env)
```

```python
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: SCons.Action._ActionAction
Class for command-execution actions.
```

```
_get_implicit_deps_heavyweight (target, source, env, executor, 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)
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=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)
get_contents (target, source, env)
get_implicit_deps (target, source, env, executor=None)
Return the implicit dependencies of this action’s command line.
get_presig (target, source, env, executor=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)
Returns the type of targets ($TARGETS, $CHANGED_TARGETS) used by this action.
get_varlist (target, source, env, executor=None)
no_batch_key (env, target, source)
presub_lines (env)
print_cmd_line (s, target, source, env)
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=False)
strfunction (target, source, env, executor=None, overrides=False)
```

```python
class SCons.Action.CommandGeneratorAction (generator, kw)
Bases: SCons.Action._ActionBase
Class for command-generator actions.
```

```
_generate (target, source, env, for_signature, executor=None)
batch_key (env, target, source)
genstring (target, source, env, executor=None)
get_contents (target, source, env)
get_implicit_deps (target, source, env, executor=None)
get_presig (target, source, env, executor=None)
Return the signature contents of this action’s command line.
```

118
get_targets (env, executor)

Returns the type of targets ($TARGETS, $CHANGED_TARGETS) used by this action.

get_varlist (target, source, env, executor=None)

no_batch_key (env, target, source)

presub_lines (env)

class SCons.Action.FunctionAction (execfunction, kw)
Bases: SCons.Action._ActionAction
Class for Python function actions.

batch_key (env, target, source)

execute (target, source, env, executor=None)

function_name ()

genstring (target, source, env)

get_implicit_deps (target, source, env)

get_presig (target, source, env)

get_contents (target, source, env)

get_implicit_deps (target, source, env)

get_targets (env, executor)

Returns the type of targets ($TARGETS, $CHANGED_TARGETS) used by this action.

get_varlist (target, source, env, executor=None)

no_batch_key (env, target, source)

presub_lines (env)

class SCons.Action.LazyAction (var, kw)

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.

_generate (target, source, env, for_signature, executor=None)

_generate_cache (env)

_get_implicit_deps_heavyweight (target, source, env, executor, 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)

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=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=None)

get_contents (target, source, env)
get_implicit_deps (target, source, env, executor=None)
    Return the implicit dependencies of this action's command line.
get_parent_class (env)
get_presig (target, source, env)
    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)
    Returns the type of targets ($TARGETS, $CHANGED_TARGETS) used by this action.
get_varlist (target, source, env, executor=None)
no_batch_key (env, target, source)
presub_lines (env)
print_cmd_line (s, target, source, env)
    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=False)
strfunction (target, source, env, executor=None, overrides=False)

class SCons.Action.ListAction (actionlist)
    Bases: SCons.Action.ActionBase
    Class for lists of other actions.
batch_key (env, target, source)
genstring (target, source, env)
gevent_contents (target, source, env)
gevent_implicit_deps (target, source, env)
gevent_presig (target, source, env)
    Return the signature contents of this action list.
    Simple concatenation of the signatures of the elements.
get_targets (env, executor)
    Returns the type of targets ($TARGETS, $CHANGED_TARGETS) used by this action.
get_varlist (target, source, env, executor=None)
no_batch_key (env, target, source)
presub_lines (env)
print_cmd_line (s, target, source, env)

class SCons.Action._ActionAction (cmdstr=<class 'SCons.Action._null'>, strfunction=<class 'SCons.Action._null'>, varlist=(), presub=<class 'SCons.Action._null'>, chdir=None, exitstatfunc=None, batch_key=None, targets='$TARGETS', **kw)
    Bases: SCons.Action.ActionBase
    Base class for actions that create output objects.
batch_key (env, target, source)
genstring (target, source, env)
gevent_contents (target, source, env)
gevent_targets (env, executor)
    Returns the type of targets ($TARGETS, $CHANGED_TARGETS) used by this action.
gevent_varlist (target, source, env, executor=None)
no_batch_key (env, target, source)
presub_lines (env)
print_cmd_line (s, target, source, env)
    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)
SCons.Action._callable_contents (obj)
    Return the signature contents of a callable Python object.
SCons.Action._code_contents (code, docstring=None)
    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:
For info on what each co_variable provides

The signature is as follows (should be byte/chars): co_argcount, len(co_varnames), len(co_cellvars), len(co_freevars), (comma separated signature for each object in co_consts), (comma separated signature for each object in co_names), (The bytecode with line number bytecodes removed from co_code)

- 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)
This is the actual “implementation” for the Action factory method, below. 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)
A factory for list actions. Convert the input list into Actions and then wrap them in a ListAction.

SCons.Action._function_contents (func)
The signature is as follows (should be byte/chars): <_code_contents (see above) from func.__code__, (comma separated _object_contents for function argument defaults), (comma separated _object_contents for any closure contents)

See also: https://docs.python.org/3/reference/datamodel.html

- 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.

Returns: Signature contents of a function. (in bytes)

class SCons.Action._null
Bases: object

SCons.Action._object_contents (obj)
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 consistant content for an action class or an instance thereof

Parameters:
- obj Should be either and 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 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.

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, instanciated 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.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='', suffix='', src_suffix='', target_factory=None, source_factory=None, target_scanner=None, source_scanner=None, emitter=None, multi=0, env=None, single_source=0, name=None, chdir=<class 'SCons.Builder._Null'>, is_explicit=1, src_builder=None, ensure_suffix=False, **overrides)

Bases: object
Base class for Builders, objects that create output nodes (files) from input nodes (files).

  _adjustixes (files, pre, suf, ensure_suffix=False)
  _create_nodes (env, target=None, source=None)
    Create and return lists of target and source nodes.
  _execute (env, target, source, overwarn={}, executor_kw={})
  _get_sdict (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)
    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)
    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 (suff)
  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)
set_suffix (suffix)
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:** `SCons.Util.Selector`

A callable dictionary that will, in turn, call the value it finds if it can.

- clear () → None. Remove all items from od.
- copy () → a shallow copy of od
- fromkeys (value=None)
  - Create a new ordered 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
- move_to_end (key, last=True)
  - Move an existing element to the end (or beginning if last is false).
  - Raise KeyError if the element does not exist.
- pop (key[, default]) → 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 (last=True)
  - Remove and return a (key, value) pair from the dictionary.
  - Pairs are returned in LIFO order if last is true or FIFO order if false.
- 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

### Class `SCons.Builder.CompositeBuilder`

**Bases:** `SCons.Util.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)
```

```
get ()
```

Retrieve the entire wrapped object

### Class `SCons.Builder.DictCmdGenerator`

**Bases:** `SCons.Util.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.

```
add_action (suffix, action)
```

Add a suffix-action pair to the mapping.
clear () → None. Remove all items from od.
copy () → a shallow copy of od
fromkeys (value=None)
    Create a new ordered 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
move_to_end (key, last=True)
    Move an existing element to the end (or beginning if last is false).
    Raise KeyError if the element does not exist.
pop (key[, default]) → 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 (last=True)
    Remove and return a (key, value) pair from the dictionary.
    Pairs are returned in LIFO order if last is true or FIFO order if false.
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.
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]
values () → an object providing a view on D's values

class SCons.Builder.DictEmitter
    Bases: SCons.Util.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 od.
copy () → a shallow copy of od
fromkeys (value=None)
    Create a new ordered 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
move_to_end (key, last=True)
    Move an existing element to the end (or beginning if last is false).
    Raise KeyError if the element does not exist.
pop (key[, default]) → 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 (last=True)
    Remove and return a (key, value) pair from the dictionary.
    Pairs are returned in LIFO order if last is true or FIFO order if false.
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.EmitterProxy (var)
    Bases: object
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.

class SCons.Builder.ListEmitter (initlist=None)
    Bases: collections.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 ()
        count (value) → integer — remove all items from S
    copy ()
    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.Builder.Override Warner (mapping)
    Bases: collections.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 () → None. Remove all items from D.
    copy ()

    superclassmethod 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
    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[k]=d if k not in D
    update ([, E], **F) → None. Update D from mapping/iterable E and F.
        If E present and has .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
    warn ()

class SCons.Builder._Null
    Bases: object

    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.
    SCons.Builder._null
        alias of SCons.Builder._Null
    SCons.Builder.is_a_Builder (obj)
“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

class SCons.CacheDir.CacheDir (path)
    Bases: object
    CacheDebug (fmt, target, cachefile)
    _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.

classmethod get_cachedir_csig (node)

    property hit_ratio: float
    is_enabled () → bool
    is_readonly () → bool

    property misses: int

    push (node)
    push_if_forced (node)
    retrieve (node)
        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.

SCons.CacheDir.CachePushFunc (target, source, env)
SCons.CacheDir.CacheRetrieveFunc (target, source, env)
SCons.CacheDir.CacheRetrieveString (target, source, env)

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:

**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!

**context.Log(msg)**

Function called to write to a log file.

**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).

**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).

**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.)

**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.)

**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.).

**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!).

**context.config_h (may be missing).**

If present, must be a string, which will be filled with the contents of a config_h file.

**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.

**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.

**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.

**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.

**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)
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(). 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.

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=1, append=True, unique=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 library 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, $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.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 the type to check
- includes (-) – sequence list of headers to include in the test code before testing the type
- language (-) – str ‘C’ or ‘C++’
- expect (-) – int if given, will test whether 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 string “#define key have”.

SCons.Conftest._LogFailed (context, text, msg)
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)
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=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
SCons.API Documentation

- 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:

from SCons.Debug import caller_trace, caller_trace()
SCons.Debug.Trace(msg, tracefile=None, mode='w', tstamp=False)
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)

SCons.Debug._dump_one_caller(key, file, level=0)
return caller's stack
SCons.Debug.caller_stack()  
return caller's stack
SCons(Debug.caller_trace(back=0)
Trace caller stack and save info into global dicts, which are printed automatically at the end of SCons execution.
SCons.Debug.countLoggedInstances(classes, file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>)
SCons.Debug.dumpLoggedInstances(classes, file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>)
SCons.Debug.dump_caller_counts(file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>)
SCons.Debug.fetchLoggedInstances(classes='*')
SCons.Debug.func_shorten(func_tuple)
SCons.Debug.listLoggedInstances(classes, file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>)
SCons.Debug.logInstanceCreation(instance, name=None)
SCons.Debug.memory()  
SCons.Debug.string_to_classes(s)

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.
SCons.Defaults.DefaultEnvironment(*args, **kw)
Initial public entry point for creating the default construction Environment.
After creating the environment, we overwrite our name (DefaultEnvironment) with the _fetch_DefaultEnvironment() function, which more efficiently returns the initialized default construction environment without checking for its existence.
(This function still exists with its _default_check because someone else (cough Script/__init__.py cough) may keep a reference to this function. So we can't use the fully functional idiom of having the name originally be a something that only creates the construction environment and then overwrites the name.)
class SCons.Defaults.NullCmdGenerator(cmd)
This is a callable class that can be used in place of other command generators if you don’t want them to do anything.

Example usage:
```python
env["DO NOTHING"] = NullCmdGenerator
env["LINKCOM"] = "$(DO NOTHING('$(LINK $SOURCES $TARGET)')"
```

---

**Bases:** object

A class for finding a construction variable on the stack and calling one of its methods.

We use this to support “construction variables” in our string eval()s that actually stand in for methods—specifically, use of “RDirs” in call to `_concat` that should actually execute the “TARGET.RDirs” method. (We used to support this 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**

```python
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.__libversionflag**

```python
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**

```python
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**

```python
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**

```python
A wrapper around _concat_ixes() that turns a list or string into a list of C preprocessor command-line definitions.
```

---

**SCons.Defaults._fetch_DefaultEnvironment**

```python
Returns the already-created default construction environment.
```

---

**SCons.Defaults._stripixes**

```python
This is a wrapper around _concat()/_concat_ixes() that checks for the existence of prefixes or suffixes on list items and strips them where it finds them. This is used by tools (like the GNU linker) that need to turn something like ‘libfoo.a’ into ‘-lfoo’.
```

---

**SCons.Defaults.chmod_func**

```python
Implementation of the Chmod action function.
```

---

**SCons.Defaults.chmod_strfunc**

```python
strfunction for the Chmod action function.
```

---

**SCons.Defaults.copy_func**

```python
Implementation of the Copy action function.
```

---

**SCons.Defaults.copy_strfunc**

```python
strfunction for the Copy action function.
```

---

**SCons.Defaults.delete_func**

```python
Implementation of the Delete action function.
```
Let the Python os.unlink() raise an error if \texttt{dest} does not exist, unless \texttt{must_exist} evaluates false (the default).

\texttt{SCons.Defaults.delete_strfunc (dest, must_exist=False) \rightarrow str}

\texttt{strfunction for the Delete action function.}

\texttt{SCons.Defaults.get_paths_str (dest) \rightarrow str}

Generates a string from \texttt{dest} for use in a strfunction.

If \texttt{dest} is a list, manually converts each elem to a string.

\texttt{SCons.Defaults.mkdir_func (dest) \rightarrow None}

Implementation of the Mkdir action function.

\texttt{SCons.Defaults.move_func (dest, src) \rightarrow None}

Implementation of the Move action function.

\texttt{SCons.Defaults.processDefines (defs) \rightarrow List[str]}

Return list of strings for preprocessor defines from \texttt{defs}.

Resolves the different forms \texttt{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.

\texttt{SCons.Defaults.touch_func (dest) \rightarrow None}

Implementation of the Touch action function.

\texttt{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.

\texttt{class SCons.Environment.Base (platform=None, tools=None, toolpath=None, variables=None, parse_flags=None, **kw)}

\texttt{Bases: SCons.Environment.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.

\texttt{Action (**args, **kw)}

\texttt{AddMethod (function, name=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.

\texttt{AddPostAction (files, action)}

\texttt{AddPreAction (files, action)}

\texttt{Alias (target, source=[], action=None, **kw)}

\texttt{AlwaysBuild (*targets)}

\texttt{Append (**kw)}

Append values to construction variables in an Environment.

The variable is created if it is not already present.

\texttt{AppendENVPath (name, newpath, envname='ENV', sep=':', delete-existing=False)}

Append path elements to the path \texttt{name} in the \texttt{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 \texttt{delete-existing} is False, a \texttt{newpath} element already in the path will not be moved to the end (it will be left where it is).

\texttt{AppendUnique (delete-existing=False, **kw)}

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.

\texttt{Builder (**kw)}
CacheDir (path, custom_class=None)
Clean (targets, files)
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)
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='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='.') → list
   Return a list of all source files.
Flatten (sequence)
GetBuildPath (files)
Glob (pattern, ondisk=True, source=False, strings=False, exclude=None)
Ignore (target, dependency)
Ignore a dependency.
Literal (string)
Local (*targets)
MergeFlags (args, unique=True) → None
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().

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=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=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)
Prepend (**kw)
Prepend values to construction variables in an Environment.
The variable is created if it is not already present.
PrependENVPath (name, newpath, envname="ENV", sep=':', delete_existing=True)
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=False, **kw)

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)

PyPackageDir (modulename)

RemoveMethod (function)

Removes the specified function’s MethodWrapper from the added_methods list, so we don’t re-bind it when making a clone.

Replace (**kw)

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)

Requires (target, prerequisite)

Specify that ‘prerequisite’ must be built before ‘target’, (but ‘target’ does not actually depend on ‘prerequisite’ and need not be rebuilt if it changes).

SConsignFile (name=’.sconsign’, dbm_module=None)

Scanner (*args, **kw)

SetDefault (**kw)

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) → SCons.Tool.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=1)

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

Find prog in the path.

_canonitizelize (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)

_changed_content (dependency, target, prev_ni, repo_node=None)

_changed_source (dependency, target, prev_ni, repo_node=None)

_changed_timestamp_match (dependency, target, prev_ni, repo_node=None)

_changed_timestamp_newer (dependency, target, prev_ni, repo_node=None)

_changed_timestamp_then_content (dependency, target, prev_ni, repo_node=None)

_find_toolpath_dir (tp)

_gsm ()

_init_special ()

Initial the dispatch tables for special handling of special construction variables.
update (other)

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)

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)

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='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).

get_src_sig_type ()

get_tgt_sig_type ()

gvars ()

Emulates the items() method of dictionaries.

keys ()

Emulates the keys() method of dictionaries.

lvars ()

scanner_map_delete (kw=None)

Delete the cached scanner map (if we need to).

setdefault (key, default=None)

Emulates the setdefault() method of dictionaries.

subst (string, raw=0, target=None, source=None, conv=None, executor=None, overrides=False)

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=0, target=None, source=None)

Calls through to SCons.Subst.scons_subst(). See the documentation for that function.

