<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCons.Script.SConsOptions module</td>
<td>108</td>
</tr>
<tr>
<td>SCons.Script.SConscript module</td>
<td>114</td>
</tr>
<tr>
<td>Module contents</td>
<td>121</td>
</tr>
<tr>
<td>SCons.Tool package</td>
<td>122</td>
</tr>
<tr>
<td>Module contents</td>
<td>122</td>
</tr>
<tr>
<td>SCons.Variables package</td>
<td>127</td>
</tr>
<tr>
<td>Submodules</td>
<td>127</td>
</tr>
<tr>
<td>SCons.Variables.BoolVariable module</td>
<td>127</td>
</tr>
<tr>
<td>SCons.Variables.EnumVariable module</td>
<td>127</td>
</tr>
<tr>
<td>SCons.Variables.ListVariable module</td>
<td>128</td>
</tr>
<tr>
<td>SCons.Variables.PackageVariable module</td>
<td>128</td>
</tr>
<tr>
<td>SCons.VariablesPathVariable module</td>
<td>129</td>
</tr>
<tr>
<td>Module contents</td>
<td>129</td>
</tr>
<tr>
<td>SCons.compat package</td>
<td>131</td>
</tr>
<tr>
<td>Module contents</td>
<td>131</td>
</tr>
<tr>
<td>Submodules</td>
<td>131</td>
</tr>
<tr>
<td>SCons.Action module</td>
<td>131</td>
</tr>
<tr>
<td>SCons.Builder module</td>
<td>138</td>
</tr>
<tr>
<td>SCons.CacheDir module</td>
<td>144</td>
</tr>
<tr>
<td>SCons.Conftest module</td>
<td>145</td>
</tr>
<tr>
<td>SCons.Debug module</td>
<td>147</td>
</tr>
<tr>
<td>SCons.Defaults module</td>
<td>148</td>
</tr>
<tr>
<td>SCons.Environment module</td>
<td>149</td>
</tr>
<tr>
<td>SCons.Errors module</td>
<td>163</td>
</tr>
<tr>
<td>SCons.Executor module</td>
<td>165</td>
</tr>
<tr>
<td>SCons.Job module</td>
<td>170</td>
</tr>
<tr>
<td>SCons.Memoize module</td>
<td>173</td>
</tr>
<tr>
<td>SCons.PathList module</td>
<td>174</td>
</tr>
<tr>
<td>SCons.SConf module</td>
<td>175</td>
</tr>
<tr>
<td>SCons.SConsign module</td>
<td>181</td>
</tr>
<tr>
<td>SCons.Subst module</td>
<td>183</td>
</tr>
<tr>
<td>SCons.Taskmaster module</td>
<td>189</td>
</tr>
<tr>
<td>Generic Taskmaster module for the SCons build engine.</td>
<td>189</td>
</tr>
<tr>
<td>Taskmaster</td>
<td>189</td>
</tr>
<tr>
<td>SCons.Util module</td>
<td>196</td>
</tr>
<tr>
<td>SCons.Warnings module</td>
<td>204</td>
</tr>
<tr>
<td>SCons.cpp module</td>
<td>208</td>
</tr>
<tr>
<td>SCons.dblite module</td>
<td>212</td>
</tr>
<tr>
<td>SCons.exitfuncs module</td>
<td>215</td>
</tr>
<tr>
<td>SCons.compat package</td>
<td>216</td>
</tr>
<tr>
<td>Module contents</td>
<td>216</td>
</tr>
<tr>
<td>Package</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------</td>
</tr>
<tr>
<td>SCons.Node package</td>
<td>216</td>
</tr>
<tr>
<td>Submodules</td>
<td>216</td>
</tr>
<tr>
<td>SCons.Node.Alias module</td>
<td>216</td>
</tr>
<tr>
<td>SCons.Node.FS module</td>
<td>225</td>
</tr>
<tr>
<td>SCons.Node.Python module</td>
<td>282</td>
</tr>
<tr>
<td>Module contents</td>
<td>290</td>
</tr>
<tr>
<td>SCons.Platform package</td>
<td>300</td>
</tr>
<tr>
<td>Submodules</td>
<td>300</td>
</tr>
<tr>
<td>SCons.Platform.aix module</td>
<td>300</td>
</tr>
<tr>
<td>SCons.Platform.cygwin module</td>
<td>300</td>
</tr>
<tr>
<td>SCons.Platform.darwin module</td>
<td>300</td>
</tr>
<tr>
<td>SCons.Platform.hpux module</td>
<td>300</td>
</tr>
<tr>
<td>SCons.Platform.irix module</td>
<td>300</td>
</tr>
<tr>
<td>SCons.Platform.mingw module</td>
<td>301</td>
</tr>
<tr>
<td>SCons.Platform.os2 module</td>
<td>301</td>
</tr>
<tr>
<td>SCons.Platform.posix module</td>
<td>301</td>
</tr>
<tr>
<td>SCons.Platform.sunos module</td>
<td>301</td>
</tr>
<tr>
<td>SCons.Platform.virtualenv module</td>
<td>301</td>
</tr>
<tr>
<td>SCons.Platform.win32 module</td>
<td>302</td>
</tr>
<tr>
<td>Module contents</td>
<td>302</td>
</tr>
<tr>
<td>SCons.Scanner package</td>
<td>303</td>
</tr>
<tr>
<td>Submodules</td>
<td>303</td>
</tr>
<tr>
<td>SCons.Scanner.C module</td>
<td>303</td>
</tr>
<tr>
<td>SCons.Scanner.D module</td>
<td>307</td>
</tr>
<tr>
<td>SCons.Scanner.Dir module</td>
<td>308</td>
</tr>
<tr>
<td>SCons.Scanner.Fortran module</td>
<td>308</td>
</tr>
<tr>
<td>SCons.Scanner.IDL module</td>
<td>309</td>
</tr>
<tr>
<td>SCons.Scanner.LaTeX module</td>
<td>309</td>
</tr>
<tr>
<td>SCons.Scanner.Prog module</td>
<td>311</td>
</tr>
<tr>
<td>SCons.Scanner.RC module</td>
<td>311</td>
</tr>
<tr>
<td>SCons.Scanner.SWIG module</td>
<td>311</td>
</tr>
<tr>
<td>Module contents</td>
<td>311</td>
</tr>
<tr>
<td>SCons.Script package</td>
<td>314</td>
</tr>
<tr>
<td>Submodules</td>
<td>314</td>
</tr>
<tr>
<td>SCons.Script.Interactive module</td>
<td>314</td>
</tr>
<tr>
<td>SCons.Script.Main module</td>
<td>316</td>
</tr>
<tr>
<td>SCons.Script.SConsOptions module</td>
<td>323</td>
</tr>
<tr>
<td>SCons.Script.SConscript module</td>
<td>330</td>
</tr>
<tr>
<td>Module contents</td>
<td>337</td>
</tr>
<tr>
<td>SCons.Tool package</td>
<td>338</td>
</tr>
<tr>
<td>Module contents</td>
<td>338</td>
</tr>
<tr>
<td>SCons.Variables package</td>
<td>343</td>
</tr>
</tbody>
</table>
This is the internal API Documentation for SCons. The Documentation is generated using the Sphinx tool. The target audience is developers working on SCons itself, so it does not clearly delineate what is “Public API” - interfaces for use in your SCons configuration scripts which have a consistency guarantee, and what is internal, so always keep the SCons manual page around for helping with such determinations.