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

Substitute a path list, turning EntryProxies into Nodes and leaving Nodes (and other objects) as-is.

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.BuilderDict (mapping, env)
Bases: collections.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.

_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
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

class SCons.Environment.BuilderWrapper (obj, method, name=None)
Bases: SCons.Util.envs.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.

clone (new_object)
Returns an object that re-binds the underlying “method” to the specified new object.

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.

class SCons.Environment.OverrideEnvironment (subject, overrides=None)
Bases: SCons.Environment.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)
   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)
   Append values to construction variables in an Environment.
   The variable is created if it is not already present.
AppendENVPath (name, newpath, envname='ENV', sep=':', delete_existing=False)
   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=False, **kw)
   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)
Clean (targets, files)
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)
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='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='.') → list
    Return a list of all source files.

Flatten (sequence)
GetBuildPath (files)
Glob (pattern, ondisk=True, source=False, strings=False, exclude=None)

Ignore (target, dependency)
    Ignore a dependency.

Literal (string)
Local (*targets)

MergeFlags (args, unique=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().

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=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=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)

Prepend (**kw)
Prepend values to construction variables in an Environment.
The variable is created if it is not already present.

PrependENVPath (name, newpath, envname='ENV', sep=':', delete_existing=True)
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=False, **kw)
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)

PyPackageDir (modulename)

RemoveMethod (function)
Removes the specified function’s MethodWrapper from the added_methods list, so we don’t re-bind it when making a clone.

Replace (**kw)
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)
Requires (target, prerequisite)
Specify that ‘prerequisite’ must be built before ‘target’, (but ‘target’ does not actually depend on ‘prerequisite’ and need not be rebuilt if it changes).

SConsignFile (name='.sconsign', dbm_module=None)
Scanner (**args, **kw)
SetDefault (**kw)
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**

```python
Tool(tool, toolpath=None, **kwargs) -> SCons.Tool.Tool
```

Find and run tool module `tool`.

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

**Value**

```python
Value(value, built_value=None, name=None)
```

Return a Value (Python expression) node.

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

**VariantDir**

```python
VariantDir(variant_dir, src_dir, duplicate=1)
```

**Whereis**

```python
Whereis(prog, path=None, pathext=None, reject=None)
```

Find `prog` in the path.

```python
_canonicalize(path)
```

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

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

```python
_changed_build(dependency, target, prev_ni=None, repo_node=None)
_changed_content(dependency, target, prev_ni, repo_node=None)
_changed_source(dependency, target, prev_ni, repo_node=None)
_changed_timestamp_match(dependency, target, prev_ni, repo_node=None)
_changed_timestamp_newer(dependency, target, prev_ni, repo_node=None)
_changed_timestamp_then_content(dependency, target, prev_ni, repo_node=None)
_find_toolpath_dir(tp)
_gsm()
_init_special()
_update(other)
_update_onlynew(other)
```

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

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

Update a dict with new keys.

Unlike the `.update` method, if the key is already present, it is not replaced.

**arg2nodes**

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

**backtick**

```python
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**

```python
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=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 API Documentation

get_src_sig_type ()
get_tgt_sig_type ()
gvars ()
items ()

   Emulates the items() method of dictionaries.
keys ()

   Emulates the keys() method of dictionaries.
lvars ()
scanner_map_delete (kw=None)

   Delete the cached scanner map (if we need to).
setdefault (key, default=None)

   Emulates the setdefault() method of dictionaries.
subst (string, raw=0, target=None, source=None, conv=None, executor=None, overrides=False)

   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 alphanumeric 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=0, target=None, source=None)
subst_list (string, raw=0, target=None, source=None, conv=None, executor=None, overrides=False)

   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, turning EntryProxies into Nodes and leaving Nodes (and other objects) as-is.
subst_target_source (string, raw=0, target=None, source=None, conv=None, executor=None, overrides=False)

   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 alphanumeric 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)

   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=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().
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)

Removes the specified function’s MethodWrapper from the added_methods list, so we don’t re-bind it when making a clone.

**_init_special** ()

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)

**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.

**gvars ()**

Emulates the items() method of dictionaries.

**keys ()**

Emulates the keys() method of dictionaries.

**lvars ()**

**setdefault** (key, default=None)

Emulates the setdefault() method of dictionaries.

**subst** (string, raw=0, target=None, source=None, conv=None, executor=None, overrides=False)

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=0, target=None, source=None)

Calls through to SCons.Subst.scons_subst_list(). See the documentation for that function.

**subst_list** (string, raw=0, target=None, source=None, conv=None, executor=None, overrides=False)

Substitute a path list, turning EntryProxies into Nodes and leaving Nodes (and other objects) as-is.

**subst_target_source** (string, raw=0, target=None, source=None, conv=None, executor=None, overrides=False)
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.

Values ()
Emulates the values() method of dictionaries.

Class SCons.Environment._Null
Bases: object

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)
SCons.Environment._delete_duplicates (l, keep_last)
Delete duplicates from a sequence, keeping the first or last.
SCons.Environment._null alias of SCons.Environment._Null
SCons.Environment._set_BUILDERS (env, key, value)
SCons.Environment._set_SCANNERS (env, key, value)
SCons.Environment._set_future_reserved (env, key, value)
SCons.Environment._set_reserved (env, key, value)
SCons.Environment.alias_builder (env, target, source)
SCons.Environment.apply_tools (env, tools, toolpath)
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)
Return if the specified string is a legitimate construction variable.

SCons.Errors module
SCons exception classes.

Used to handle internal and user errors in SCons.

Exception SCons.Errors.BuildError (node=None, errstr='Unknown error', status=2, exitstatus=2, filename=None, executor=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.

```python
exception SCons.Errors.ExplicitExit (node=None, status=None, *args)
    Bases: Exception
exception SCons.Errors.InternalError
    Bases: Exception
exception SCons.Errors.MSVCError
    Bases: OSError
exception SCons.Errors.SConsEnvironmentError
    Bases: Exception
exception SCons.Errors.StopError
    Bases: Exception
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.

```python
SCons.Executor.AddBatchExecutor (key, executor)
class SCons.Executor.Batch (targets=[], sources=[])  
    Bases: object
    Remembers exact association between targets and sources of executor.
    sources
    targets
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 ()
_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)
  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)
add_pre_action (action)
add_sources (sources)
  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 ()
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

get_implicit_deps ()

Return the executor’s implicit dependencies, i.e. the nodes of the commands to be executed.

get_kw (kw={})

get_lvars ()

get_sources ()

get_timestamp ()

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.

get_unignored_sources (node, ignore=())

lvars
nullify ()
overridelist
post_actions
pre_actions
prepare ()

Preparatory checks for whether this Executor can go ahead and (try to) build its targets.

scan (scanner, node_list)

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)

scan_targets (scanner)

set_action_list (action)

SCons.Executor.GetBatchExecutor (key)

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 ()

Morph this Null executor to a real Executor object.

_unchanged_sources_list
_unchanged_targets_list
action_list
add_post_action (action)
add_pre_action (action)
batches
builder_kw
cleanup ()
env
get_action_list ()
get_action_side_effects ()
get_action_targets ()
get_all_children ()
get_all_prerequisites ()
get_all_sources ()
get_all_targets ()
get_build_env ()
SCons API Documentation

get_build_scanner_path ()
get_contents ()
get_unignored_sources (*args, **kw)
lvars
overridelist
post_actions
pre_actions
prepare ()
set_action_list (action)

class SCons.Executor_NullEnvironment (*args, **kwargs)
Bases: SCons.Util.types.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_TLSList (func)
Bases: collections.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>
apend (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.Executor_TLSObject (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)
SCons.Executor.execute_null_str (obj)
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.).

149
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
@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.

```python
class SCons.Memoize.CountDict(cls_name, method_name, keymaker):
    Bases: SCons.Memoize.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)
        Counts whether the computed key value is already present in the memoization dictionary (a hit) or not (a miss).
    display()
    key()
```

```python
class 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().
```

```python
class SCons.Memoize.CountValue(cls_name, method_name):
    Bases: SCons.Memoize.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)
        Counts whether the memoized value has already been set (a hit) or not (a miss).
    display()
    key()
```

```python
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()
    key()
```

```python
SCons.Memoize.Dump(title=None)
    Dump the hit/miss count for all the counters collected so far.
SCons.Memoize.EnableMemoization()
```

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.

```python
SCons.PathList.PathList(pathlist)
    Returns the cached _PathList object for the specified pathlist, creating and caching a new object as necessary.
```
SCons API Documentation

**Bases:** object

An actual PathList object.

```python
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.

```python
SCons.SConf.CheckCC (context)
SCons.SConf.CheckCHeader (context, header, include_quotes='''''')
A test for a C header file.
SCons.SConf.CheckCXX (context)
SCons.SConf.CheckCXXHeader (context, header, include_quotes='''''')
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:

```python
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.

```python
AppendLIBS (lib_name_list, unique=False)
BuildProg (text, ext)
CompileProg (text, ext)
CompileSharedObject (text, ext)
Display (msg)
Log (msg)
Message (text)
        Inform about what we are doing right now, e.g. 'Checking for SOMETHING ... ' 
PrependLIBS (lib_name_list, unique=False)
Result (res)
        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='', language=None)
SCons.SConf.CheckFunc (context, function_name, header=None, language=None)
SCons.SConf.CheckHeader (context, header, include_quotes='<>', language=None)
        A test for a C or C++ header file.
        A test for a C++ header file.
SCons.SConf.CheckLib (context, library=None, symbol='main', header=None, language=None, autoadd=True, append=True, unique=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.

SCons.SConf.CheckLibWithHeader (context, libs, header, language, call=None, autoadd=True, append=True, unique=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.

SCons.SConf.CheckMember (context, aggregate_member, header=None, language=None)
Returns the status (False : failed, True : ok).

SCons.SConf.CheckProg (context, prog_name)
Simple check if a program exists in the path. Returns the path for the application, or None if not found.

SCons.SConf.CheckSHCC (context)
SCons.SConf.CheckSHCXX (context)

SCons.SConf.CheckType (context, type_name, includes='', language=None)
SCons.SConf.CheckTypeSize (context, type_name, includes='', language=None, expect=None)

exception SCons.SConf.ConfigureCacheError (target)
Bases: SCons.SConf.SConfError
Raised when a use explicitly requested the cache feature, but the test is run the first time.

args
with_traceback ()

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

exception SCons.SConf.ConfigureDryRunError (target)
Bases: SCons.SConf.SConfError
Raised when a file or directory needs to be updated during a Configure process, but the user requested a dry-run

args
with_traceback ()

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

SCons.SConf.CreateConfigHBuilder (env)
Called if necessary just before the building targets phase begins.

SCons.SConf.NeedConfigHBuilder ()

SCons.SConf.SConf (**kw)

class SCons.SConf.SConfBase (env, custom_tests={}, conf_dir='$CONFIGUREDIR', log_file='$CONFIGURELOG', config_h=None, _depth=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.

AddTest (test_name, test_instance)
Adds test_class to this SConf instance. It can be called with self.test_name(...)

AddTests (tests)
Adds all the tests given in the tests dictionary to this SConf instance

BuildNodes (nodes)
Tries to build the given nodes immediately. Returns 1 on success, 0 on error.

Define (name, value=None, comment=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="")
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="")
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).

_class_ SCons.SConf.SConfBuildInfo
Bases: SCons.Node.FS.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.

createDir (node)
Private method. Reset to non-piped spawn

_shutdown ()
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.

_format_ (names=0)
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.

current_version_id = 2
dependency_map
format (names=0)
merge (other)
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 ()
Prepare 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)

class SCons.SConf.SConfBuildTask (tm, targets, top, node)
Bases: SCons.Taskmaster.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 ()
collect_node_states ()
display (message)
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)
Logs the original builder messages, given the SConfBuildInfo instance bi.

exc_clear ()
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)
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 ()
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 ()
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 ()

155
Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s callback methods.

fail_continue()
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()
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()
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()
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()
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()
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='node')

easy except SCons.SConf.SConfError(msg)
Bases: SCons.Errors.UserError

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

easy except SCons.SConf.SConfWarning
Bases: SCons.Warnings.SConsWarning

args
with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.
SCons API Documentation

SCons.SConf.SetBuildType (buildtype)
SCons.SConf.SetCacheMode (mode)
   Set the Configure cache mode. mode must be one of “auto”, “force”, or “cache”.
SCons.SConf.SetProgressDisplay (display)
   Set the progress display to use (called from SCons.Script)
class SCons.SConf.Streamer (orig)
   Bases: object
   ‘Sniffer’ for a file-like writable object. Similar to the unix tool tee.
   flush ()
  .getvalue ()
       Return everything written to orig since the Streamer was created.
   write (str)
   writelines (lines)
SCons.SConf._createConfigH (target, source, env)
SCons.SConf._createSource (target, source, env)
SCons.SConf._set_conftest_node (node)
SCons.SConf._stringConfigH (target, source, env)
SCons.SConf._stringSource (target, source, env)
SCons.SConf.createIncludesFromHeaders (headers, leaveLast, include_quotes="""")

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)
   do_not_store_info (filename, node)
   get_entry (filename)
       Fetch the specified entry attribute.
   merge ()
   set_entry (filename, obj)
       Set the entry.
   store_info (filename, node)

class SCons.SConsign.DB (dir)
   Bases: SCons.SConsign.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)
   do_not_store_info (filename, node)
   get_entry (filename)
       Fetch the specified entry attribute.
   merge ()
   set_entry (filename, obj)
       Set the entry.
   store_info (filename, node)
   write (sync=1)

class SCons.SConsign.Dir (fp=None, dir=None)
   Bases: SCons.SConsign.Base
   do_not_set_entry (filename, obj)
   do_not_store_info (filename, node)
   get_entry (filename)
       Fetch the specified entry attribute.
merge ()
    set_entry (filename, obj)
    Set the entry.
store_info (filename, node)
class SCons.SConsign.DirFile (dir)
    Bases: SCons.SConsign.Dir
    Encapsulates reading and writing a per-directory .sconsign file.
do_not_set_entry (filename, obj)
do_not_store_info (filename, node)
get_entry (filename)
    Fetch the specified entry attribute.
merge ()
    set_entry (filename, obj)
    Set the entry.
store_info (filename, node)
write (sync=1)
    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)
    Arrange for all signatures to be stored in a global .sconsign.db* file.
SCons.SConsign.ForDirectory
    alias of SCons.SConsign.DB
SCons.SConsign.Get_DataBase (dir)
SCons.SConsign.Reset ()
    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
    convert_from_sconsign (dir, name)
    convert_to_sconsign ()
    current_version_id = 2
    ninfo
SCons.SConsign.corrupt_dblite_warning (filename)
SCons.SConsign.current_sconsign_filename ()
SCons.SConsign.write ()

SCons.Subst module

SCons string substitution.
class SCons.Subst.CmdStringHolder (cmd, literal=None)
    Bases: collections.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
encode (encoding='utf-8', errors='strict')
endswith (suffix, start=0, end=922372036854775807)

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=922372036854775807)

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 ()

isalnum ()

isalpha ()

isascii ()

isdigit ()

isidentifier ()

islower ()

isnumeric ()

isspace ()

istitle ()

isupper ()

join (seq)

ljust (width, *args)

lower ()

lstrip (chars=None)

maketrans ()

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode
ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must
be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the
same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the
result.

partition (sep)

removeprefix (prefix, /)

removesuffix (suffix, /)

replace (old, new, maxsplit=-1)

rfind (sub, start=0, end=922372036854775807)

rindex (sub, start=0, end=922372036854775807)

rjust (width, *args)

rpartition (sep)

rsplit (sep=None, maxsplit=-1)

rstrip (chars=None)

split (sep=None, maxsplit=-1)

splitlines (keepends=False)

startswith (prefix, start=0, end=922372036854775807)

strip (chars=None)

swapcase ()

title ()

translate (*args)

upper ()

zfill (width)

class SCons.Subst.ListSubber (env, mode, conv, gvars)
Bases: collections.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)
add_to_current_word (x)
```

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)
Handle the “close strip” $) token.
copy ()
count (value) → integer -- return number of occurrences of value

expan(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)
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.
inset (i, item)
S.insert(index, value) – insert value before index

```
next_line ()
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 ()
Arrange for the next word to start a new word.
open strip (x)
```
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)
Substitute expansions in an argument or list of arguments.
This serves as a wrapper for splitting up a string into separate tokens.
this_word()

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()

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.

In practice, this might be a wash performance-wise, but it's a little cleaner conceptually...

_create_nodelist()

_gen_nodelist()

_return_nodelist()

class SCons.Subst.NullNodeList(*args, **kwargs)

Bases: SCons.Util.types.NullSeq

_instance

SCons.Subst.SetAllowableExceptions(*excepts)

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()

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 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.

class SCons.Subst.Targets_or_Sources(nl)

Bases: collections.UserList

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: collections.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()
    clear() → None -- remove all items from S
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]]) → 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()
    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={}, conv=None, overrides=False)
    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={}, conv=None, overrides=False)
    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.dict(target, source)
    Create a dictionary for substitution of special construction variables.
    This translates the following special arguments:
        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.Util module
SCons utility functions
SCons API Documentation

This package contains routines for use by other parts of SCons.

**class** SCons.Util.CLVar (initlist=None)

Bases: collections.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']
```

**_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:

```python
class Foo(Proxy):
    __str__ = Delegate('__str__')
```

**class** SCons.Util.DispatchingFormatter (formatters, default_formatter)

Bases: logging.Formatter

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.

```python
default_msec_format = "%s,%03d"
default_time_format = "%Y-%m-%d %H:%M:%S"
```

```python
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.
```

```python
formatException(ei)
    Format and return the specified exception information as a string.
    This default implementation just uses traceback.print_exception()
```

```python
formatMessage(record)
```

```python
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.
```

```python
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)
```

```python
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.
```

```python
class SCons.Util.LogicalLines(fileobj)
    Bases: object
    Wrapper class for the logical_lines method.
    Allows us to read all "logical" lines at once from a given file object.
    readlines()
```

```python
class SCons.Util.NodeList(initlist=None)
    Bases: collections.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.
```
>>> someList = NodeList(['  foo  ', '  bar  '])
>>> someList.strip()
['foo', 'bar']

__getattr__(name) → SCons.Util.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 objA 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 __*__() 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.

```python
def get()
    Retrieve the entire wrapped object
```

```python
SCons.Util.RegError
    alias of SCons.Util._NoError
SCons.Util.RegGetValue (root, key)
SCons.Util.RegOpenKeyEx (root, key)
```

```python
class SCons.Util.Selector
    Bases: collections.OrderedDict
    A callable ordered dictionary that maps file suffixes to dictionary values. We preserve the order in which items are added so that get_suffix() calls always return the first suffix added.
    clear () → None. Remove all items from od.
copy () → a shallow copy of od
dictkeys (value=None)
    Create a new ordered dictionary with keys from iterable and values set to value.

def get (key, default=None, /)
    Return the value for key if key is in the dictionary, else default.

def items ()
    → a set-like object providing a view on D’s items

def keys ()
    → a set-like object providing a view on D’s keys
move_to_end (key, last=True)
    Move an existing element to the end (or beginning if last is false).

```python
def pop (key=[], default=None) → v, remove specified key and return the corresponding value.
If the key is not found, return the default if given; otherwise, raise a KeyError.
Pairs are returned in LIFO order if last is true or FIFO order if false.

```python
def popitem (last=True)
    Remove and return a (key, value) pair from the dictionary.

```python
def 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
def 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
def values ()
    → an object providing a view on D’s values
```

```python
class SCons.Util.Split (arg) → list
    Returns a list of file names or other objects.
    If arg is a string, it will be split on strings of white-space characters 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 as a list containing just the object.

```python
>>> print(Split(" this is a string "))
['this', 'is', 'a', 'string']

```python
>>> print(Split(["stringlist"," preserving ", " spaces "]))
['stringlist', ' preserving ', ' spaces ']
```

```python
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.

```python
def write (arg)
    writelines (arg)
```

```python
class SCons.Util.UniqueList (initlist=None)
    Bases: collections.UserList
    A list which maintains uniqueness.
```
Uniquing is lazy: rather than being assured 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 “in” don’t have to eat the uniquing time.

```python
__make_unique ()
_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
    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 () → 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.Util.WhereIs (file, path=None, pathext=None, reject=None) → Optional [str]
Return the path to an executable that matches file.
    Searches the given path for file, respecting any filename extensions pathext (on the Windows platform only), and returns the full path to the matching command. If no command is found, return None.
    If path is not specified, os.environ[PATH] is used. If pathext is not specified, os.environ[PATHEXT] is used. Will not select any path name or names in the optional reject list.
```

exception SCons.Util._NoError
Bases: Exception
    `__args, **kwds`
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=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, s2) → bool
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, 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'>))
SCons.Util.flatten (obj, isinstance=<built-in function isinstance>, StringTypes=<class 'str'>,
    <class 'collections.UserString'>), SequenceTypes=<class 'list'>,
    <class 'tuple'>,
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>, StringType=(<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.
Same as flatten(), but it does not handle the single scalar case. This is slightly more efficient when one knows that the sequence to flatten can not be a scalar.

SCons.Util.get_native_path(path) → 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=0, showtags=False, margin=[0], visited=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
- **margin** – the format of the left margin to use for children of root. 1 results in a pipe, and 0 results in no pipe.
- **visited** – a dictionary of visited nodes in the current branch if prune is 0, or in the whole tree if prune is 1.
- **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=0, margin=[0], visited=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. 1 results in a pipe, and 0 results in no pipe.
- **visited** – a dictionary of visited nodes in the current branch if prune is 0, or in the whole tree if prune is 1.

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)

```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]]
```

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’s assumed that they do enjoy a total ordering. Then `unique()` will usually work in $O(N \log_2(N))$ time.

If that’s not possible either, the sequence elements must support equality-testing. Then `unique()` will usually work in quadratic time.

```python
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.

```python
SCons.Util.wait_for_process_to_die (pid)
```

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.Warnings module**

The SCons warnings framework.

```python
exception SCons.Warnings.CacheCleanupErrorWarning
   Bases: SCons.Warnings.SConsWarning
   args
   with_traceback ()
       Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.
exception SCons.Warnings.CacheVersionWarning
   Bases: SCons.Warnings.WarningOnByDefault
   args
   with_traceback ()
       Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.
exception SCons.Warnings.CacheWriteErrorWarning
```
Bases: SCons.Warnings.SConsWarning
args
with_traceback ()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.CorruptSConsignWarning
Bases: SCons.Warnings.WarningOnByDefault
args
with_traceback ()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.DependencyWarning
Bases: SCons.Warnings.SConsWarning
args
with_traceback ()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.DeprecatedDebugOptionsWarning
Bases: SCons.Warnings.MandatoryDeprecatedWarning
args
with_traceback ()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.DeprecatedMissingSConscriptWarning
Bases: SCons.Warnings.DeprecatedWarning
args
with_traceback ()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.DeprecatedOptionsWarning
Bases: SCons.Warnings.MandatoryDeprecatedWarning
args
with_traceback ()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.DeprecatedSourceCodeWarning
Bases: SCons.Warnings.FutureDeprecatedWarning
args
with_traceback ()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.DeprecatedWarning
Bases: SCons.Warnings.SConsWarning
args
with_traceback ()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.DevelopmentVersionWarning
Bases: SCons.Warnings.SConsWarning
args
with_traceback ()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.DuplicateEnvironmentWarning
Bases: SCons.Warnings.SConsWarning
args
with_traceback ()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.FortranCxxMixWarning
Bases: SCons.Warnings.LinkWarning
args
with_traceback ()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.FutureDeprecatedWarning
Bases: SCons.Warnings.SConsWarning
args
with_traceback ()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.FutureReservedVariableWarning
Bases: SCons.Warnings.WarningOnByDefault
args
with_traceback ()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.LinkWarning
Bases: SCons.Warnings.WarningOnByDefault
args
with_traceback ()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.MandatoryDeprecatedWarning
Bases: SCons.Warnings.DeprecatedWarning
args
with_traceback ()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.MisleadingKeywordsWarning
Bases: SCons.Warnings.WarningOnByDefault
args
with_traceback ()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.MissingSConscriptWarning
Bases: SCons.Warnings.WarningOnByDefault
args
with_traceback ()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.NoObjectCountWarning
Bases: SCons.Warnings.WarningOnByDefault
args
with_traceback ()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.NoParallelSupportWarning
Bases: SCons.Warnings.WarningOnByDefault
args
with_traceback ()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.PythonVersionWarning
Bases: SCons.Warnings.DeprecatedWarning
args
with_traceback ()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.ReservedVariableWarning
Bases: SCons.Warnings.WarningOnByDefault
args
with_traceback ()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.SConsWarning
Bases: SCons.Errors.UserError
args
with_traceback ()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.Warnings.StackSizeWarning
Bases: SCons.Warnings.WarningOnByDefault
args
SCons API Documentation

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

_exception SCons.Warnings.TargetNotBuiltWarning
    Bases: SCons.Warnings.SConsWarning
    args
    with_traceback ()
        Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

_exception SCons.Warnings.TaskmasterNeedsExecuteWarning
    Bases: SCons.Warnings.DeprecatedWarning
    args
    with_traceback ()
        Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

_exception SCons.Warnings.ToolQtDeprecatedWarning
    Bases: SCons.Warnings.DeprecatedWarning
    args
    with_traceback ()
        Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

_exception SCons.Warnings.VisualBasicMissingWarning
    Bases: SCons.Warnings.SConsWarning
    args
    with_traceback ()
        Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

_exception SCons.Warnings.VisualBasicVersionMismatch
    Bases: SCons.Warnings.WarningOnByDefault
    args
    with_traceback ()
        Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

_exception SCons.WarningsVisualStyleMissingWarning
    Bases: SCons.Warnings.SConsWarning
    args
    with_traceback ()
        Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

_exception SCons.Warnings.VisualBasicVersionMismatch
    Bases: SCons.Warnings.WarningOnByDefault
    args
    with_traceback ()
        Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

SCons.Warnings.enableWarningClass (clazz)
    Enables all warnings of type clazz or derived from clazz.

SCons.Warnings.process_warn_strings (arguments)
    Process requests to enable/disable warnings.
    The requests are strings passed to the –warn option or the SetOption('warn') function.
    An argument to this option should be of the form "warning-class" or "no-warning-class". The warning class is munged
    and has the suffix "Warning" added in order to get an actual class name from the classes above, which we need to
    pass to the {enable,disable}WarningClass() functions.
    For example, "deprecated" will enable the DeprecatedWarning class. "no-dependency" will disable the
    DependencyWarning class.
    As a special case, –warn=all and –warn=no-all will enable or disable (respectively) the base class of all SCons
    warnings.

SCons.Warnings.suppressWarningClass (clazz)
    Suppresses all warnings of type clazz or derived from clazz.

SCons.Warnings.warn (clazz, *args)
    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=True)
Set global _warningAsException flag.

**Parameters:** flag – value to set warnings-as-exceptions to [default: True]

**Returns:** The previous value.

SCons.cpp module

SCons C Pre-Processor module

SCons.cpp.CPP_to_Python(s)

Converts a C pre-processor expression into an equivalent Python expression that can be evaluated.

SCons.cpp.CPP_to_Python_Ops_Sub(m)

SCons.cpp.Cleanup_CPP_Expressions(ts)

class SCons.cpp.DumbPreProcessor(*args,**kw)

Bases: SCons.cpp.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.

__do_if_else_condition(condition)

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)

do_define(t)

Default handling of a #define line.

do_elif(t)

Default handling of a #elif line.

do_else(t)

Default handling of a #else line.

do endif(t)

Default handling of a #endif line.

do if(t)

Default handling of a #if line.

do ifndef(t)

Default handling of a #ifndef line.

do ifdef(t)

Default handling of a #ifdef line.

do import(t)

Default handling of a #import line.

do include(t)

Default handling of a #include line.

do include_next(t)

Default handling of a #include line.

do nothing(t)

Null method for when we explicitly want the action for a specific preprocessor directive to do nothing.

do undef(t)

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)
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)
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()
Pops the previous dispatch table off the stack and makes it the current one.
save()
Pushes the current dispatch table on the stack and re-initializes the current dispatch table to the default.
scons_current_file(t)
start_handling_includes(t=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)
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=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)
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)
do_define(t)
Default handling of a #define line.
do_elif(t)
Default handling of a #elif line.
do_else(t)
Default handling of a #else line.
do_endif(t)
Default handling of a #endif line.
do_if (t)
  Default handling of a #if line.
doifdef (t)
  Default handling of a #ifdef line.
doifndef (t)
  Default handling of a #ifndef line.
doincude (t)
  Default handling of a #include line.
doincude_next (t)
  Default handling of a #include line.
donone (t)
  Null method for when we explicitly want the action for a specific preprocessor directive to do nothing.
dounderf (t)
  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#import_file (t)
  Finds the #include file for a given preprocessor tuple.
initializ_result (fname)
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)
resolve#import (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.
restor ()
  Pops the previous dispatch table off the stack and makes it the current one.
save ()
  Pushes the current dispatch table on the stack and re-initializes the current dispatch table to the default.
scons_current_file (t)
start_handling_includes (t=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)
  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.

SCons.dblite module

dblite.py module contributed by Ralf W. Grosse-Kunstleve. Extended for Unicode by Steven Knight.
SCons API Documentation

SCons.dblite._exercise()

class SCons.dblite.dblite(file_base_name, flag, mode)

    Bases: object

Squirrel away references to the functions in various modules that we'll use when our __del__() method calls our sync() method during shutdown. We might get destroyed when Python is in the midst of tearing down the different modules we import in an essentially arbitrary order, and some of the various modules's global attributes may already be wiped out from under us.

See the discussion at:


_check_writable()

_open (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.getpreferredencoding(False) 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>
<tr>
<td>'U'</td>
<td>universal newline mode (deprecated)</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.

'U' mode is deprecated and will raise an exception in future versions of Python. It has no effect in Python 3. Use newline to control universal newlines mode.

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.

equarding 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.

enewline controls how universal newlines works (it only applies to text mode). It can be None, ‘
’, ‘n’, ‘r’, and ‘rn’. 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 closefd 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 byte array 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 a BytesIO can be used like a file opened in a binary mode.

_os_chmod (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, 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.

_os_chown (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, stat 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
_os_replace (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.

```python
_pickle_protocol = 4
```

```python
__shutil_copyfile (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
_time_time ()
```

Return the current time in seconds since the Epoch. Fractions of a second may be present if the system clock provides them.

```python
close ()
keys ()
sync ()
```

SCons.dblite.open (`file`, `flag=None`, `mode=438`)
SCons.compat package

SCons.exitfuncs module

Register functions which are executed when SCons exits for any reason.

SCons.exitfuncs._run_exitfuncs()
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)
register a function to be executed upon normal program termination
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.

```python
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)
    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)
class Attrs
    Bases: object
    shared
BuildInfo
Decider (function)
    GetTag (key)
        Return a user-defined tag.
NodeInfo
Tag (key, value)
    Add a user-defined tag.
    _add_child (collection, set, child)
        Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.
    _children_get ()
    _children_reset ()
    _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)
        Adds prerequisites
    add_source (source)
        Adds sources.
    add_to_implicit (deps)
    add_to_waiting_parents (node)
        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)
    add_wkid (wkid)
        Add a node to the list of kids waiting to be evaluated
    all_children (scan=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 ()
        A “builder” for aliases.
    builder
    builder_set (builder)
    built ()
        Called just after this node is successfully built.
cached
changed (node=None, allowcache=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=1)
Return a list of the node’s direct children, minus those that are ignored by this node.
children_are_up_to_date ()
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 ()
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 ()
convert ()
del_binfo ()

deletes
depends
depends_set
disambiguate (must_exist=None)
env
eval_set (env, safe=0)
executor
executor_cleanup ()
Let the executor clean up any cached information.
exists ()
Does this 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.
nodes - 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=1)
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 ()

get_target_scanner ()

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.

has_explicit_builder ()
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()
  Returns true if this node is an conftest node
is_derived()
  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()
  Always pass the string representation of a Node to the command interpreter literally.
is_sconscript()
  Returns true if this node is an sconscript
is_under(dir)
is_up_to_date()
  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()
  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()
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.
new_binfo()
new_ninfo()
ninfo
nocache
noclean
postprocess()
  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 ()
        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 ()
        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 ()
        Remove cached executor; forces recompute when needed.
retrieve_from_cache ()
        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.
xexists ()
        Does this node exist locally or in a repository?
scan ()
        Scan this node’s dependents for implicit dependencies.
scanner_key ()
sconsign ()
        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=1)
        Set the Node’s always_build value.
set_executor (executor)
        Set the action executor for this node.
set_explicit (is_explicit)
set_nocache (nocache=1)
        Set the Node’s nocache value.
set_noclean (noclean=1)
        Set the Node’s noclean value.
set_precious (precious=1)
        Set the Node’s precious value.
set_pseudo (pseudo=True)
        Set the Node’s precious value.
set_specific_source (source)
set_state (state)
side_effect
side_effects
sources
sources_set
state
store_info
str_for_display()
target_peers
visited()
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: SCons.Node.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)
      Restore the attributes from a pickled state.
bact
bactsig
bdepends
bdependsgigs
bimplicit
bimplicitsigs
bsources
bsourcesigs
current_version_id = 2
merge(other)
   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: collections.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
   __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)
restore the attributes from a pickled state.

current_version_id = 2
field_list = ['csig']
merge (other)
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.