SCons package

Module contents

Subpackages

SCons.Node package

Submodules

SCons.Node.Alias module

scons.Node.Alias

Alias nodes.

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

class SCons.Node.Alias.Alias (name)

class Attrs
Bases: object

shared

BuildInfo
alias of AliasBuildInfo

Decider (function)

GetTag (key)
Return a user-defined tag.

NodeInfo
alias of AliasNodeInfo

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

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()

close()

del_binfo()
   Delete the build info from this node.

depends

depends_set

disambiguate(must_exist=None)

eval

eval_set(env, safe=0)

evaluator

evaluator_cleanup()
   Let the evaluator clean up any cached information.

exists()
   Does this node exist?

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()
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 __nonzero__ 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_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 __nonzero__ 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.

reexists()
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

bact
bactsig
bdepends
bdependsgs
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(**kwargs)
Bases: collections.UserDict

Alias(name, **kw)

__mutableMapping__marker = <object object>

__abc_impl = <_abc_data object>

clear() → None. Remove all items from D.
class SCons.Node.FS

scons.Node.FS

File system nodes.

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

This holds a "default_fs" variable that should be initialized with an FS that can be used by scripts or modules looking for the canonical default.
A generic class for file system entries. This class is for when we don’t know yet whether the entry being looked up is a file or a directory. Instances of this class can morph into either Dir or File objects by a later, more precise lookup.

Note: this class does not define __cmp__ and __hash__ for efficiency reasons. SCons does a lot of comparing of Node.FS.{Base,Entry,File,Dir} objects, so those operations must be as fast as possible, which means we want to use Python’s built-in object identity comparisons.

class Attrs
    Bases: object

    shared

BuildInfo
    alias of SCons.Node.BuildInfoBase

Decider(func)

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)

_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()
SCons Project API Documentation

```python
_globl(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.
```
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**

**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)
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.

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_csSig ()

get_contents ()
    Fetch the contents of the entry.

get_csSig ()

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_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 __nonzero__ 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_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 __nonzero__ 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.

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

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
alias of 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 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.

**_Dir__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.

**_Dir__resetDuplicate** *(node)*  

**_Rfindalldirs_key** *(pathlist)*  

**_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_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)*  
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

**_labyrinth**

**_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

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=\text{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

env_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 \text{\_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

getRepositories()
    Returns a list of repositories for this directory.

get_abspath()
    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()

g_get_env()

g_get_env_scanner(env, kw={})

g_get_executor(create=1)
Fetch the action executor for this node. Create one if there isn’t already one, and requested to do so.

g_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).

g_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.

g_get_internal_path()

g_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()

g_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()
  Return the latest timestamp from among our children

get_tpath()

getmtime()

getsize()

glob(pathname, ondisk=True, source=False, strings=False, exclude=None)
  Returns a list of Nodes (or strings) matching a specified pathname pattern.
  Pathname patterns follow UNIX shell semantics: * matches any-length strings of any characters, ? matches any character, and [ ] can enclose lists or ranges of characters. Matches do not span directory separators.
  The matches take into account Repositories, returning local Nodes if a corresponding entry exists in a Repository (either an in-memory Node or something on disk).
  By default, the glob() function matches entries that exist on-disk, in addition to in-memory Nodes. Setting the “ondisk” argument to False (or some other non-true value) causes the glob() function to only match in-memory Nodes. The default behavior is to return both the on-disk and in-memory Nodes.
  The “source” argument, when true, specifies that corresponding source Nodes must be returned if you’re globbing in a build directory (initialized with VariantDir()). The default behavior is to return Nodes local to the VariantDir().
  The “strings” argument, when true, returns the matches as strings, not Nodes. The strings are path names relative to this directory.
  The “exclude” argument, if not None, must be a pattern or a list of patterns following the same UNIX shell semantics. Elements matching a least one pattern of this list will be excluded from the result.
  The underlying algorithm is adapted from the glob.glob() function in the Python library (but heavily modified), and uses fnmatch() under the covers.

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 __nonzero__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra
calls and slowing things down immensely.

```python
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).

```python
ignore
ignore_set
```

```python
implicit
```

```python
implicit_set
```

```python
includes
```

```python
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.

```python
is_explicit
```

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

```python
is_sconscript()
```
Returns true if this node is an sconscript

```python
is_under(dir)
```

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

```python
isdir()
```

```python
isfile()
```

```python
islink()
```

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

```python
linked
```

```python
lstat()
```

```python
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()
```

```python
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 __nonzero__ 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 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.

```python
set_explicit(is_explicit)
```

```python
set_local()
```

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

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

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

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

```python
set_specific_source(source)
```

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

```python
set_state(state)
```

```python
side_effect
```

```python
side_effects
```

```python
sources
```

```python
sources_set
```

```python
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).
```

```python
srcdir
```

```python
srcdir_duplicate(name)
```

```python
srcdir_find_file(filename)
```

```python
srcdir_list()
```

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

```python
stat()
```

```python
state
```

```python
store_info
```

```python
str_for_display()
```

```python
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
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.FS.DirNodeInfo
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 (type, do, ignore)
Bases: object

set(list)

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)

_abspath

_add_child(collection, set, child)
Add '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()
__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**

**bininfo**

**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_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**

**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
eenv
    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.
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.

**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_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 __nonzero__ 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_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\(\text{dir}\)

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 \_nonzero\_ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.

must_be_same\(\text{klass}\)

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`

`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()

`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.

`rexists()`
  Does this node exist locally or in a repository?

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

`root`

`rstr()`
  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
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

_EntryProxy__get_abspath()

_EntryProxy__get_base_path()

Return the file’s directory and file name, with the suffix stripped.