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


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.

classAttrs

Bases: object

BuildInfo

alias of SCons.Node.BuildInfoBase

Decider (function)

GetTag (key)

Return a user-defined tag.

NodeInfo

alias of SCons.Node.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)

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

_abspath
_add_child (collection, set, child)
Adds 'child' to 'collection', first checking 'set' to see if it's already present.
_children_get ()
_children_reset ()
_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=True, source=False, strings=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)
  Adds prerequisites
add_source (source)
  Adds sources.
add_to_implicit (deps)
add_to_waiting_parents (node)
  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)
add_wkid (wkid)
  Add a node to the list of kids waiting to be evaluated
all_children (scan=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.
SCons.compat package

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)
built ()
   Called just after this node is successfully built.
cached
changed (node=None, allowcache=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=1)
   Return a list of the node’s direct children, minus those that are ignored by this node.
children_are_up_to_date ()
   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 ()
   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 ()
cwd
del_binfo ()
   Delete the build info from this node.
depends
depends_set
dir
disambiguate (must_exist=None)
duplicate
depends
dir
duplicate
env
env_set (env, safe=0)
executor
executor_cleanup ()
   Let the executor clean up any cached information.
exists ()
   Does this 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.
Reference to parent Node.FS object

fs

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

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_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

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_contents ()

Fetch the contents of the entry.

get_csig ()

get_dir ()

get_env ()

get_env_scanner (env, kw={})

get_executor (create=1)

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_relpah ()

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 ()
getmtime ()
getsize ()

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.

has_explicit_builder ()
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 ()
Returns true if this node is an conftest node

is_derived ()
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 ()
Always pass the string representation of a Node to the command interpreter literally.

is_sconscript ()
Returns true if this node is an sconscript

is_under (dir)

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

isdir ()
isfile ()
islink ()

linked

lstat ()

make_ready ()
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 ()
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.

name
new_binfo ()
new_ninfo ()
ninfo
nocache
noclean
postprocess ()

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 ()

Try to push a node into a cache

ref_count
release_target_info ()

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 ()

Remove cached executor; forces recompute when needed.

retrieve_from_cache ()

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 ()

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

sbuilder
SCons.compat package

scan()
    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=1)
    Set the Node’s always_build value.
set_executor(executor)
    Set the action executor for this node.
set_explicit(is_explicit)
set_local()
set_nocache(nocache=1)
    Set the Node’s nocache value.
set_noclean(noclean=1)
    Set the Node’s noclean value.
set_precious(precious=1)
    Set the Node’s precious value.
set_pseudo(pseudo=True)
    Set the Node’s precious value.
set_specific_source(source)
set_src_builder(builder)
    Set the source code builder for this node.
set_state(state)
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()
    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: SCons.Node.FS.Base
A class for directories in a file system.
class Attrs
Bases: object
shared
BuildInfo
Decider (function)
Dir (name, create=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 SCons.Node.FS.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)
   Add a user-defined tag.
__Rfindalldirs_key (pathlist)
__clearRepositoryCache (duplicate=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)
__str__ ()
   A Node.FS.Base object’s string representation is its path name.
_add_child (collection, set, child)
   Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.
_children_get ()
_children_reset ()
_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=True, source=False, strings=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.

_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.

_addRepository (dir)
_add_dependency (depend)
_add_ignore (depend)
_add_prerequisite (prerequisite)
_add_source (source)
_add_to_implicit (deps)
_add_to_waiting_parents (node)
_add_to_waiting_s_e (node)
_add_wkid (wkid)
_all_children (scan=1)
_alter_targets ()
_always_build
_attributes
_binfo
_build (**kw)
_add a node to the list of kids waiting to be evaluated

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

Return any corresponding targets in a variant directory.

A null “builder” for directories.

Called just after this node is successfully built.

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=1)
  Return a list of the node’s direct children, minus those that are ignored by this node.
children_are_up_to_date ()
  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 ()
  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 ()
contentsig
cwd
del_binfo ()
  Delete the build info from this node.
depends
depends_set
dir
dir_on_disk (name)
dirname
disambiguate (must_exist=None)
diskcheck_match ()
do_duplicate (src)
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=0)
executor
executor_cleanup ()
  Let the executor clean up any cached information.
exists ()
  Does this node exist?
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
SCons.compat package

return something that would actually work in a command line; it can return any kind of nonsense, so long as it does
not change.

def Reference to parent Node.FS object
getRepositories ()
  Returns a list of repositories for this directory.
get_abspath () \to \text{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=1)
  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 () \to \text{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_tpath ()
getmtime ()
getsize ()
glob (pathname, ondisk=True, source=False, strings=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()**

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()**

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()**

Returns true if this node is an conftest node

**is_derived()**

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()**

Always pass the string representation of a Node to the command interpreter literally.

**is_sconscript()**

Returns true if this node is an sconscript

**is_under(dir)**

**is_up_to_date()**

If any child is not up-to-date, then this directory isn’t, either.

**isdir()**

**isfile()**

**islink()**

**link(srcdir, duplicate)**

Set this directory as the variant directory for the supplied source directory.

**linked**

**lstat()**

**make_ready()**

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()**

**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.

name

new_binfo ()
new_ninfo ()
ninfo
nocache
noclean
on_disk_entries
postprocess ()

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 ()

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 ()

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 ()

Remove cached executor; forces recompute when needed.
retrieve_from_cache ()
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?

rfile ()
root
rstr ()
A Node.FS.Base object’s string representation is its path name.

sbuilder
scan ()
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=1)
Set the Node’s always_build value.

set_executor (executor)
Set the action executor for this node.

set_explicit (is_explicit)
set_local ()
set_nocache (nocache=1)
Set the Node’s nocache value.

set_noclean (noclean=1)
Set the Node’s noclean value.

set_precious (precious=1)
Set the Node’s precious value.

set_pseudo (pseudo=True)
Set the Node’s precious value.

set_specific_source (source)
set_src_builder (builder)
Set the source code builder for this node.

set_state (state)
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).

srmdir
srmdir_duplicate (name)

srmdir_find_file (filename)

srmdir_list ()
srncnode ()
Dir has a special need for srncnode()...if we have a srmdir attribute set, then that is our srncnode.
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_dir
visited()
Called just after this node has been visited (with or without a build).
waiting_parents
waiting_s_e
walk(func, arg)
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: SCons.Node.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)
Restore the attributes from a pickled state.
bact
bactsig
bdepends
bdependsligs
bimplicit
bimplicitsigs
bsources
bsourcesigs
current_version_id = 2
merge(other)
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
__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)
    Restore the attributes from a pickled state. The version is discarded.
convert(node, val)
current_version_id = 2
format(field_list=None, names=0)
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.

```python
def merge(other):
    # 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.compat package

```
_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=True, source=False, strings=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)
    Adds prerequisites
add_source (source)
    Adds sources.
add_to_implicit (deps)
add_to_waiting_parents (node)
    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)
add_wkid (wkid)
    Add a node to the list of kids waiting to be evaluated
all_children (scan=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)
built ()
    Called just after this node is successfully built.
cached
cachedir_csigh
cachesign
changed (node=None, allowcache=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()

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

#### children (scan=1)

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

#### children_are_up_to_date ()

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.


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

#### clear ()

#### clear_memoized_values ()

#### contentsig

#### cwd

Delete the build info from this node.

#### depends

#### depends_set

#### dir

#### dirname

#### disambiguate (must_exist=None)

#### diskcheck_match ()

#### duplicate

#### entries

#### env

#### env_set (env, safe=0)

#### executor

#### executor_cleanup ()

Let the executor clean up any cached information.

#### exists ()

Does this node exists?

#### explain ()

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=1)
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()
   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()
gmtime()
getsize()
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.
has_explicit_builder()
   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()
   Returns true if this node is an conftest node
is_derived()
   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()
   Always pass the string representation of a Node to the command interpreter literally.
is_sconscript()
   Returns true if this node is an sconscript
is_under(dir)
is_up_to_date()
   Default check for whether the Node is current: unknown Node subtypes are always out of date, so they will always
   get built.
isdir()
isfile()
islink()
linked
lstat()
make_ready()
   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)

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 ()

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 ()

Try to push a node into a cache

ref_count
ref_path (other)
release_target_info ()

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 ()

Remove cached executor; forces recompute when needed.

retrieve_from_cache ()

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?

rfile ()
We're a generic Entry, but the caller is actually looking for a File at this point, so morph into one.

root

rstr ()

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

sbuilder

scan ()

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=1)

Set the Node's always_build value.

set_executor (executor)

Set the action executor for this node.

set_explicit (is_explicit)

set_local ()

set_nocache (nocache=1)

Set the Node's nocache value.

set_noclean (noclean=1)

Set the Node's noclean value.

set_precious (precious=1)

Set the Node's precious value.

set_pseudo (pseudo=True)

Set the Node's precious value.

set_specific_source (source)

set_src_builder (builder)

Set the source code builder for this node.

set_state (state)

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 ()

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

waiting_parents
class SCons.Node.FS.EntryProxy (subject)

Bases: SCons.Util.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_relpath ()

__get_rsrcdir ()

Returns the directory containing the source node linked to this node via VariantDir(), or the directory of this node if not linked.

__get_rsrcnode ()

__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_srcnode ()

__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>, 'rsrcdir': <function EntryProxy.__get_rsrcdir>, 'rsrcpath': <function EntryProxy.__get_rsrcnode>, 'srcdir': <function EntryProxy.__get_srcdir>, 'srcpath': <function EntryProxy.__get_srcnode>, 'suffix': <function EntryProxy.__get_suffix>, '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.

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: SCons.Node.FS.LocalFS

Dir (name, directory=None, create=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=1)

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=1)
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

Globs

This is mainly a shim layer

PyPackageDir

Locate the directory of a given python module name

For example scons might resolve to Windows: C:Python27Libsite-packagesscons-2.5.1 Linux: /usr/lib/scons

This can be useful when we want to determine a toolpath based on a python module name

Repository(*dirs)

Specify Repository directories to search.

VariantDir (variant_dir, src_dir, duplicate=1)

Link the supplied variant directory to the source directory for purposes of building files.

_lookup (p, directory, fsclass, create=1)

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=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)
exists (path)
get_max_drift ()
get_root (drive)

Returns the root directory for the specified drive, creating it if necessary.

getcwd ()
gettexttime (path)
getsize (path)
isdir (path)
isfile (path)
islink (path)
link (src, dst)
listdir (path)
lstat (path)
makedirs (path, mode=511, exist_ok=False)
mkdir (path, mode=511)
open (path)
readlink (file)
rename (old, new)
scandir (path)
set_SConstruct_dir (dir)
set_max_drift (max_drift)
stat (path)
symlink (src, dst)
unlink (path)
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: SCons.Node.FS.Base
A class for files in a file system.
class Attrs
    Bases: object
    shared
BuildInfo
    alias of SCons.Node.FS.FileBuildInfo
Decider (function)
Dir (name, create=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 SCons.Node.FS.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)
    Add a user-defined tag.
_Rfindalldirs_key (pathlist)
    dmap_cache = {}
    dmap_sig_cache = {}
    setattr (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__ ()
    A Node.FS.Base object’s string representation is its path name.
_abspath
_add_child (collection, set, child)
    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
:return:
_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 ()
_createDir ()
_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)

Returns 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=True, source=False, strings=False)
_labspath
_local
_memo
_morph ()

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)

Adds prerequisites
add_source (source)

Adds sources.
add_to_implicit (deps)
add_to_waiting_parents (node)

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)
add_wkid (wkid)

Add a node to the list of kids waiting to be evaluated
all_children (scan=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)
built()
  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=False)
  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)
changed_since_last_build
changed_timestamp_match (target, prev_ni, repo_node=None)
  Return True if the timestamps don't match or if there is no previous timestamp
  @param target:
  @param prev_ni:
  Information about the node from the previous build
  @return:
changed_timestamp_newer (target, prev_ni, repo_node=None)
changed_timestamp_then_content (target, prev_ni, node=None)
  Used when decider 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 NodetoInfo 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=1)
  Return a list of the node's direct children, minus those that are ignored by this node.
children_are_up_to_date ()
  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

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

contentsig

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

convert_old_entry

convert_sig_attrs = ['bsourcesigs', 'bimplicit_sigs', 'bdependsigs']

cwd

del_binfo

Delete the build info from this node.

depends
depends_set
dir
dirname
disambiguate (must_exist=None)
diskcheck_match ()
do_duplicate (src)
duplicate
duplets
duplets_set
duplets_type

eval

env
eval_set (env, safe=0)
executor
executor_cleanup ()

Let the executor clean up any cached information.

events ()

Does this 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.
get_content_hash() → str
  Compute and return the hash for this file.
get_contents() → bytes
  Return the contents of the file as bytes.
get_contents_sig()
  A helper method for get_cachedir_bsig.
  It computes and returns the signature for this node’s contents.
get_csig() → str
  Generate a node’s content signature.
get_dir()
get_env()
get_env_scanner(env, kw={})
get_executor(create=1)
  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 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.
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_max_drift_csig() → Optional[str]
  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_relpath()
  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.

generate ()
generate_subst_proxy ()
generate_suffix ()
generate_target_scanner ()
generate_text_contents () → str

Return the contents of the file in text form.

This 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.

generate_timestamp () → int
generate_tpath ()
generatemtime ()
generate_size ()
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.

has_explicit_builder ()

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 ()

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_chunks = 65536
ignore
ignore_set
implicit
implicit_set
includes
is_conftest ()

Returns true if this node is an conftest node.

is_derived ()

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 ()

Always pass the string representation of a Node to the command interpreter literally.

is_sconscript ()
Returns true if this node is an sconscript

```
is_under (dir)
is_up_to_date ()
```

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 ()
isfile ()
islink ()
linked
lstat ()
make_ready ()
```

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.

```
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 ()
```

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

```
prepare ()
```

Prepare for this file to be created.

```
prerequisites
pseudo
push_to_cache ()
```

Try to push the node into a cache

```
ref_count
rel_path (other)
release_target_info ()
```

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

```
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.
SCons.compat package

rentry()
repositories
reset_executor()
    Remove cached executor; forces recompute when needed.
retrieve_from_cache()
    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()
    A Node.FS.Base object’s string representation is its path name.
sbuilder
scan()
    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=1)
    Set the Node’s always_build value.
set_executor(executor)
    Set the action executor for this node.
set_explicit(is_explicit)
set_local()
set_nocache(nocache=1)
    Set the Node’s nocache value.
set_noclean(noclean=1)
    Set the Node’s noclean value.
set_precious(precious=1)
    Set the Node’s precious value.
set_pseudo(pseudo=True)
    Set the Node’s precious value.
set_specific_source(source)
set_src_builder(builder)
    Set the source code builder for this node.
set_state(state)
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
SCons.compat package

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 ()
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: SCons.Node.BuildInfoBase
This is info loaded from sconsign.

Attributes unique to FileBuildInfo:

```python
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

```python
__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)
Restore the attributes from a pickled state.
```

```python
bact
bactsig
bdepends
bdependsigs
bimplicit
bimplicitsigs
bsources
bsourcesigs
convert_from_sconsign (dir, name)
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 ()
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.
```

```python
current_version_id = 2
dependency_map
format (names=0)
merge (other)
```
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 ()
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).

```python
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.
```

```python
class SCons.Node.FS.FileNodeInfo

    __getstate__ ()
    __setstate__ (state)
    convert (node, val)

    csig
    current_version_id = 2
    field_list = ['csig', 'timestamp', 'size']
    format (field_list=None, names=0)
    fs = None

    merge (other)
    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)
```

```python
SCons.Node.FS.LinkFunc (target, source, env)
```

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 …

```python
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)
exists (path)
getmtime (path)
getsize (path)
isdir (path)
isfile (path)
islink (path)
link (src, dst)
listdir (path)
lstat (path)
makedirs (path, mode=511, exist_ok=False)
mkdir (path, mode=511)
open (path)
readlink (file)
rename (old, new)
scandir (path)
stat (path)
symlink (src, dst)
unlink (path)

SCons.Node.FS.LocalString (target, source, env)
SCons.Node.FS.MkdirFunc (target, source, env)

class SCons.Node.FS.RootDir (drive, fs)
Bases: SCons.Node.FS.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 SCons.Node.FS.DirBuildInfo

Decider (function)

Dir (name, create=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 SCons.Node.FS.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)
    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)
  Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.

_children_get ()

_children_reset ()

_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=True, source=False, strings=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=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 ()
  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
SCons.compat package

absopath
addRepository (dir)
  Adds dependencies.
add_dependency (depend)
  Adds dependencies.
add_ignore (depend)
  Adds dependencies to ignore.
add_prerequisite (prerequisite)
  Adds prerequisites
add_source (source)
  Adds sources.
add_to_implicit (deps)
add_to_waiting_parents (node)
  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)
add_wkid (wkid)
  Add a node to the list of kids waiting to be evaluated
all_children (scan=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)
  A null “builder” for directories.
builder
builder_set (builder)
built ()
  Called just after this node is successfully built.
cached
cachedir_csig
cachesig
changed (node=None, allowcache=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=1)
  Return a list of the node’s direct children, minus those that are ignored by this node.
children_are_up_to_date ()
  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 ()
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 ()
contentsig
cwd
del_binfo ()
    Delete the build info from this node.
depends
depends_set
dir
dir_on_disk (name)
dirname
disambiguate (must_exist=None)
diskcheck_match ()
do_duplicate (src)
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 rentry_exists_on_disk
entry_labspath (name)
entry_path (name)
entry_tpath (name)
env
evns (env, safe=0)
exsector
executor
executor_cleanup ()
    Let the executor clean up any cached information.
exists ()
    Does this 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 () \rightarrow 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 ()
    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=1)
    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.

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

get_timestamp () \rightarrow \text{int}
Return the latest timestamp from among our children

get_tpath ()
gmtime ()
getsize ()
glob (\text{pathname, ondisk=\text{True}, source=\text{False}, strings=\text{False}, exclude=\text{None}}) \rightarrow \text{list}
Returns a list of Nodes (or strings) matching a pathname pattern.
Pathname patterns follow POSIX shell syntax:

\begin{verbatim}
*      matches everything
?      matches any single character
[seq]  matches any character in seq (ranges allowed)
[!seq] matches any char not in seq
\end{verbatim}

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.

\textbf{Parameters:}

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

has_builder ()
Return whether this Node has a builder or not.
In Boolean tests, this turns out to be a \textit{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 ()
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).
SCons.compat package

is_conftest()
  Returns true if this node is an conftest node
is_derived()
  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()
  Always pass the string representation of a Node to the command interpreter literally.
is_sconscript()
  Returns true if this node is an sconscript
is_under(dir)
is_up_to_date()
  If any child is not up-to-date, then this directory isn’t, either.
isdir()
isfile()
islink()
link(srcdir, duplicate)
  Set this directory as the variant directory for the supplied source directory.
linked
lstat()
make_ready()
  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.
name
new_binfo()
new_ninfo()
ninfo
nocache
noclean
on_disk_entries
path
postprocess()
  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 ()
    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 ()
    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 ()
    Remove cached executor; forces recompute when needed.
retrieve_from_cache ()
    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 ()
    Scan this node’s dependents for implicit dependencies.
scanner_key ()
    A directory does not get scanned.
scanners
scconsign ()
    Return the .scconsign file info for this directory.
sourced
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=1)
    Set the Node’s always_build value.
set.executor (executor)
Set the action executor for this node.

set_explicit (is_explicit)
set_local ()
set_nocache (nocache=1)
  Set the Node’s nocache value.
set_noclean (noclean=1)
  Set the Node’s noclean value.
set_precious (precious=1)
  Set the Node’s precious value.
set_pseudo (pseudo=True)
  Set the Node’s precious value.
set_specific_source (source)
set_src_builder (builder)
  Set the source code builder for this node.
set_state (state)
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 ()
srncnode ()
  Dir has a special need for srncnode()...if we have a srcdir attribute set, then that is our srncnode.
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 ()
  Called just after this node has been visited (with or without a build).
waiting_parents
waiting_s_e
walk (func, arg)
  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)
class  SCons.Node.FS._Null
SCons.compat package

Bases: object
SCons.Node.FS._classEntry
    alias of SCons.Node.FS.Entry
SCons.Node.FS._copy_func (fs, src, dest)
SCons.Node.FS._hardlink_func (fs, src, dst)
SCons.Node.FS._my_normcase (x)
SCons.Node.FS._my_splitdrive (p)
SCons.Node.FS._softlink_func (fs, src, dst)
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)
SCons.Node.FS.ignore_diskcheck_match (node, predicate, errorfmt)
SCons.Node.FS.initialize_do_splitdrive ()
SCons.Node.FS.invalidate_node_memos (targets)
    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)
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)
SCons.Node.FS.set_duplicate (duplicate)

SCons.Node.Python module

Python nodes.

class SCons.Node.Python.Value (value, built_value=None, name=None)
    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
Decider (function)
GetTag (key)
    Return a user-defined tag.
NodeInfo
Tag (key, value)
    Add a user-defined tag.
_add_child (collection, set, child)
SCons.compat package

    Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.
    
    _children_get ()
    _children_reset ()
    _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)
        Adds prerequisites
    add_source (source)
        Adds sources.
    add_to_implicit (deps)
    add_to_waiting_parents (node)
        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)
    add_wkid (wkid)
        Add a node to the list of kids waiting to be evaluated
    all_children (scan=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)
    built ()
        Called just after this node is successfully built.
    cached
    changed (node=None, allowcache=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=1)
    Return a list of the node's direct children, minus those that are ignored by this node.
children_are_up_to_date ()
    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 ()
    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 ()
del_binfo ()
    Delete the build info from this node.
depends
depends_set
disambiguate (must_exist=None)
env
eval_set (env, safe=0)
executor
executor_cleanup ()
    Let the executor clean up any cached information.
exists ()
    Does this 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=1)

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 ()
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 ()

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 ()

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
generate_set
implicit
implicit_set
SCons.compat package

includes
  is_conftest ()
    Returns true if this node is an conftest node
  isDerived ()
    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.
  isExplicit
  isLiteral ()
    Always pass the string representation of a Node to the command interpreter literally.
  isSconscript ()
    Returns true if this node is an sconscript
  isUnder (dir)
  isUpToDate ()
    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
  makeReady ()
    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 ()
  multipleSideEffectHasBuilder ()
    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.
  newBinfo ()
  newNinfo ()
  ninfo
  nocache
  noclean
  postprocess ()
    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
  pushToCache ()
    Try to push a node into a cache
  read ()
    Return the value. If necessary, the value is built.
  refCount
  releaseTargetInfo ()
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()
Remove a text representation, suitable for displaying to the user, of the include tree for the sources of this node.
reset_executor()
Remove cached executor; forces recompute when needed.
retrieve_from_cache()
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.
xists()
Does this node exist locally or in a repository?
scan()
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=1)
Set the Node’s always_build value.
set_executor(executor)
Set the action executor for this node.
set_explicit(is_explicit)
set_nocache(nocache=1)
Set the Node’s nocache value.
set_nocheck(noclean=1)
Set the Node’s noclean value.
set_precious(precious=1)
Set the Node’s precious value.
set_pseudo(pseudo=True)
Set the Node’s precious value.
set_specific_source(source)
set_state(state)
side_effect
side_effects
sources
sources_set
state
store_info
str_for_display()
target_peers
visited()
Called just after this node has been visited (with or without a build).
waiting_parents
waiting_s_e
wkids
write(built_value)
Set the value of the node.
SCons.compat package

class SCons.Node.Python.ValueBuildInfo
    Bases: SCons.Node.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)
        Restore the attributes from a pickled state.
    bact
    bactsig
    bdepends
    bdependsig
    bimplicitsig
    bsources
    bsourcesigs
    current_version_id = 2
    merge (other)
        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.

    __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)
        Restore the attributes from a pickled state.
    convert (node, val)
    csig
    current_version_id = 2
    field_list = ['csig']
    format (field_list=None, names=0)
    merge (other)
        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)
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)
class SCons.Node.BuildInfoBase
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 NodeInfo 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)  
Restore the attributes from a pickled state.
```

```
bact
bactsig
bdepends
bdependsigs
bimplicit
bimplicitsigs
bsources
bsourcesigs
current_version_id = 2
merge(other)  
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

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 SCons.Node.BuildInfoBase

Decider (function)

GetTag (key)

Return a user-defined tag.

NodelInfo

alias of SCons.Node.NodeInfoBase

Tag (key, value)

Add a user-defined tag.

_add_child (collection, set, child)

Adds ‘child’ to ‘collection’, first checking ‘set’ to see if it’s already present.

_children_get ()

_children_reset ()

_func_exists

_func_get_contents

_func_is_derived

_func_rexists

_func_target_from_source

_get_scanner (env, initial_scanner, root_node_scanner, kw)

_mem

_specific_sources

tags

add_dependency (depend)

Adds dependencies.

add_ignore (depend)

Adds dependencies to ignore.

add_prerequisite (prerequisite)
SCons.compat package

    Adds prerequisites
add_source (source)
    Adds sources.
add_to_implicit (deps)
add_to_waiting_parents (node)
    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)
add_wkid (wkid)
    Add a node to the list of kids waiting to be evaluated
all_children (scan=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)
built ()
    Called just after this node is successfully built.
cached
changed (node=None, allowcache=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=1)
    Return a list of the node’s direct children, minus those that are ignored by this node.
children_are_up_to_date ()
    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 ()
    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 ()
del_binfo ()
    Delete the build info from this node.
depends
depends_set
disambiguate (must_exist=None)
env
env_set (env, safe=0)
executor
executor_cleanup ()
    Let the executor clean up any cached information.
exists ()
    Does this 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 ()
get_csig ()
get_env ()
get_env_scanner (env, kw=*)
get_executor (create=1)
    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)
    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 ()
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.

has_explicit_builder ()
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 ()
Returns true if this node is an conftest node
is-derived ()
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 ()
Always pass the string representation of a Node to the command interpreter literally.

is_sconscript ()
Returns true if this node is an sconscript
is_up_to_date ()
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 ()
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 ()
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.

new_binfo ()
new_ninfo ()
ninfo
nocache
noclean
postprocess ()
  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 ()
  Try to push a node into a cache
ref_count
release_target_info ()
  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 ()
  Remove cached executor; forces recompute when needed.
retrieve_from_cache ()
  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 ()
  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=1)
  Set the Node’s always_build value.
set_executor (executor)
   Set the action executor for this node.

set_explicit (is_explicit)

set_nocache (nocache=1)
   Set the Node’s nocache value.

set_noclean (noclean=1)
   Set the Node’s noclean value.

set_precious (precious=1)
   Set the Node’s precious value.

set_pseudo (pseudo=True)
   Set the Node’s precious value.

set_specific_source (source)

set_state (state)

side_effect

side_effects

sources

sources_set

state

store_info

target_peers

visited ()
   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)
   Restore the attributes from a pickled state. The version is discarded.

class SCons.Node.NodeList (initlist=None)
Bases: collections.UserList

   _abc_impl = _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 \((i, \text{item})\)
\[
\mathrm{S}\.\text{insert}\left(index, \text{value}\right) \rightarrow \text{item} \quad \text{insert value before index}
\]
\[
\text{pop}([\text{index}]) \rightarrow \text{item} \quad \text{remove and return item at index (default last)}.
\]
\[
\text{Raise IndexError if list is empty or index is out of range.}
\]
remove \((\text{item})\)
\[
\mathrm{S}\.\text{remove}\left(\text{value}\right) \rightarrow \text{item} \quad \text{remove first occurrence of value. Raise ValueError if the value is not present.}
\]
reverse()
\[
\mathrm{S}\.\text{reverse}() \rightarrow \text{IN PLACE}
\]
sort(*\text{args}, **\text{kwds})

\begin{verbatim}
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()
\end{verbatim}

SCons.Node.changed_since_last_build_alias (node, target, prev_ni, repo_node=None)
SCons.Node.changed_since_last_build_entry (node, target, prev_ni, repo_node=None)
SCons.Node.changed_since_last_build_node (node, target, prev_ni, repo_node=None)
\[
\text{Must be overridden in a specific subclass to return True if this Node (a dependency) has changed since the last time it was used to build the specified target. prev_ni is this Node's state (for example, its file timestamp, length, maybe content signature) as of the last time the target was built.}
\]
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)
SCons.Node.changed_since_last_build_state_changed (node, target, prev_ni, repo_node=None)
SCons.Node.classname (obj)
SCons.Node.decide_source (node, target, prev_ni, repo_node=None)
SCons.Node.decide_target (node, target, prev_ni, repo_node=None)
SCons.Node.do_nothing (node, parent)
SCons.Node.do_nothing_node (node)
SCons.Node.exists_always (node)
SCons.Node.exists_base (node)
SCons.Node.exists_entry (node)
\[
\text{Return if the Entry exists. Check the file system to see what we should turn into first. Assume a file if there's no directory.}
\]
SCons.Node.exists_file (node)
SCons.Node.exists_none (node)
SCons.Node.get_children (node, parent)
SCons.Node.get_contents_dir (node)
\[
\text{Return content signatures and names of all our children separated by new-lines. Ensure that the nodes are sorted.}
\]
SCons.Node.get_contents_entry (node)
\[
\text{Fetch the contents of the entry. Returns the exact binary contents of the file.}
\]
SCons.Node.get_contents_file (node)
SCons.Node.get_contents_none (node)
SCons.Node.ignore_cycle (node, stack)
SCons.Node.is_derived_node (node)
\[
\text{Returns true if this node is derived (i.e. built).}
\]
SCons.Platform package

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)
SCons.Node.store_info_pass (node)
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)
SCons.Platform.aix.get_xlc (env, xlc=None, packages=[])
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)

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)
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)

SCons.Platform.virtualenv module

"Platform" support for a Python virtualenv.