_EntryProxy__get_dir()

_EntryProxy__get_file()

_EntryProxy__get_filebase()

_EntryProxy__get_posix_path()

Return the path with / as the path separator, regardless of platform.

_EntryProxy__get_rsrcdir()

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

_EntryProxy__get_rsrcnode()

_EntryProxy__get_srcdir()

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

_EntryProxy__get_srcnode()

_EntryProxy__get_suffix()

_EntryProxy__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>, '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>, 'win32': <function EntryProxy.__get_windows_path>, 'windows': <function EntryProxy.__get_windows_path>}

39
get ()
Retrieve the entire wrapped object

exception SCons.Node.FS.EntryProxy AttributeError (entry_proxy, attribute)
Bases: AttributeError
An AttributeError subclass for recording and displaying the name of the underlying Entry involved in an
AttributeError exception.

args

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=None, 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=None, 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=None, 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)
Globs
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-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=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=0)
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()`

`getmtime(path)`

`getsize(path)`

`isdir(path)`

`isfile(path)`

`islink(path)`

`link(src, dst)`

`listdir(path)`

`lstat(path)`

`makedirs(path)`

`mkdir(path)`

`open(path)`

`readlink(file)`

`rename(old, new)`

`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 `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 `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.

>_File__dmap_cache = {}_

>_File__dmap_sig_cache = {}_

>_Rfindalldirs_key (pathlist)_

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

Returns: dictionary of file->signature mappings

```python
_defn
_defn
_defn

_defn

_defn

_defn

_defn

_defn

_defn

_defn

_defn

_defn

_defn

_defn

_defn
```

Return a list of corresponding csigs from previous build in order of the node/files in children.

```python
_defn
```

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

Returns: List of csigs for provided list of children

```python
_defn
```

```python
_defn
```

```python
_defn
```

```python
_defn
```

```python
_defn
```

```python
_defn
```

```python
_defn
```

```python
_defn
```

```python
_defn
```

Turn a file system node into a File object.

```python
_defn
```

```python
_defn
```

```python
_defn
```

```python
_defn
```

```python
_defn
```

```python
_defn
```

```python
_defn
```

```python
_defn
```

```python
_defn
```

```python
_defn
```

```python
_defn
```

```python
_defn
```

```python
_defn
```

```python
_defn
```
**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_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
  :param target: :param prev_ni:
  Information about the node from the previous build

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

  Returns: Boolean - Indicates if node(File) has changed.

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', 'bdependsigns']

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

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()**
Compute and return the MD5 hash for this file.

**get_contents()**
Fetch the contents of the entry.

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

**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_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.

**getImplicit_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()**
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_size()

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()
    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()

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 __nonzero__ 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.

**ignore**

**ignore_set**

**implicit**

**implicit_set**

**includes**

**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.

**md5_chunksize = 64**
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 __nonzero__ 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* file are used to figure out which
linker to use for the resulting Program (gcc vs. g++)! That's why we check for the 'keep_targetinfo' attribute,
config Nodes and the Interactive mode just don't allow an early release of most variables.
In the same manner, we can't simply remove the self.attributes here. The smart linking relies on the shared flag,
and some parts of the java Tool use it to transport information about nodes...
@see: built() and Node.release_target_info()

released_target_info

remove()
Remove this file.
render_include_tree()
Return a text representation, suitable for displaying to the user, of the include tree for the sources of this node.

rentry()

repositories

reset_executor()
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

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

bact
bactsig
bdepends
bdependssigs
bimplicit
bimplicitsigs
bsources
bsourcessigs

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 SCon struct 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 SCon struct 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 nodes to search in. Can be represented as a list, a
tuple, or a callable that is called with no arguments and returns the list or tuple.
returns The node created from the found file.

class SCons.Node.FS.FileNodeInfo

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.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)
mkdir (path)
open (path)
readlink (file)
rename (old, new)
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 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 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.

_Dir__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.

_Dir__resetDuplicate(node)

_Rfindalldirs_key(pathlist)

_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

_abspath
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.

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.

Adds dependencies.
Adds dependencies to ignore.
Adds prerequisites
Adds sources.
Adds 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`