SCons.Platform.virtualenv.ImportVirtualenv (env)

   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.Virtualenv ()

   Returns path to the virtualenv home if scons is executing within a virtualenv or None, if not.
SCons.Platform.virtualenv._enable_virtualenv_default ()
SCons.Platform.virtualenv._ignore_virtualenv_default ()
SCons.Platform.virtualenv._inject_venv_path (env, path_list=None)

   Modify environment such that SCons will take into account its virtualenv when running external tools.
SCons.Platform.virtualenv._inject_venv_variables (env)
SCons.Platform.virtualenv._is_path_in (path, base)

   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=[])
SCons.Platform package

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 PROCESSOR_ARCHITEW6432 or PROCESSOR_ARCHITECTURE environment variables).
SCons.Platform.win32.get_program_files_dir ()
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:

```
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:

```
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:
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":  # GCC requires double Windows slashes, let's use UNIX separator
        return WINPATHSEP_RE.sub(r"/\", arg)
    return arg

env["TEMPFILEARGESCFUNC"] = tempfile_arg_esc_func
```

_SCons.Scanner_ package

Submodules

_SCons.Scanner.C_ module

Dependency scanner for C/C++ code.

**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.CCPPConditionalScanner (*args, **kwargs)**

Bases: SCons.cpp.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.
SCons.Scanner package

This is the main public entry point.

_do_if_else_condition(condition)

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)

do_define (t)

Default handling of a #define line.

do_elif (t)

Default handling of a #elif line.

do_else (t)

Default handling of a #else line.

do_endif (t)

Default handling of a #endif line.

do_if (t)

Default handling of a #if line.

do_ifdef (t)

Default handling of a #ifdef line.

do ifndef (t)

Default handling of a #ifndef line.

do_import (t)

Default handling of a #import line.

do_include (t)

Default handling of a #include line.

do_include_next (t)

Default handling of a #include line.

do_nothing (t)

Null method for when we explicitly want the action for a specific preprocessor directive to do nothing.

do_undefined (t)

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)

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)

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 ()

Pops the previous dispatch table off the stack and makes it the current one.

save ()

Pushes the current dispatch table on the stack and re-initializes the current dispatch table to the default.

scons_current_file (t)

start_handling_includes (t= 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)
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.

recursive_nodes (nodes)
select (node)

class SCons.Scanner.C.SConsCPPScanner (*args, **kwargs)
Bases: SCons.cpp.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)
Common logic for evaluating the conditions on #if, #ifdef and #ifndef lines.

_match_tuples (tuples)
_parse_tuples (contents)
_all_include (t)
do_define (t)

_do_elif (t)
Default handling of a #elif line.
do_else (t)
Default handling of a #else line.
do_endif (t)
Default handling of a #endif line.
do_if (t)
Default handling of a #if line.
do_ifdef (t)
Default handling of a #ifdef line.
do_ifndef (t)
Default handling of a #ifndef line.
do_import (t)
Default handling of a #import line.
do_include (t)
Default handling of a #include line.
do_include_next (t)
Default handling of a #include line.
do_nothing (t)
Null method for when we explicitly want the action for a specific preprocessor directive to do nothing.
do_undef (t)
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(jing 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)
   Pre-processes a file contents.
   Is used by tests
process_contents (contents)
   Pre-processes a file contents.
read_file (file)
   Pre-processes a file.
   This is the main internal entry point.
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 ()
   Pops the previous dispatch table off the stack and makes it the current one.
save ()
   Pushes the current dispatch table on the stack and re-initializes the current dispatch table to the default.
scons_current_file (t)
start_handling_includes (t=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)
   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.

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: SCons.Scanner.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
static recurse_all_nodes(nodes)
static recurse_no_nodes(nodes)
add_scanner(skey, scanner)
add_skey(skey)
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)
SCons.Scanner.Dir.scan_in_memory(node, env, path=())
“Scans” a Node.FS.Dir for its in-memory entries.
SCons.Scanner.Dir.scan_on_disk(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.

```python
class SCons.Scanner.Fortran.F90Scanner(name, suffixes, path_variable, use_regex, incl_regex, def_regex, *args, **kwargs)
    Bases: SCons.Scanner.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.
- `path` – tuple of paths from the `path_function`
SCons.Scanner package

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)
add_skey (skey)
    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=source_dir, target=None)
scan (node, env, path=())
select (node)
static sort_key (include)

SCons.Scanner.Fortran.FortranScan (path_variable='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.Scanner.Java module

SCons.Scanner.Java.JavaScanner ()
    Scanner for .java files.
    New in version 4.4.
SCons.Scanner.Java._collect_classes (classlist, dirname, files)
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.
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: SCons.Scanner.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:

```bash
$ 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[\\texttt{class|style}]{}  
FIXME: also look for the argument of \texttt{bibliographystyle}{}

```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
_latex_names (include_type, filename)
static _recurse_all_nodes (nodes)
static _recurse_no_nodes (nodes)
add_scanner (skey, scanner)
add_skey (skey)
    Add a skey to the list of skyes
canonical_text (text)

    Standardize an input TeX-file contents.
```

**Currently:**

- removes comments, unwrapping comment-wrapped lines,
- env\_variables = [\texttt{TEXINPUTS}, \texttt{BIBINPUTS}, \texttt{BSTINPUTS}, \texttt{INDEXSTYLE}]
- find\_include (include, source\_dir, path)
- get\_skyes (env=None)
- keyword\_paths = ['addbibresource': \texttt{BIBINPUTS}, 'addglobalbib': \texttt{BIBINPUTS}, 'addsectionbib': \texttt{BIBINPUTS}, 'bibliography': \texttt{BIBINPUTS}, 'bibliographystyle': \texttt{BSTINPUTS}, 'include': \texttt{TEXINPUTS}, 'includegraphics': \texttt{TEXINPUTS}, 'input': \texttt{TEXINPUTS}, 'listinputlisting': \texttt{TEXINPUTS}, 'makeindex': \texttt{INDEXSTYLE}, 'usepackage': \texttt{TEXINPUTS}]
- path (env, dir=None, target=None, source=None)
- scan (node, subdir=')
- 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)
SCons.Scanner package

```python
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

- alias of `SCons.Scanner.LaTeX._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 `SCons.Scanner.ScannerBase`

**class SCons.Scanner.Classic(name, suffixes, path_variable, regex, *args, **kwargs)**

Bases: `SCons.Scanner.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.

```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.

- `static _recurse_all_nodes(nodes)`
SCons.Scanner package

```python
static _recurse_no_nodes(nodes)
add_scanner(skey, scanner)
add_skey(skey)
    Add a key 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)

class SCons.Scanner.ClassicCPP(name, suffixes, path_variable, regex, *args, **kwargs)
    Bases: SCons.Scanner.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)
add_skey(skey)
    Add a key to the list of skeys
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)
sort_key(include)

class SCons.Scanner.Current(*args, **kwargs)
    Bases: SCons.Scanner.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)
add_skey(skey)
    Add a key to the list of skeys
get_skeys(env=None)
```
SCons.Scanner package

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

```python
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.
```

```python
class SCons.Scanner.ScannerBase (function, name='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')
```
SCons.Scanner package

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)
add_skey(skey)
    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: SCons.Scanner.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.)

```python
static _recurse_all_nodes(nodes)
static _recurse_no_nodes(nodes)
add_scanner(skey, scanner)
add_skey(skey)
    Add a skey to the list of skeys
```
SCons.Script package

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

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

SCons.Script package

Submodules

SCons.Script.Interactive module

SCons interactive mode.

class SCons.Script.Interactive.SConsInteractiveCmd (**kw)
    Bases: cmd.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)
    _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=None)
        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)
    completedefault (*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)
        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)
    do_build (argv)
        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)
        exit Exit SCons interactive mode.
    do_help (argv)
        help [COMMAND] Prints help for the specified COMMAND. ‘h’ and ‘?’ are synonyms.
    do_shell (argv)
        shell [COMMANDLINE] Execute COMMANDLINE in a subshell. ‘sh’ and ‘!’ are synonyms.
    do_version (argv)
        version Prints SCons version information.
    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.
precmd (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)

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: SCons.Taskmaster.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 ()
    display (message)
        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=2)
    exc_clear ()
    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)
    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 ()
    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 ()
    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 ()
    Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s
callback methods.
    fail_continue ()
    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 ()
    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 ()
    Make a task ready for execution
    make_ready_all ()
    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 ()
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='node')

class SCons.Script.Main.CleanTask (tm, targets, top, node)
Bases: SCons.Taskmaster.AlwaysTask
An SCons clean task.
LOGGER = None
_abc_impl = <_abc._abc_data object>
_clean_targets (remove=True)
_exception_raise ()
  Raises a pending exception that was recorded while getting a Task ready for execution.
_get_files_to_clean ()
_no_exception_to_raise ()
display (message)
  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 ()
  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)
  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 ()
  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 ()
  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 ()
  Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s callback methods.
fail_continue ()
  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()
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().

fs_delete(path, pathstr, remove=True)

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

make_ready()
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()
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()
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()
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.

remove()

show()

trace_message(node, description='node')

class SCons.Script.Main.CountStats
do_append(label)
do_nothing(*args, **kw)
do_print()
enable(outfp)
class SCons.Script.Main.FakeOptionParser
Bases: object
A do-nothing option parser, used for the initial OptionsParser variable.
During normal SCons operation, the OptionsParser is created right away by the main() function. Certain tests 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)
    values = <SCons.Script.Main.FakeOptionParser.FakeOptionValues object>
SCons.Script.Main.GetBuildFailures ()
SCons.Script.Main.GetOption (name)
class SCons.Script.Main.MemStats
do_append (label)
do_nothing (*args, **kw)
do_print ()
enable (outfp)
SCons.Script.Main.PrintHelp (file=None)
SCons.Script.Main.Progress (*args, **kw)
class SCons.Script.Main.Progressor (obj, interval=1, file=None, overwrite=False)
    Bases: object
count = 0
erase_previous ()
prev = ''
replace_string (node)
spinner (node)
string (node)
target_string = '$TARGET'
write (s)
class SCons.Script.Main.QuestionTask (tm, targets, top, node)
    Bases: SCons.Taskmaster.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 ()
display (message)
        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 ()
        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)
        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 ()
        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()
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()
Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s callback methods.

fail_continue()
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()
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()
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()
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()
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()
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='node')

debug SCons.Script.Main.SConsPrintHelpException
SCons.Script package

Bases: Exception
        args
        with_traceback ()
        Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

SCons.Script.Main.SetOption (name, value)
class SCons.Script.Main.Stats
        Bases: object
        do_nothing (*args, **kw)
        enable (outfp)
class SCons.Script.Main.TreePrinter (derived=False, prune=False, status=False, sLineDraw=False)
        Bases: object
        display ()
        get_all_children (node)
        get_derived_children (node)

SCons.Script.Main.ValidateOptions (throw_exception=False) \rightarrow None
        Validate options passed to SCons on the command line.
        If you call this after you set all your command line options with AddOption(), it will verify that all command line options
        are valid. So 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 (bool) – (Optional) Should this function raise an error if there’s an invalid
            option on the command line, or issue a message and exit with error status.

        Raises:
            SConsBadOptionError – If throw_exception is True and there are invalid options on
            command line.

New in version 4.5.0.

SCons.Script.Main._SConstruct_exists (dirname=’, repositories=[], filelist=None)
        This function checks that an SConstruct file exists in a directory. If so, it returns the path of the file. By default, it
        checks the current directory.

SCons.Script.Main._build_targets (fs, options, targets, target_top)
SCons.Script.Main._create_path (plist)
SCons.Script.Main._exec_main (parser, values)
SCons.Script.Main._load_all_site_scons_dirs (topdir, verbose=False)
        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 ()
        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)
        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)
        Handle syntax errors. Print out a message and show where the error occurred.

SCons.Script.Main._scons_user_error (e)
        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)
        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)
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 package

SCons.Script.Main.main()
SCons.Script.Main.path_string(label, module)
SCons.Script.Main.python_version_deprecated(version=sys.version_info(major=3, minor=10, micro=10, releaselevel='final', serial=0))
SCons.Script.Main.python_version_string()
SCons.Script.Main.python_version_unsupported(version=sys.version_info(major=3, minor=10, micro=10, releaselevel='final', serial=0))
SCons.Script.Main.revert_io()
SCons.Script.Main.test_load_all_site_scons_dirs(d)
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 --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.

---

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

---

### SCons.Script.SConsOptions.SConsIndentedHelpFormatter

Bases: optparse.IndentedHelpFormatter

Variables used to indicate that invalid command line options were specified

---

```python
Exception.with_traceback() – 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)
```

---

**format_heading(heading)**

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.

### 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)
Formats the usage message.
indent ()
set_long_opt_delimiter (delim)
set_parser (parser)
set_short_opt_delimiter (delim)
store_option_strings (parser)
class SCons.Script.SConsOptions.SConsOption (*opts, **attrs)
Bases: optparse.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_kwargs', 'help', 'metavar']
CHECK_METHODS = [Option._check_action, Option._check_type, Option._check_choice, Option._check_dest, Option._check_nargs, Option._check_callback, 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': check_choice, 'complex': check_builtin, 'float': check_builtin, 'int': check_builtin, 'long': check_builtin}
_check_action ()
_check_callback ()
_check_choice ()
_check_const ()
_check_dest ()
_check_nargs ()
_check_nargs_optional ()
_check_opt_strings (opts)
_check_type ()
_setattrs (attrs)
_set_opt_strings (opts)
SCons.Script package

```python
check_value(opt, value)
convert_value(opt, value)
get_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: optparse.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(opt)
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.

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: optparse.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, i.e. 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 : [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)
```

268
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.

_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.

d_error(msg)

overridden OptionValueError exception handler

de_exit(status=0, msg=None)

de_expand_prog_name(s)

de_format_description(formatter)

de_format_epilog(formatter)

de_format_help(formatter=None)

de_format_option_help(formatter=None)

de_get_default_values()

de_get_description()

de_get_option(opt_str)

de_get_option_group(opt_str)

de_get_prog_name()

de_get_usage()

de_get_version()

has_option(opt_str)

_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
SCons.Script package

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_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 ()
    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: optparse.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 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 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 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

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


SCons.Script.SConsOptions.diskcheck_convert (value)
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).

```python
AppendUnique (delete_existing=False, **kw)
```

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.

```python
Builder (**kw)
```

```python
CacheDir (path, custom_class=None)
```

```python
Clean (targets, files)
```

```python
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.

```python
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.

```python
Configure (*args, **kw)
```

```python
Decider (function)
```

```python
Default (*targets)
```

```python
Depends (target, dependency)
```

Explicitly specify that ‘target’s depend on ‘dependency’.

```python
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

```python
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.

```python
Dir (name, *args, **kw)
```

```python
Dump (key=None, format='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”)

```python
static EnsurePythonVersion (major, minor)
```

Exit abnormally if the Python version is not late enough.

```python
static EnsureSConsVersion (major, minor, revision=0)
```

Exit abnormally if the SCons version is not late enough.

```python
Entry (name, *args, **kw)
```

```python
Environment (**kw)
```

```python
Execute (action, *args, **kw)
```

Directly execute an action through an Environment

```python
static Exit (value=0)
```

```python
Export (*vars, **kw)
```

```python
File (name, *args, **kw)
```

```python
FindFile (file, dirs)
```

```python
FindInstalledFiles ()
```

returns the list of all targets of the Install and InstallAs Builder.

```python
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

```python
FindSourceFiles (node='.') → list
Return a list of all source files.
```

```python
Flatten (sequence)
GetBuildPath (files)
static GetLaunchDir ()
GetOption (name)
Glob (pattern, ondisk=True, source=False, strings=False, exclude=None)
Help (text, append=False)
Ignore (target, dependency)
Ignore a dependency.
Import (*vars)
Literal (string)
Local (*targets)
MergeFlags (args, unique=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().

**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.

```python
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.
```

```python
ParseConfig (command, function=None, unique=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)
```

```python
ParseDepends (filename, must_exist=None, only_one=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.
```

```python
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**)
Prepend (**kw**)
Prepend values to construction variables in an Environment.
The variable is created if it is not already present.
PrependENVPath (name, newpath, envname='ENV', sep=':', delete_existing=True)
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=False, **kw**)
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**)
PyPackageDir (modulename)
RemoveMethod (function)
Removes the specified function’s MethodWrapper from the added_methods list, so we don’t re-bind it when making a clone.
Replace (**kw**)
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**)
Requires (target, prerequisite)
Specify that ‘prerequisite’ must be built before ‘target’, (but ‘target’ does not actually depend on ‘prerequisite’ and need not be rebuilt if it changes).
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)
Scanner (*args, **kw)
SetDefault (**kw)
SetOption (name, value)
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) → SCons.Tool.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=1)

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)
_changed_content (dependency, target, prev_ni, repo_node=None)
_changed_source (dependency, target, prev_ni, repo_node=None)
_changed_timestamp_match (dependency, target, prev_ni, repo_node=None)
_changed_timestamp_newer (dependency, target, prev_ni, repo_node=None)
_changed_timestamp_then_content (dependency, target, prev_ni, repo_node=None)
_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 ()

update (other)
   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)
   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)

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.

```python
get_CacheDir()
```

Fetch the builder with the specified name from the environment.

```python
get_factory(factory, default='File')
```

Return a factory function for creating Nodes for this construction environment.

```python
get_scanner(key)
```

Find the appropriate scanner given a key (usually a file suffix).

```python
get_src_sig_type()
get_tgt_sig_type()
gvars()
```

Emulates the `items()` method of dictionaries.

```python
keys()
```

Emulates the `keys()` method of dictionaries.

```python
lvars()
```

```python
scanner_map_delete(kw=None)
```

Delete the cached scanner map (if we need to).

```python
setdefault(key, default=None)
```

`setdefault()` method of dictionaries.

```python
subst(string, raw=0, target=None, source=None, conv=None, executor=None, overrides=False)
```

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=0, target=None, source=None)
```

```python
subst_list(string, raw=0, target=None, source=None, conv=None, executor=None, overrides=False)
```

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, turning EntryProxies into Nodes and leaving Nodes (and other objects) as-is.

```python
subst_target_source(string, raw=0, target=None, source=None, conv=None, executor=None, overrides=False)
```

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
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.

```python
values()
```

Emulates the `values()` method of dictionaries.

---

SCons.Script package

276
SCons.Script package

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, must_exist=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 value. On first warning, print a deprecation message.

Parameters:
- f (str) – path of missing configuration file
- must_exist (bool) – if true, fail. If false, but not None, allow the file to be missing. The default is None, which means issue the warning. The default is deprecated.

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=False)
class SCons.Script.TargetList (initlist=None)
    Bases: collections.UserList
    _abc_impl = <abc._abc_data object>
    _add_Default (list)
    _clear ()
    _do_nothing (*args, **kw)
    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)
SCons.Script package

SCons.Script._Add_Targets(tlist)
SCons.Script._Get_Default_Targets(d, fs)
SCons.Script._Set_Default_Targets(env, tlist)
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=1)

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.

```python
class SCons.Taskmaster.Job.InterruptState
    Bases: object
    set()

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.
    _reset_sig_handler()
        Restore the signal handlers to their previous state (before the call to _setup_sig_handler()).
    _setup_sig_handler()
        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>>)
        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)
```

Note: The above text is a natural representation of the content in the image.
SCons.Taskmaster package

Bases: object
class State (value):
  Bases: enum.Enum
  An enumeration.
  COMPLETED = 3
  READY = 0
  SEARCHING = 1
  STALLED = 2
class Worker (owner):
  Bases: threading.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()
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.

_adjust_stack_size()
_restore_stack_size(prev_size)
_setup_logging()
_start_workers()
_work()
start()
trace_message(message)

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()
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)

put(task)
Put task into request queue.

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

class SCons.Taskmaster.AlwaysTask (tm, targets, top, node)
Bases: SCons.Taskmaster.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 ()
display (message)
   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 ()
   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)
   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 ()
   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 ()
   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 ()
   Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s callback methods.

fail_continue ()
   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 ()
   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 ()
   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 ()
   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 ()
   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 ()
   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='node')

class SCons.Taskmaster.OutOfDateTask (tm, targets, top, node)
Bases: SCons.Taskmaster.Task
LOGGER = None
_abcs_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 ()
display(message)
   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 ()
   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)
   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 ()
   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 ()
   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 ()
   Called when the task has been successfully executed and the Taskmaster instance doesn't want to call the Node's
   callback methods.
fail_continue ()
   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 ()
   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 ()
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 ()
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 ()
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='node')
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.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 ()
display (message)
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 ()
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)
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().
evacuated()
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.
evacuated_with_callbacks()
Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s callback methods.
Evacuated 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.
evacuated_without_callbacks()
Called when the task has been successfully executed and the Taskmaster instance doesn’t want to call the Node’s callback methods.
fail_continue()
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()
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().
fail()
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()
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()
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.

```python
trace_message (node, description='node')
```

class SCons.Taskmaster.Taskmaster (targets=\[], tasker=None, order=None, trace=None)

Bases: object

The Taskmaster for walking the dependency DAG.

```python
_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.

```python
_validate_pending_children ()
```

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:

```
  |  |  |
  ^ |  |
```

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) ------------+
```

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.
SCons.Taskmaster package

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)
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()
Stops the current build completely.

sm_trace_node (node)
will_not_build (nodes, node_func=<function Taskmaster.<lambda>>)
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 ()
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)
class SCons.Tool.Tool (name, toolpath=None, **kwargs)
SCons.Tool package

```
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 Tool 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)
Searches the list of associated Tool modules for one that exists, and applies that to the construction environment.
remove_methods (env)
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)
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='$_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='$_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=False) → Optional[str]
Find the location of a tool using various means.
Mainly for windows where tools aren’t all installed in /usr/bin, etc.
```
SCons.Variables package

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.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.Variables.EnumVariable(EnumVariable (key, help, default, allowed_values, map={}, ignorecase=0) →
Tuple[str, str, str, Callable, Callable]
Return a tuple describing an enumaration 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:

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 module

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.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

```
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

```
opts = Variables()
opts.Add(
    PackageVariable(
        key='x11',
        help='use X11 installed here (yes = search some places)',
        default='yes'
    )
)
...
```

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")
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.

class SCons.Variables.Variables (files=None, args=None, is_global=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)
SCons.Variables package

Add \((\text{key, } *\text{args, } **\text{kwargs}) \to \text{None})\)

Adds an option.

**Parameters:**
- \text{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.

**Keyword Arguments:**
- \text{*args} – optional positional arguments, corresponding to the four named kwargs below.
- \text{help} – help text for the options (Default value = “”)
- \text{default} – default value for option (Default value = None)
- \text{validator} – function called to validate the option’s value (Default value = None)
- \text{converter} – function to be called to convert the option’s value before putting it in the environment. (Default value = None)
- \text{**kwargs} – arbitrary keyword arguments used by the variable itself.

AddVariables \((*\text{optlist}) \to \text{None})\)

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 \((\text{env, key, help, default, actual, aliases}=\text{None}) \to \text{str})\)

Generates help text for the options.

**Parameters:**
- \text{env} – an environment that is used to get the current values of the options.
- \text{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 \(\text{filename, env}) \to \text{None})\)

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:**
- \text{filename} – Name of the file to save into
- \text{env} – the environment get the option values from

UnknownVariables () \to \text{dict}

Returns unknown variables.

Identifies options that were not known, declared options in this object.

Update \((\text{env, args}=\text{None}) \to \text{None})\)

Updates an environment with the option variables.

**Parameters:**
- \text{env} – the environment to update.
- \text{args} (optional) – a dictionary of keys and values to update in \text{env}. If omitted, uses the variables from the commandline.

_do_add \((\text{key, help}=\text{None}, \text{default}=\text{None}, \text{validator}=\text{None}, \text{converter}=\text{None}, **\text{kwargs}) \to \text{None})\)

aliasfmt = "\text{\n%s: %s in default: %s in actual: %s in aliases: %s in}"
Indices and Tables