`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).

```python

clear_memoized_values()
```

```python
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
env_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

goingRepositories()
Returns a list of repositories for this directory.

going_abspath()
Get the absolute path of the file.

going_all_rdirs()

going_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.

going_build_env()
Fetch the appropriate Environment to build this node.

going_build_scanner_path(scanner)
Fetch the appropriate scanner path for this node.

going_builder(default_builder=None)
Return the set builder, or a specified default value

going_cachedir_csиг()

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

going_csиг()
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
cd node should return the hash of its contents.

going_dir()

going_env()

going_env_scanner(env, kw={})

going_executor(create=1)
Fetch the action executor for this node. Create one if there isn’t already one, and requested to do so.

going_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()
    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_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()
    Return the latest timestamp from among our children

get_tpath()

gmtime()

getsize()
**glob** *(pathname, ondisk=True, source=False, strings=False, exclude=None)*

Returns a list of Nodes (or strings) matching a specified pathname pattern.

Pathname patterns follow UNIX shell semantics: * matches any-length strings of any characters, ? matches any character, and [] can enclose lists or ranges of characters. Matches do not span directory separators.

The matches take into account Repositories, returning local Nodes if a corresponding entry exists in a Repository (either an in-memory Node or something on disk).

By default, the glob() function matches entries that exist on-disk, in addition to in-memory Nodes. Setting the “ondisk” argument to False (or some other non-true value) causes the glob() function to only match in-memory Nodes. The default behavior is to return both the on-disk and in-memory Nodes.

The “source” argument, when true, specifies that corresponding source Nodes must be returned if you’re globbing in a build directory (initialized with VariantDir()). The default behavior is to return Nodes local to the VariantDir().

The “strings” argument, when true, returns the matches as strings, not Nodes. The strings are path names relative to this directory.

The “exclude” argument, if not None, must be a pattern or a list of patterns following the same UNIX shell semantics. Elements matching a least one pattern of this list will be excluded from the result.

The underlying algorithm is adapted from the glob.glob() function in the Python library (but heavily modified), and uses fnmatch() under the covers.

**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 __nonzero__ 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_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.

**isdire**
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 __nonzero__ 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

cache

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.

scansign()
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).

csrcdir

csrcdir_duplicate(name)
srcdir_find_file(filename)

cendir_list()

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

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
  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()
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(list)

SCons.Node.FS.set_duplicate(duplicate)

---

SCons.Node.Python module

scons.Node.Python
Python nodes.

class SCons.Node.Python.Value(value, built_value=None, name=None)
A class for Python variables, typically passed on the command line or generated by a script, but not from a file or some other source.

class Attrs
Bases: object

shared

BuildInfo
alias of ValueBuildInfo

Decider(function)

GetTag(key)
Return a user-defined tag.

NodeInfo
alias of ValueNodeInfo

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()

changed_since_last_build

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 contents for signature calculations. :return: bytes

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()**
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 __nonzero__ 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-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 __nonzero__ 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

**bact**

**bactsig**

**bdepends**

**bdependssigs**

**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.


    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.

Module contents

SCons.Node
    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.

    bact

    bactsig

    bdepends

    bdependsigs

    bimplicit

    bimplicitcmds
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 BuildInfoBase

Decider (function)

GetTag(key)
Return a user-defined tag.

NodeInfo
alias of 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

_getScanner(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

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()
    Fetch the contents of the entry.

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 __nonzero__ 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_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 __nonzero__ 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.

    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

_UserList__cast (other)

_abcs_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, item)
    S.insert(index, value) – insert value before index

pop ([, index]) → item -- remove and return item at index (default last).
    Raise IndexError if list is empty or index is out of range.

remove (item)
    S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

reverse ()
    S.reverse() – reverse IN PLACE

sort (*args, **kwds)

class SCons.Node.Walker (node, kids_func=<function get_children>, cycle_func=<function ignore_cycle>, eval_func=<function do_nothing>)
Bases: object
An iterator for walking a Node tree.
This is depth-first, children are visited before the parent. The Walker object can be initialized with any node, and returns the next node on the descent with each get_next() call. get the children of a node instead of calling 'children'. 'cycle_func' is an optional function that will be called when a cycle is detected.
This class does not get caught in node cycles caused, for example, by C header file include loops.

get_next ()
    Return the next node for this walk of the tree.
    This function is intentionally iterative, not recursive, to sidestep any issues of stack size limitations.

is_done ()
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.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

SCons.Platform.aix

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.

\[ \text{generate}(\text{env}) \]

SCons.Platform.aix.

\[ \text{get_xlc}(\text{env}, \text{xlc}=\text{None}, \text{packages}=[]) \]

### SCons.Platform.cygwin module

SCons.Platform.cygwin

Platform-specific initialization for Cygwin systems.

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

SCons.Platform.cygwin.

\[ \text{generate}(\text{env}) \]

### SCons.Platform.darwin module

SCons.Platform.darwin

Platform-specific initialization for Mac OS X systems.

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

SCons.Platform.darwin.

\[ \text{generate}(\text{env}) \]

### SCons.Platform.hpux module

SCons.Platform.hpux

Platform-specific initialization for HP-UX systems.

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

SCons.Platform.hpux.

\[ \text{generate}(\text{env}) \]

### SCons.Platform.irix module

SCons.Platform.irix

Platform-specific initialization for SGI IRIX systems.

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

SCons.Platform.irix.

\[ \text{generate}(\text{env}) \]

### SCons.Platform.mingw module

SCons.Platform.mingw

Platform-specific initialization for the MinGW system.
**SCons.Platform.os2 module**

SCons.Platform.os2

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**

SCons.Platform.posix

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**

SCons.Platform.sunos

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**

SCons.Platform.virtualenv

Support for 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**

SCons.Platform.win32

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.

```python
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()
    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

SCons platform selection.

This 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.

```python
class SCons.Platform.PlatformSpec(name, generate):
    Bases: object

class SCons.Platform.TempFileMunge(cmd, cmdstr=None):
    Bases: object
    Convert long command lines to use a temporary file.
```
You can set an Environment variable (usually TEMPFILE) to this, then call it with a string argument, and it will perform temporary file substitution on it. This is used to circumvent limitations on the length of command lines.

Example:

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

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

```python
env["TEMPFILEPREFIX"] = '-0'  # diab compiler
env["TEMPFILEPREFIX"] = '-via' # arm tool chain
env["TEMPFILEPREFIX"] = ''    # (the empty string) PC Lint
```

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

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

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

```python
__print_cmd_str(target, source, env, cmdstr)
```

SCons.Platform.

```
platform_default()
```

Return the platform string for our execution environment.

The returned value should map to one of the SCons/Platform/*.py files. Since scons is architecture independent, though, we don’t care about the machine architecture.

```python
platform_module(name='darwin')
```

Return the imported module for the platform.

This looks for a module name that matches the specified argument. If the name is unspecified, we fetch the appropriate default for our execution environment.

---

**SCons.Scanner package**

**Submodules**

**SCons.Scanner.C module**

SCons.Scanner.C

This module implements the 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,**kw)
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.

```python
__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, **kw)
    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.

    _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.

```python
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)
```

```python
def SCons.Scanner.D.dictify_CPPDEFINES(env)
```

**SCons.Scanner.D module**

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

```python
class SCons.Scanner.D.D
    Bases: SCons.Scanner.Classic

    _recurse_all_nodes(nodes)

    _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)

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(**kw)
Return a prototype Scanner instance for "scanning" directory Nodes for their in-memory entries

SCons.Scanner.Dir.DirScanner(**kw)
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

SCons.Scanner.Fortran
This module implements the dependency scanner for Fortran code.

class SCons.Scanner.Fortran.F90Scanner(name, suffixes, path_variable, use_regex, incl_regex, def_regex, *args, **kw)
   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.

   _recurse_all_nodes(nodes)

   _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, env, path=())

   select(node)
**SCons.Scanner.Fortran module**

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**

SCons.Scanner.IDL

This module implements the dependency scanner for IDL (Interface Definition Language) files.

SCons.Scanner.IDL.IDLScan()

Return a prototype Scanner instance for scanning IDL source files

**SCons.Scanner.LaTeX module**

SCons.Scanner.LaTeX

This module implements the 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, **kw)

  Bases: SCons.Scanner.Base

  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\{class|style\}{}

  ```latex
  _latex_names(include_type, filename)
  _recurse_all_nodes(nodes)
  _recurse_no_nodes(nodes)
  add_scanner(skey, scanner)
  add_skey(skey)
    Add a skey to the list of skeys
  canonical_text(text)
Standardize an input TeX-file contents.

Currently:

- removes comments, unwrapping comment-wrapped lines.