```python
fmt = '%s
 default: %s
 actual: %s
'
instance = None
keys () → list
    Returns the keywords for the options.
```

Indices and Tables

- genindex
- modindex
- search
Index

__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)
__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_srcdir() (SCons.Node.FS.EntryProxy method)
__get_srcnode() (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.File method)
(SCons.Node.FS.RootDir method)
(SCons.Script.SConsOptions.SConsValues method)
(SCons.Util.NodeList method)
(SCons.Util.Proxy method)

__getitem__() (SCons.Util.NodeList 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)
__resetDuplicate() (SCons.Node.FS.Dir method)


__str__() (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.Environment.OverrideEnvironment method)
(SCons.Script.SConscript.SConsEnvironment method)
_hardlink_func() (in module SCons.Node.FS)
_Have() (in module SCons.Conftest)
_ignore_virtualenv_default() (in module SCons.Platform.virtualenv)
_init_parsing_state() (SCons.Script.SConsOptions.SConsOptionParser method)
_init_special() (SCons.Environment.Base method)
  (SCons.Environment.OverrideEnvironment method)
  (SCons.Environment.SubstitutionEnvironment method)
  (SCons.Script.SConscript.SConsEnvironment method)
  (SCons.Taskmaster.Job.Worker attribute)
_inject_venv_path() (in module SCons.Platform.virtualenv)
_inject_venv_variables() (in module SCons.Platform.virtualenv)
_instance (SCons.Subst.NullNodeList attribute)
_is_path_in() (in module SCons.Platform.virtualenv)
_labspath (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)
_lang2suffix() (in module SCons.Conftest)
_latex_names() (SCons.Scanner.LaTeX.LaTeX method)
_load_all_site_scons_dirs() (in module SCons.Script.Main)
_load_site_scons_dir() (in module SCons.Script.Main)
_local (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)
_LogFailed() (in module SCons.Conftest)
_lookup() (SCons.Node.FS.FS method)
_lookup_abs() (SCons.Node.FS.RootDir method)
_lookupDict (SCons.Node.FS.RootDir attribute)
_main() (in module SCons.Script.Main)
_match_long_opt() (SCons.Script.SConsOptions.SConsOptionParser method)
_match_tuples() (SCons.cpp.DumbPreProcessor method)
  (SCons.cpp.PreProcessor method)
  (SCons.Scanner.C.SConsCPPConditionalScanner method)
  (SCons.Scanner.C.SConsCPPScanner method)
_memo (SCons.Executor.Executor attribute)
  (SCons.Executor.Null 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)
_morph() (SCons.Executor.Null method)
  (SCons.Node.FS.Dir method)
  (SCons.Node.FS.File method)
  (SCons.Node.FS.RootDir method)
_my_normcase() (in module SCons.Node.FS)
_my_splitdrive() (in module SCons.Node.FS)
_no_exception_to_raise() (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)
_node_errors() (in module SCons.Builder)
_NoError
_null (class in SCons.Action)
_Null (class in SCons.Builder)
null (in module SCons.Builder)

_open() (SCons.dblite.dblite method)
_os_chmod() (SCons.dblite.dblite method)
_os_chown() (SCons.dblite.dblite method)
_os_replace() (SCons.dblite.dblite method)
_parse_tuples() (SCons.cpp.DumbPreProcessor method)
_process_long_opt() (SCons.Script.SConsOptions.SConsOptionParser method)
_process_short_opts() (SCons.Script.SConsOptions.SConsOptionParser method)
_process_tuples() (SCons.cpp.DumbPreProcessor method)
_proxy (SCons.Node.FS.Base attribute)
_recurs_all_nodes() (SCons.Scanner.Classic static method)
_readconfig() (SCons.CacheDir.CacheDir method)
_recurse_all_nodes() (SCons.Scanner.Classic static method)
_recurse_no_nodes() (SCons.Scanner.Classic static method)
_relay_path_key() (SCons.Node.FS.Dir method)
_remove_list() (in module SCons.Subst)

_path (SCons.Node.FS.Base attribute)
_path_elements (SCons.Node.FS.Base attribute)
_path_list (class in SCons.PathList)
_pickle_dump() (SCons.dblite.dblite static method)
_pickle_protocol (SCons.dblite.dblite attribute)
_populate_option_list() (SCons.Script.SConsOptions.SConsOptionParser method)
_print_cmd_str() (SCons.Platform.TempFileMunge method)
_process_args() (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)

CacheCleanupErrorWarning
  (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)
CacheDebug() (SCons.CacheDir.CacheDir method)
CacheDir (class in SCons.CacheDir)
CacheDir() (SCons.Environment.Base method)
  (SCons.Environment.OverrideEnvironment method)
  (SCons.Script.SConscript.SConsEnvironment method)
cachedir_csig (SCons.Node.FS.Dir attribute)
  (SCons.Node.FS.Entry attribute)
  (SCons.Node.FS.File attribute)
  (SCons.Node.FS.RootDir attribute)
cachepath() (SCons.CacheDir.CacheDir method)
CachePushFunc() (in module SCons.CacheDir)
CacheRetrieveFunc() (in module SCons.CacheDir)
CacheRetrieveString() (in module SCons.CacheDir)
cachesig (SCons.Node.FS.Dir attribute)
  (SCons.Node.FS.Entry attribute)
  (SCons.Node.FS.File attribute)
  (SCons.Node.FS.RootDir attribute)
CacheVersionWarning
CacheWriteErrorWarning
CallableSelector (class in SCons.Builder)
caller_stack() (in module SCons.Debug)
caller_trace() (in module SCons.Debug)
canonical_text() (SCons.Scanner.LaTeX.LaTeX method)
capitalize() (SCons.Subst.CmdStringHolder method)
case_sensitive_suffixes() (in module SCons.Util)
casefold() (SCons.Subst.CmdStringHolder method)
CConditionalScanner() (in module SCons.Scanner.C)
center() (SCons.Subst.CmdStringHolder 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)
changed_content() (SCons.Node.FS.File method)
changed_since_last_build (SCons.Node.Alias.Alias 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)
changed_since_last_build_alias() (in module SCons.Node)
changed_since_last_build_entry() (in module SCons.Node)
changed_since_last_build_node() (in module SCons.Node)
changed_since_last_build_python() (in module SCons.Node)
changed_since_last_build_state_changed() (in module SCons.Node)
changed_state() (SCons.Node.FS.File method)
cached_timestamp_match() (SCons.Node.FS.File method)
cached_timestamp_newer() (SCons.Node.FS.File method)
cached_timestamp_then_content() (SCons.Node.FS.File method)
chdir() (SCons.Node.FS.FS 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)
CHECK_METHODS
  (SCons.Script.SConsOptions.SConsOption attribute)
check_value()
  (SCons.Script.SConsOptions.SConsOption method)
check_values()
  (SCons.Script.SConsOptions.SConsOptionParser method)
CheckBuilder() (in module SCons.Conftest)
CheckCC() (in module SCons.Conftest)
  (in module SCons.SConf)
CheckCHeader() (in module SCons.SConf)
CheckContext (class in SCons.SConf)
CheckCXX() (in module SCons.Conftest)
  (in module SCons.SConf)
CheckCXXHeader() (in module SCons.SConf)
CheckDeclaration() (in module SCons.Conftest)
  (in module SCons.SConf)
CheckFunc() (in module SCons.Conftest)
  (in module SCons.SConf)
CheckHeader() (in module SCons.Conftest)
  (in module SCons.SConf)
CheckLib() (in module SCons.Conftest)
  (in module SCons.SConf)
CheckLibWithHeader() (in module SCons.SConf)
CheckMember() (in module SCons.Conftest)
  (in module SCons.SConf)
CheckProg() (in module SCons.Conftest)
  (in module SCons.SConf)
CheckSHCC() (in module SCons.Conftest)
  (in module SCons.SConf)
CheckSHCXX() (in module SCons.Conftest)
  (in module SCons.SConf)
  (in module SCons.SConf)
CheckType() (in module SCons.Conftest)
  (in module SCons.SConf)
CheckTypeSize() (in module SCons.Conftest)
  (in module SCons.SConf)
  (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)
children_are_up_to_date() (SCons.Node.Alias.Alias method)
chmod() (SCons.Node.FS.FS method)
  (SCons.Node.FS.LocalFS method)
chmod_func() (in module SCons.Defaults)
chmod_strfunc() (in module SCons.Defaults)
Classic (class in SCons.Scanner)
ClassicCPP (class in SCons.Scanner)
classname() (in module SCons.Scanner)
Clean() (SCons.Environment.Base method)
  (SCons.Environment.OverrideEnvironment method)
  (SCons.Script.SConscript.SConsEnvironment 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)
createIncludesFromHeaders() (in module SCons.SConf)
CreateJarBuilder() (in module SCons.Tool)
CreateJavaClassDirBuilder() (in module SCons.Tool)
CreateJavaClassFileBuilder() (in module SCons.Tool)
CreateJavaHBuilder() (in module SCons.Tool)
createLoadableModuleBuilder() (in module SCons.Tool)
createObjBuilders() (in module SCons.Tool)
createProgBuilder() (in module SCons.Tool)
createSharedLibBuilder() (in module SCons.Tool)
createStaticLibBuilder() (in module SCons.Tool)
CScanner() (in module SCons.Scanner.C)
    (SCons.Node.FS.FileNodeInfo attribute)
    (SCons.Node.FS.RootDir attribute)
Current (class in SCons.Scanner)
current_sconsign_filename() (in module SCons.SConsign)
    (SCons.Node.FS.DirBuildInfo attribute)
    (SCons.Node.FS.DirNodeInfo attribute)
    (SCons.Node.FS.FileBuildInfo attribute)
    (SCons.Node.FS.FileNodeInfo attribute)
    (SCons.Node.FS.RootDir attribute)
    (SCons.Node.NodeInfoBase attribute)
    (SCons.SConf.SConfBuildInfo attribute)
    (SCons.SConsign.SConsignBuildInfo attribute)
cwd (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)

D
D (class in SCons.Scanner.D)
DB (class in SCons.SConsign)
dblite (class in SCons.dblite)
decide_source() (in module SCons.Node)
decide_target() (in module SCons.Node)
Decider() (SCons.Environment.Base method)
    (SCons.Environment.OverrideEnvironment method)
    (SCons.Node.FS.Base method)
    (SCons.Node.FS.File method)
    (SCons.Node.FS.Entry method)
    (SCons.Node.FS.File method)
    (SCons.Node.FS.RootDir method)
    (SCons.Node.FS.RootDir method)
    (SCons.SConsign.SConsignBuildInfo attribute)
dedent() (SCons.Script.SConsOptions.SConsIndentedHelpFormatter method)
default() (SCons.Script.Interactive.SConsInteractiveCmd method)
Default() (SCons.Script.SConscript.SConsEnvironment method)
default_copy_from_cache() (in module SCons.Environment)
default_copy_to_cache() (in module SCons.Environment)
default_decide_source() (in module SCons.Environment)
default_decide_target() (in module SCons.Environment)
default_exitstatfunc() (in module SCons.Action)
default_msec_format (SCons.Util.DispatchingFormatter attribute)
default_time_format (SCons.Util.DispatchingFormatter attribute)
DefaultEnvironment() (in module SCons.Defaults)
DefaultEnvironmentCall (class in SCons.Script.SConscript)
DefaultToolList() (in module SCons.Platform)
Define() (SCons.SConf.SConfBase method)
    (SCons.Node.FS.Base method)
    (SCons.Node.FS.Dir method)
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.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.FileNodeInfo 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.SConsOptionGroup method)
  (SCons.Script.SConsOptions.SConsOptionParser method)
format_option_help() (SCons.Script.SConsOptions.SConsOptionGroup 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.Environment.BuilderDict class method)
  (SCons.Node.Alias.AliasNameSpace class method)
  (SCons.Util.Selector method)
FS (class in SCons.Node.FS)
  (SCons.Node.FS.Base attribute)
  (SCons.Node.FS.Dir attribute)
  (SCons.Node.FS.DirNodeInfo attribute)
  (SCons.Node.FS.Entry attribute)
  (SCons.Node.FS.File attribute)
  (SCons.Node.FS.FileNodeInfo attribute)
  (SCons.Node.FS.RootDir attribute)
fs_delete() (SCons.Script.Main.CleanTask method)
func_shorten() (in module SCons.Debug)
function_name() (SCons.Action.FunctionAction method)
FunctionAction (class in SCons.Action)
FunctionEvaluator (class in SCons.cpp)
FutureDeprecatedWarning
FutureReservedVariableWarning
generate() (in module SCons.Platform.aix)
   (in module SCons.Platform.cygwin)
   (in module SCons.Platform.darwin)
   (in module SCons.Platform.hpux)
   (in module SCons.Platform.irix)
   (in module SCons.Platform.os2)
   (in module SCons.Platform.posix)
   (in module SCons.Platform.sunos)
   (in module SCons.Platform.win32)
GenerateHelpText() (SCons.Variables.Variables method)
genstring() (SCons.Action._ActionAction method)
   (SCons.Action.ActionBase method)
   (SCons.Action.CommandGeneratorAction method)
   (SCons.Action.FunctionAction method)
   (SCons.Action.LazyAction method)
   (SCons.Action.ListAction method)
get() (SCons.Builder.CallableSelector method)
   (SCons.Builder.CompositeBuilder method)
   (SCons.Builder.DictCmdGenerator method)
   (SCons.Builder.DictEmitter method)
   (SCons.Builder.OverrideWarner method)
   (SCons.Environment.Base method)
   (SCons.Environment.BuilderDict method)
   (SCons.Environment.OverrideEnvironment method)
   (SCons.Environment.SubstitutionEnvironment method)
   (SCons.Node.Alias.AliasNameSpace method)
   (SCons.Node.FS.EntryProxy method)
   (SCons.Script.SConscript.SConsEnvironment method)
   (SCons.Taskmaster.Job.ThreadPool method)
   (SCons.Util.Proxy method)
   (SCons.Util.Selector method)
   (SCons.Node.FS.RootDir method)
get_action_list() (SCons.Executor.Executor method)
   (SCons.Executor.Null method)
get_action_side_effects() (SCons.Executor.Executor method)
   (SCons.Executor.Null method)
get_action_targets() (SCons.Executor.Executor method)
   (SCons.Executor.Null method)
generate_all() (SCons.Executor.Executor method)
get_all_children() (SCons.Executor.Executor method)
   (SCons.Executor.Null method)
get_all_prerequisites() (SCons.Executor.Executor method)
   (SCons.Executor.Null method)
get_all_rdirs() (SCons.Node.FS.Dir method)
   (SCons.Node.FS.RootDir method)
get_all_sources() (SCons.Executor.Executor method)
   (SCons.Executor.Null method)
get_all_targets() (SCons.Executor.Executor method)
   (SCons.Executor.Null method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Node.FS.Dir method)
get_all() (SCons.Node.FS.Entry method)
get_all() (SCons.Node.FS.File method)
get_all() (SCons.Node.FS.RootDir method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCons.Executor.Executor method)
get_all() (SCon...
get_default_values()
(SCons.Script.SConsOptions.SConsOptionParser
method)

get_DefaultEnvironmentProxy() (in module
SCons.Script.SConscript)

get Derived_children() (SCons.Script.Main.TreePrinter
method)

get_description()
(SCons.Script.SConsOptions.SConsOptionGroup
method)

get_dir() (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_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)

(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_factory() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Script.SConscript.SConsEnvironment
method)

get_found_includes() (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)

get Implicit_deps() (SCons.Action.CommandAction
method)
(SCons.Action.CommandGeneratorAction method)
(SCons.Action.FunctionAction method)
(SCons.Action.LazyAction method)
(SCons.Action.ListAction method)
(SCons.Executor.Executor 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_internal_path() (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_kw() (SCons.Executor.Executor method)

get_labspath() (SCons.Node.FS.Base method)
(SCons.Node.FS.Dir method)
(SCons.Node.FS.Entry 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_chunksize (SCons.Node.FS.File attribute)

Help() (SCons.Script.SConscript.SConsEnvironment method)
HelpFunction() (in module SCons.Script)

hit_ratio (SCons.CacheDir.CacheDir property)
LocalFS (class in SCons.Node.FS)
LocalString() (in module SCons.Node.FS)
Log() (SCons.SConf.CheckContext method)
LOGGER (SCons.SConf.SConfBuildTask attribute)
   (SCons.Script.Main.BuildTask attribute)
   (SCons.Script.Main.CleanTask attribute)
   (SCons.Script.Main.QuestionTask attribute)
   (SCons.Taskmaster.AlwaysTask attribute)
   (SCons.Taskmaster.OutOfDateTask attribute)
   (SCons.Taskmaster.Task attribute)
logical_lines() (in module SCons.Util)
LogicalLines (class in SCons.Util)
logInstanceCreation() (in module SCons.Debug)
lookup() (SCons.Node.Alias.AliasNameSpace method)
lower() (SCons.Subst.CmdStringHolder method)
lstat() (SCons.Node.FS.Base method)
   (SCons.Node.FS.Dir method)
   (SCons.Node.FS.Entry method)
   (SCons.Node.FS.File method)
   (SCons.Node.FS.FS method)
   (SCons.Node.FS.LocalFS method)
   (SCons.Node.FS.RootDir method)
lstrip() (SCons.Subst.CmdStringHolder method)
lvars (SCons.Executor.Executor attribute)
   (SCons.Executor.Null attribute)
lvars() (SCons.Environment.Base method)
   (SCons.Environment.OverrideEnvironment method)
   (SCons.Environment.SubstitutionEnvironment method)
   (SCons.Script.SConscript.SConscriptEnvironment method)
M
main() (in module SCons.Script.Main)
make_path_relative() (in module SCons.Util)
   (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_ready_all() (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)
make_ready_current() (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)
makedirs() (SCons.Node.FS.FS method)
   (SCons.Node.FS.LocalFS method)
maketrans() (SCons.Subst.CmdStringHolder method)
MandatoryDeprecatedWarning
match_splitext() (in module SCons.Builder)
memory() (in module SCons.Debug)
MemStats (class in SCons.Script.Main)
   (SCons.Node.BuildInfoBase method)
   (SCons.Node.FS.DirBuildInfo method)
   (SCons.Node.FS.DirNodeInfo method)
   (SCons.Node.FS.FileBuildInfo method)
   (SCons.Node.FS.FileNodeInfo method)
move_func() (in module SCons.Defaults)
move_to_end() (SCons.Builder.CallableSelector method)
                   (SCons.Builder.DictCmdGenerator method)
                   (SCons.Builder.DictEmitter method)
                   (SCons.Util.Selector method)
mro() (SCons.compat.NoSlotsPyPy method)
MSVCError
multiple_side_effect_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)
must_be_same() (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)

name (SCons.Node.FS.Base attribute)
                   (SCons.Node.FS.Dir attribute)
                   (SCons.Node.FS.Entry attribute)
                   (SCons.Node.FS.EntryProxyAttributeError attribute)
                   (SCons.Node.FS.File attribute)
                   (SCons.Node.FS.RootDir attribute)
                   (SCons.Taskmaster.Job.Worker property)
native_id (SCons.Taskmaster.Job.NewParallel.Worker property)
                   (SCons.Taskmaster.Job.Worker property)
NeedConfigHBuilder() (in module SCons.SConf)
needs_execute() (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)
normpath_match() (in module SCons.Node.FS)
                   (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)

NewParallel (class in SCons.Taskmaster.Job)
NewParallel.State (class in SCons.Taskmaster.Job)
NewParallel.Worker (class in SCons.Taskmaster.Job)
next_line() (SCons.Subst.ListSubber method)
next_task() (SCons.Taskmaster.Taskmaster method)
next_word() (SCons.Subst.ListSubber 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)
NLWrapper (class in SCons.Subst)
no_batch_key() (SCons.Action._ActionAction method)
  (SCons.Action.ActionBase method)
  (SCons.Action.CommandGeneratorAction method)
  (SCons.Action.FunctionAction method)
  (SCons.Action.LazyAction method)
  (SCons.Action.ListAction method)
NO_DEFAULT_VALUE (SCons.Script.SConsOptions.SConsIndentedHelpFormatter attribute)
no_next_candidate() (SCons.Taskmaster.Taskmaster method)
no_tlb() (in module SCons.Scanner.RC)
  (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)
Node (class in SCons.Node)
Node.Attrs (class in SCons.Node)
node_conv() (in module SCons.PathList)
  (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)
NodeInfoBase (class in SCons.Node)
NodeList (class in SCons.Node)
  (class in SCons.Util)
nohelp (SCons.Script.Interactive.SConsInteractiveCmd attribute)
non_sconf_nodes (SCons.SConf.SConfBuildTask attribute)
NoObjectCountWarning
NoParallelSupportWarning
NoSlotsPyPy (class in SCons.compat)
NoSubstitutionProxy() (in module SCons.Environment)
Null (class in SCons.Executor)
NullCmdGenerator (class in SCons.Defaults)
NullEnvironment (class in SCons.Executor)
nullify() (SCons.Executor.Executor method)
NullNodeList (class in SCons.Subst)
O
obj (SCons.Node.FS.EntryProxyAttributeError attribute)
on_disk_entries (SCons.Node.FS.Dir attribute)
(SCons.Node.FS.Entry attribute)
(SCons.Node.FS.File attribute)
(SCons.Node.FS.RootDir attribute)
onecmd() (SCons.Script.Interactive.SConsInteractiveCmd method)
only_dirs() (in module SCons.Scanner.Dir)
open() (in module SCons.dblite)
(SCons.Node.FS.FS method)
(SCons.Node.FS.LocalFS method)
open_strip() (SCons.Subst.ListSubber method)
OutOfDateTask (class in SCons.Taskmaster)
Override() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Environment.SubstitutionEnvironment method)
(SCons.Script.SConscript.SConsEnvironment method)
OverrideEnvironment (class in SCons.Environment)
overridelist (SCons.Executor.Executor attribute)
(SCons.Executor.Null attribute)
OverrideWarner (class in SCons.Builder)
PackageVariable() (in module SCons.Variables)
parse_args() (SCons.Script.SConsOptions.SConsOptionParser method)
ParseConfig() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Script.SConscript.SConsEnvironment method)
ParseDepends() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Script.SConscript.SConsEnvironment method)
ParseFlags() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Environment.SubstitutionEnvironment method)
(SCons.Script.SConscript.SConsEnvironment method)
 parseline() (SCons.Script.Interactive.SConsInteractiveCmd method)
Parser() (in module SCons.Script.SConsOptions)
partition() (SCons.Subst.CmdStringHolder method)
path (SCons.Node.FS.RootDir attribute)
path() (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)
 path_string() (in module SCons.Script.Main)
PathList() (in module SCons.PathList)
PDFLaTeXScanner() (in module SCons.Scanner.LaTeX)
piped_env_spawn() (in module SCons.Platform.posix)
piped_spawn() (in module SCons.Platform.win32)
Platform() (in module SCons.Platform)
(SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Script.SConscript.SConsEnvironment method)
platform_default() (in module SCons.Platform)
platform_module() (in module SCons.Platform)
PlatformSpec (class in SCons.Platform)
pop() (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.Script.TargetList method)
(SCons.Subst.ListSubber method)
(SCons.Subst.Targets_or_Sources method)
(SCons.Util.CLVar method)
(SCons.Util.NodeList 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.Main.CleanTask 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.UniqueList method)
remove_methods() (SCons.Tool.ToolInitializer method)
remove_option()
(SCons.Script.SConsOptions.SConsOptionGroup method)
(SCons.Script.SConsOptions.SConsOptionParser method)
RemoveMethod() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Script.SubstitutionEnvironment method)
(SCons.Script.SConscript.SConsEnvironment method)
removeprefix() (SCons.Subst.CmdStringHolder method)
removesuffix() (SCons.Subst.CmdStringHolder method)
rename() (SCons.Node.FS.FS method)
(SCons.Node.FS.LocalFS method)
rename_module() (in module SCons.compat)
render_include_tree() (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)
render_tree() (in module SCons.Util)
rentry() (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)
rentry_exists_on_disk() (SCons.Node.FS.Dir method)
(SCons.Node.FS.RootDir method)
reparse_local_options()
(SCons.Script.SConsOptions.SConsOptionParser method)
Replace() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Script.SConscript.SConsEnvironment method)
replace() (SCons.Subst.CmdStringHolder method)
replace_string() (SCons.Script.Main.Progressor method)
ReplaceIxes() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Script.SConscript.SConsEnvironment method)
repositories (SCons.Node.FS.Dir attribute)
(SCons.Node.FS.Entry attribute)
(SCons.Node.FS.File attribute)
(SCons.Node.FS.RootDir attribute)
Repository() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Node.FS.FS method)
(SCons.Script.SConscript.SConsEnvironment method)
Requires() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Node.FS.FS method)
(SCons.Script.SConscript.SConsEnvironment method)
ReservedVariableWarning
Reset() (in module SCons.SConsign)
(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.File method)
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.Builder.BuilderBase 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)
set_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)
set_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)
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)
    (SCons.Taskmaster.Job.Worker method)
setdefault() (SCons.Builder.CallableSelector method)
    (SCons.Builder.DictCmdGenerator method)
    (SCons.Builder.DictEmitter 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)
spawn() (in module SCons.Platform.win32)
spawnve() (in module SCons.Platform.win32)
SpecialAttrWrapper (class in SCons.Subst)
spinner() (SCons.Script.Main.Progressor method)
Split() (in module SCons.Util)
  (SCons.Environment.Base method)
  (SCons.Environment.OverrideEnvironment method)
  (SCons.Script.SConscript.SConsEnvironment method)
split() (SCons.Subst.CmdStringHolder method)
splitext() (in module SCons.Util)
  (SCons.Builder.BuilderBase 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)
srmdir_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)
    (SCons.Scanner.C.SConsCPPConditionalScanner method)
    (SCons.Scanner.C.SConsCPPScanner method)
startswith() (SCons.Subst.CmdStringHolder method)
stat() (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.RootDir method)
srcdir_duplicate() (SCons.Node.FS.Dir method)
  (SCons.Node.FS.RootDir attribute)
  (SCons.Node.FS.RootDir 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)
StaticObjectEmitter() (in module SCons.Defaults)
Stats (class in SCons.Script.Main)

write() (in module SCons.SConsign)

writelines() (SCons.SConf.Streamer 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.hpx
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.Variables
SCons.Variables.BoolVariable
SCons.Variables.EnumVariable
SCons.Variables.ListVariable
SCons.Variables.PackageVariable
SCons.Variables.PathVariable
SCons.Warnings