```python
env_variables = ['TEXINPUTS', 'BIBINPUTS', 'BSTINPUTS', 'INDEXSTYLE']

find_include(include, source_dir, path)

get_skeys(env=None)

keyword_paths = {'addbibresource': 'BIBINPUTS', 'addglobalbib': 'BIBINPUTS', 'addsectionbib': 'BIBINPUTS',
                 'bibliography': 'BIBINPUTS', 'bibliographystyle': 'BSTINPUTS', 'include': 'TEXINPUTS', 'includegraphics':
                 'TEXINPUTS', 'input': 'TEXINPUTS', 'listinputlisting': '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

SCons.Scanner.Prog.ProgramScanner(**kw)
    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=())
    This scanner 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

SCons.Scanner.RC
    This module implements the 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

**SCons.Scanner.SWIG**
This module implements the dependency scanner for SWIG code.

**SCons.Scanner.SWIG.SWIGScanner()**

### Module contents

**SCons.Scanner**
The Scanner package for the SCons software construction utility.

```python
class SCons.Scanner.Base (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
The base class for dependency scanners. This implements straightforward, single-pass scanning of a single file.

  _recurse_all_nodes (nodes)

  _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.Classic (name, suffixes, path_variable, regex, *args, **kw)
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_key() methods), the regular expression passed to the constructor must return the name of the include file in group 0.

  _recurse_all_nodes (nodes)

  _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)
```
SCons Project API Documentation

```python
scan(node, path=())

select(node)

sort_key(include)

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

_recurse_all_nodes(nodes)

_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)

sort_key(include)

class SCons.Scanner.Current(*args, **kw)
Bases: SCons.Scanner.Base
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).

_recurse_all_nodes(nodes)

_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.FindPathDirs(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.
SCons.Scanner.Scanner(function, *args, **kw)
```

97
Public interface factory function for creating different types of Scanners based on the different types of “functions” that may be supplied. TODO: Deprecate this some day. We’ve moved the functionality inside the Base 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.Selector(dict, *args, **kw):
    Bases: SCons.Scanner.Base
    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 Base 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.)
    
    _recurse_all_nodes(nodes)
    _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)
```

```python
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

```python
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)

completenames(text, *ignore)

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'
SCons Project API Documentation

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 = ‘=’

undoc_header = ‘Undocumented commands:’
use_rawinput = 1
SCons.Script.Interactive.interact (fs, parser, options, targets, target_top)

SCons.Script.Main module
SCons.Script
This file implements 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.
   _abc_impl = <_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.

```python
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.

```python
exc_info()
```
Returns info about a recorded exception.

```python
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().

```python
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.

```python
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.

```python
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().

```python
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().

```python
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().

```python
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(method, node, description='node')**

A 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 ()

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(method, 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.
    __abc_impl = <_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:

```python
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(method, 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._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=None)
 Load all of the predefined site_scons dir. Order is significant; we load them in order from most generic
(machine-wide) to most specific (topdir). The verbose argument is only for testing.

SCons.Script.Main._load_site_scons_dir(topdir, site_dir_name=None)
 Load the site directory under topdir.
 If a site dir name is supplied use it, else use default “site_scons” Prepend site dir to sys.path. If a “site_tools” subdir
exists, prepend to toolpath. Import “site_init.py” from site dir if it exists.

SCons.Script.Main._main(parser)

SCons.Script.Main._scons_internal_error()
 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.fetch_win32_parallel_msg()
SCons.Script.Main.find_deepest_user_frame(tb)
Find the deepest stack frame that is not part of SCons.
Input is a “pre-processed” stack trace in the form returned by traceback.extract_tb() or traceback.extract_stack()

SCons.Script.Main.main()
SCons.Script.Main.path_string(label, module)
SCons.Script.Main.python_version_deprecated(version=sys.version_info(major=3, minor=7, micro=8, releaselevel='final', serial=0))
SCons.Script.Main.python_version_string()
SCons.Script.Main.python_version_unsupported(version=sys.version_info(major=3, minor=7, micro=8, 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 an options parser object initialized with the standard SCons options.

```
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.

dedent ()

expand_default (option)

format_description (description)

format_epilog (epilog)

format_heading (heading)
This translates any heading of “options” or “Options” into “SCons Options.” Unfortunately, we have to do this here, because those titles are hard-coded in the optparse calls.

format_option (option)
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.

`format_option_strings(option)`
Return a comma-separated list of option strings & metavariables.

`format_usage(usage)`

`indent()`

`set_long_opt_delimiter(delim)`

`set_parser(parser)`

`set_short_opt_delimiter(delim)`

`store_option_strings(parser)`

```python
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_kargs', 'help', 'metavar']

    CHECK_METHODS = [Option._check_action, Option._check_type, Option._check_choice, Option._check_dest, Option._check_const, 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 ()

_set_attrs(attrs)

_set_opt_strings(opts)

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, ...)

   add_options (option_list)

destroy ()
   see OptionParser.destroy().

format_description (formatter)

format_help (formatter)
   Format an option group’s help text, outdenting the title 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(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.

_process_short_opts(rargs, values)

_share_option_mappings(parser)

def add_local_option(*args, **kw)
    Adds a local option to the parser.
    This is initiated by a SetOption() 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.

def add_option(Option)
    add_option(opt_str, ..., kwarg=val, ...)

def add_option_group(*args, **kwargs)

def add_options(option_list)

def check_values(values : Values, args : [string])
    -> (values : Values, args : [string])
Check that the supplied option values and leftover arguments are valid. Returns the option values and leftover arguments (possibly adjusted, possibly completely new – whatever you like). Default implementation just returns the passed-in values; subclasses may override as desired.

**destroy()**
Declare that you are done with this OptionParser. This cleans up reference cycles so the OptionParser (and all objects referenced by it) can be garbage-collected promptly. After calling destroy(), the OptionParser is unusable.

**disable_interspersed_args()**
Set parsing to stop on the first non-option. Use this if you have a command processor which runs another command that has options of its own and you want to make sure these options don’t get confused.

**enable_interspersed_args()**
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior. See also disable_interspersed_args() and the class documentation description of the attribute allow_interspersed_args.

**error(msg : string)**
Print a usage message incorporating ‘msg’ to stderr and exit. If you override this in a subclass, it should not return – it should either exit or raise an exception.

**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(opt_str)**

**get_option_group(opt_str)**

**get_prog_name()**

**get_usage()**

**get_version()**

**has_option(opt_str)**

**parse_args(args=None, values=None)**

Parse 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.

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-opt 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 or not 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.

```
_update(dict, mode)
_update_careful(dict)
  Update the option values from an arbitrary dictionary, but only use keys from dict that already have a
corresponding attribute in self. Any keys in dict without a corresponding attribute are silently ignored.
_update_loose(dict)
  Update the option values from an arbitrary dictionary, using all keys from the dictionary regardless of whether
they have a corresponding attribute in self or not.
ensure_value(attr, value)
read_file(filename, mode='careful')
read_module(modname, mode='careful')
set_option(name, value)
  Sets an option from an SConscript file.
SCons.Script.SConscriptOptions.diskcheck_convert(value)
```

---

**SCons.Script.SConscript module**

SCons.Script.SConscript

This module defines the Python API provided to SConscript and SConstruct 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)

---

114
AddMethod(function, name=None)
Add 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 existing construction variables in an Environment.
AppendENVPath(name, newpath, envname='ENV', sep=':', delete_existing=0)
Append path elements to the path 'name' in the 'ENV' 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 0, a newpath which is already in the path will not be moved to the end (it will be left where it is).
AppendUnique(delete_existing=0, **kw)
Append values to existing construction variables in an Environment, if they're not already there. If delete_existing is 1, removes existing values first, so values move to end.
Builder(**kw)
CacheDir(path)
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's depend on 'dependency'.
Detect(progs)
Return the first available program in progs.
Parameters: progs (str or list) – one or more command names to check for
Returns str: first name from progs that can be found.
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** *(optional)* – specify the format to serialize to. "pretty" generates a pretty-printed string, "json" a JSON-formatted string. (Default value = None, equivalent to "pretty")

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

Exit abnormally if the Python version is not late enough.

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

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

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

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

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='.')
```

returns a list of all source files.

```python
Flatten(sequence)
```

```python
GetBuildPath(files)
```

```python
GetLaunchDir()
```

```python
GetOption(name)
```

116
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=1, dict=None)
  Merge the dict in args into the construction variables of this env, or the passed-in dict. If args is not a dict, it is converted into a dict using ParseFlags. If unique is not set, the flags are appended rather than merged.
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=1)
  Use the specified function to parse the output of the command in order to modify the current environment. The 'command' can be a string or a list of strings representing a command and its arguments. 'Function' is an optional argument that takes the environment, the output of the command, and the unique flag. If no function is specified, MergeFlags, which treats the output as the result of a typical 'X-config' command (i.e. gtk-config), will merge the output into the appropriate variables.
ParseDepends(filename, must_exist=None, only_one=0)
  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)
  Parse the set of flags and return a dict with the flags placed in the appropriate entry. The flags are treated as a typical set of command-line flags for a GNU-like toolchain and used to populate the entries in the dict immediately below. If one of the flag 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 existing construction variables in an Environment.
PrependENVPath(name, newpath, envname='ENV', sep=': ', delete_existing=1)
  Prepend path elements to the path ‘name’ in the ‘ENV’ 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 0, a new path which is already in the path will not be moved to the front (it will be left where it is).

**PrependUnique**(`delete_existing=0,**kw`)  
Prepend values to existing construction variables in an Environment, if they’re not already there. If `delete_existing` is 1, 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.
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, **kw)`

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

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

`whereis(prog, path=None, pathext=None, reject=[])`  
Find prog in the path.

`_canonicalize(path)`

`_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)`

`_copy2_from_cache(src, dst)`

`_copy_from_cache(src, dst)`

`_exceeds_version(major, minor, v_major, v_minor)`  
Return 1 if ’major’ and ’minor’ are greater than the version in ’v_major’ and ’v_minor’, and 0 otherwise.

`_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.

`_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(dict)`
Update an environment’s values directly, bypassing the normal checks that occur when users try to set items.

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

```python
backtick(command)
```

```python
get(key, default=None)
    Emulates the get() method of dictionaries.
```

```python
get_CacheDir()
```

```python
get_builder(name)
    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(skey)
    Find the appropriate scanner given a key (usually a file suffix).
```

```python
get_src_sig_type()
```

```python
get_tgt_sig_type()
```

```python
gvars()
```

```python
has_key(key)
    Emulates the has_key() method of dictionaries.
```

```python
items()
    Emulates the items() method of dictionaries.
```

```python
keys()
    Emulates the keys() method of dictionaries.
```

```python
gvars()
```

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

```python
subst(string, raw=0, target=None, source=None, conv=None, executor=None)
    Recursively interpolates construction variables from the Environment into the specified string, returning the expanded result. Construction variables are specified by a $ prefix in the string and begin with an initial underscore or alphabetic character followed by any number of underscores or alphanumeric characters. The construction variable names may be surrounded by curly braces to separate the name from trailing characters.
```

```python
subst_kw(kw, raw=0, target=None, source=None)
```

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

```python
subst_path(path, target=None, source=None)
    Substitute a path list, 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)
    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.

**exception** SCons.Script.SConscript.SConscriptReturn

Bases: Exception

args

with_traceback() -> Exception

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

SCons.Script.SConscript.SConscript_exception(file=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>)

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. On first warning, print a deprecation message.

Parameters:
- f (str) – path of missing configuration file
- must_exist (bool) – raise exception if file does not exist

Raises: UserError if ‘must_exist’ is True or if global – SCons.Script.no_missing_sconscript is True.

---

**Module contents**

SCons.Script

This file implements 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

__UserList__cast (other)

__abc_impl = __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._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\( (flag=1) \)
   Set behavior on missing file in SConscript() call. Returns previous value

\textbf{SCons.Tool package}

\textbf{Module contents}

SCons.Tool

SCons tool selection.

This 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 definition.

\textbf{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 Project API Documentation

SCons.Tool.CreateJavaClassDirBuilder (env)
SCons.Tool.CreateJavaClassFileBuilder (env)
SCons.Tool.CreateJavaFileBuilder (env)
SCons.Tool.CreateJavaHBuilder (env)
SCons.Tool.CreateLibSymlinks (env, symlinks)
  Physically creates symlinks. The symlinks argument must be a list in form [(link, linktarget), ...], where link and
  linktarget are SCons nodes.
SCons.Tool.EmitLibSymlinks (env, symlinks, libnode, **kw)
  Used by emitters to handle (shared/versioned) library symlinks
SCons.Tool.FindAllTools (tools, env)
SCons.Tool.FindTool (tools, env)
SCons.Tool.Initializers (env)
SCons.Tool.LibSymlinksActionFunction (target, source, env)
SCons.Tool.LibSymlinksStrFun (target, source, env, *args)
SCons.Tool.StringizeLibSymlinks (symlinks)
  Converts list with pairs of nodes to list with pairs of node paths (strings). Used mainly for debugging.

class SCons.Tool.Tool (name, toolpath=None, **kw)
  Bases: object

  _load_dotted_module_py2 (short_name, full_name, searchpaths=None)

  _tool_module ()

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.

  get_builder (env)
    Returns the appropriate real Builder for this method name after having the associated ToolInitializer object apply
    the appropriate Tool module.

class SCons.Tool._ImpLibInfoSupport
  Bases: object

  get_lib_noversionsymlinks (env, *args, **kw)
```python
get_lib_prefix(env, *args, **kw)
get_lib_suffix(env, *args, **kw)
get_lib_version(env, *args, **kw)

property libtype

class SCons.Tool._LdModInfoSupport
    Bases: object
    
    get_lib_noversionsymlinks(env, *args, **kw)
    get_lib_prefix(env, *args, **kw)
    get_lib_suffix(env, *args, **kw)
    get_lib_version(env, *args, **kw)

property libtype

class SCons.Tool._LibInfoGeneratorBase (libtype, infoname)
    Bases: object
    Generator base class for library-related info such as suffixes for versioned libraries, symlink maps, sonames etc. It handles commonalities of SharedLibrary and LoadableModule
    
    _support_classes = {'ImpLib': <class 'SCons.Tool._ImpLibInfoSupport'>, 'LdMod': <class 'SCons.Tool._LdModInfoSupport'>, 'ShLib': <class 'SCons.Tool._ShLibInfoSupport'>}
    
    generate_versioned_lib_info (env, args, result=None, **kw)
    get_lib_noversionsymlinks(env, *args, **kw)
    get_lib_prefix(env, *args, **kw)
    get_lib_suffix(env, *args, **kw)
    get_lib_version(env, *args, **kw)
    get_versioned_lib_info_generator(**kw)

property libtype

class SCons.Tool._LibNameGenerator (libtype)
    Bases: SCons.Tool._LibInfoGeneratorBase
    Generates "unmangled" library name from a library file node.
    Generally, it's thought to revert modifications done by prefix/suffix generators (_LibPrefixGenerator/_LibSuffixGenerator) used by a library builder. For example, on gnu-link the suffix generator used by SharedLibrary builder appends $SHLIBVERSION to $SHLIBSUFFIX producing node name which ends with "$SHLIBSUFFIX.$SHLIBVERSION". Correspondingly, the implementation of _LibNameGenerator replaces "$SHLIBSUFFIX.$SHLIBVERSION" with "$SHLIBSUFFIX" in the node's basename. So that, if $SHLIBSUFFIX is ".so", $SHLIBVERSION is "0.1.2" and the node path is "foo/bar/libfoo.so.0.1.2", the _LibNameGenerator shall return "libfoo.so". Other link tools may implement it's own way of library name unmangling.
    
    _support_classes = {'ImpLib': <class 'SCons.Tool._ImpLibInfoSupport'>, 'LdMod': <class 'SCons.Tool._LdModInfoSupport'>, 'ShLib': <class 'SCons.Tool._ShLibInfoSupport'>}
    
    generate_versioned_lib_info (env, args, result=None, **kw)
    get_lib_noversionsymlinks(env, *args, **kw)
```

SCons Project API Documentation
get_lib_prefix(env, *args, **kw)
get_lib_suffix(env, *args, **kw)
get_lib_version(env, *args, **kw)
get_versioned_lib_info_generator(**kw)

property libtype
class SCons.Tool._LibPrefixGenerator (libtype)
Bases: SCons.Tool._LibInfoGeneratorBase
Library prefix generator, used as target_prefix in SharedLibrary and LoadableModule builders

_support_classes = {'ImpLib': <class 'SCons.Tool._ImpLibInfoSupport'>, 'LdMod': <class 'SCons.Tool._LdModInfoSupport'>, 'ShLib': <class 'SCons.Tool._ShLibInfoSupport'>}

generate_versioned_lib_info(env, args, result=None, **kw)
get_lib_noversionsymlinks(env, *args, **kw)
get_lib_prefix(env, *args, **kw)
get_lib_suffix(env, *args, **kw)
get_lib_version(env, *args, **kw)
get_versioned_lib_info_generator(**kw)

property libtype
class SCons.Tool._LibSonameGenerator (libtype)
Bases: SCons.Tool._LibInfoGeneratorBase
Library soname generator. Returns library soname (e.g. libfoo.so.0) for a given node (e.g. /foo/bar/libfoo.so.0.1.2)

_support_classes = {'ImpLib': <class 'SCons.Tool._ImpLibInfoSupport'>, 'LdMod': <class 'SCons.Tool._LdModInfoSupport'>, 'ShLib': <class 'SCons.Tool._ShLibInfoSupport'>}

generate_versioned_lib_info(env, args, result=None, **kw)
get_lib_noversionsymlinks(env, *args, **kw)
get_lib_prefix(env, *args, **kw)
get_lib_suffix(env, *args, **kw)
get_lib_version(env, *args, **kw)
get_versioned_lib_info_generator(**kw)

property libtype
class SCons.Tool._LibSuffixGenerator (libtype)
Bases: SCons.Tool._LibInfoGeneratorBase
Library suffix generator, used as target_suffix in SharedLibrary and LoadableModule builders

_support_classes = {'ImpLib': <class 'SCons.Tool._ImpLibInfoSupport'>, 'LdMod': <class 'SCons.Tool._LdModInfoSupport'>, 'ShLib': <class 'SCons.Tool._ShLibInfoSupport'>}

generate_versioned_lib_info(env, args, result=None, **kw)
get_lib_noversionsymlinks (env, *args, **kw)

get_lib_prefix (env, *args, **kw)

get_lib_suffix (env, *args, **kw)

get_lib_version (env, *args, **kw)

get_versioned_lib_info_generator (**kw)

property libtype

class SCons.Tool._LibSymlinkGenerator (libtype)
Bases: SCons.Tool._LibInfoGeneratorBase
Library symlink map generator. It generates a list of symlinks that should be created by SharedLibrary or LoadableModule builders

_support_classes = {'ImpLib': <class 'SCons.Tool._ImpLibInfoSupport'>, 'LdMod': <class 'SCons.Tool._LdModInfoSupport'>, 'ShLib': <class 'SCons.Tool._ShLibInfoSupport'>}

generate_versioned_lib_info (env, args, result=None, **kw)

get_lib_noversionsymlinks (env, *args, **kw)

get_lib_prefix (env, *args, **kw)

get_lib_suffix (env, *args, **kw)

get_lib_version (env, *args, **kw)

get_versioned_lib_info_generator (**kw)

property libtype

class SCons.Tool._ShLibInfoSupport
Bases: object

get_lib_noversionsymlinks (env, *args, **kw)

get_lib_prefix (env, *args, **kw)

get_lib_suffix (env, *args, **kw)

get_lib_version (env, *args, **kw)

property libtype

SCons.Tool._call_env_subst (env, string, *args, **kw)

SCons.Tool._call_linker_cb (env, callback, args, result=None)
Returns the result of env['LINKCALLBACKS'][callback](args) if env['LINKCALLBACKS'] is a dictionary and env['LINKCALLBACKS'][callback] is callable. If these conditions are not met, return the value provided as the *result argument. This function is mainly used for generating library info such as versioned suffixes, symlink maps, sones etc. by delegating the core job to callbacks configured by current linker tool

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)
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.

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)
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.

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)
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.

SCons.Tool.tool_list(platform, env)

SCons.Variables package

Submodules

SCons.Variables.BoolVariable module

SCons.Variables.BoolVariable
This file defines the option type for SCons implementing true/false values.
Usage example:
```python
opts = Variables()
opts.Add(BoolVariable('embedded', 'build for an embedded system', 0))
...
if env['embedded'] == 1:
...
```

The input parameters describe a boolean option, thus they are returned with the correct converter and validator appended. The ‘help’ text will by appended by ’(yes|no) to show the valid valued. The result is usable for input to opts.Add().

SCons.Variables.EnumVariable module

SCons.Variables.EnumVariable
This file defines the option type for SCons allowing only specified input-values.
Usage example:
```python
opts = Variables()
opts.Add(EnumVariable('debug', 'debug output and symbols', 'no',
```
allowed_values=('yes', 'no', 'full'),
map={}, ignorecase=2))
...
if env['debug'] == 'full':
...

SCons.Variables.EnumVariable.

**SCons.Variables.ListVariable**

This file defines the option type for SCons implementing ‘lists’. A ‘list’ option may either be ‘all’, ‘none’ or a list of names separated by comma. After the option has been processed, the option value holds either the named list elements, all list elements or no list elements at all.

Usage example:

```python
list_of_libs = Split('x11 gl qt ical')
opts = Variables()
opts.Add(ListVariable('shared',
                      'libraries to build as shared libraries',
                      'all',
                      elems = list_of_libs))
...
for lib in list_of_libs:
    if lib in env['shared']:
        env.SharedObject(...)
    else:
        env.Object(...)
```

SCons.Variables.ListVariable.

**SCons.Variables.PackageVariable**

This file defines the option type for SCons implementing ‘package activation’. To be used whenever a ‘package’ may be enabled/disabled and the package path may be specified.

Usage example:

**Examples:**
x11=no (disables X11 support) x11=yes (will search for the package installation dir) x11=/usr/local/X11 (will check this path for existence)

To replace autoconf’s –with-xxx=yyy

```python
opts = Variables()
 opts.Add(PackageVariable('x11',
    'use X11 installed here (yes = search some places',
    'yes'))
...
 if env['x11'] == True:
    dir = ... search X11 in some standard places ...
    env['x11'] = dir
if env['x11']:
    ... build with x11 ...
```

SCons.Variables.PackageVariable

The input parameters describe a ‘package list’ option, thus they are returned with the correct converter and validator appended. The result is usable for input to opts.Add() .

A ‘package list’ option may either be ‘all’, ‘none’ or a list of package names (separated by space).

SCons.Variables.PathVariable

This file defines an option type for SCons implementing path settings.

To be used whenever a user-specified path override should be allowed.

**Arguments to PathVariable are:**

- option-name = name of this option on the command line (e.g. “prefix”)
- option-help = help string for option
- option-dflt = default value for this option validator = [optional] validator for option value. Predefined validators 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). The key is the name of the option, the val is the path specified for the option, and the env is the env to which the Options have been added.

**Usage example:**

**Examples:**

prefix=/usr/local

```python
opts = Variables()

opts = Variables()
 opts.Add(PathVariable('qtdir',
    'where the root of Qt is installed',
    qtdir, PathIsDir))
opts.Add(PathVariable('qt_includes',
    'where the Qt includes are installed',
    '$qtdir/includes', PathIsDirCreate))
opts.Add(PathVariable('qt_libraries',
    'where the Qt library is installed',
    '$qtdir/lib'))
```

**Module contents**

SCons.Variables

This file defines the Variables class that is used to add 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)

Add (key='', default=None, validator=None, converter=None, **kw)

Add an option.

Parameters:

- key – the name of the variable, or a list or tuple of arguments
- help – optional help text for the options (Default value = “”)
- default – optional default value for option (Default value = None)
- validator – optional function called to validate the option’s value (Default value = None)
- converter – optional function to be called to convert the option’s value before putting it in the environment. (Default value = None)
- **kw – keyword args, unused.

AddVariables (*optlist)

Add a list of options.
Each list element is a tuple/list of arguments to be passed on to the underlying method for adding options.

Example:

```python
opt.AddVariables(
    ('debug', '', 0),
    ('CC', 'The C compiler'),
    ('VALIDATE', 'An option for testing validation', 'notset', validator, None),
)
```

FormatVariableHelpText (env, key, help, default, actual, aliases=[])}

GenerateHelpText (env, sort=None)

Generate the help text for the options.

env - an environment that is used to get the current values
of the options.

cmp - Either a function as follows: The specific sort function should take two arguments and return -1, 0 or 1 or a boolean to indicate if it should be sorted.

Save (filename, env)

Saves all the options in the given file. This file can then be used to load the options next run. This can be used to create an option cache file.
filename - Name of the file to save into env - the environment get the option values from

UnknownVariables ()

Returns any options in the specified arguments lists that were not known, declared options in this object.

Update (env, args=None)

Update an environment with the option variables.
env - the environment to update.
SCons Project API Documentation

```python
_do_add(key, help='', default=None, validator=None, converter=None)
```

```python
format = 'in%s: %s default: %s in actual: %s'
format_ = 'in%s: %s default: %s in actual: %s aliases: %s'
```

```python
instance = None
```

```python
keys()  
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 that we’d like
to use in SCons but which only show up in later versions of Python than the early, old version(s) we still support.

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.

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
    Bases: type
    Metaclass for PyPy compatibility.
    PyPy does not work well with __slots__ and __class__ assignment.
```

```python
mro ()
    Return a type’s method resolution order.
```

```python
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.Action

This encapsulates 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 OO 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 MD5 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.

gen_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 ascii/bytes and not unicode (or py3 strings)

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 ActionFactroy 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

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)

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)

subst(s, target, source, env)

subst_args(target, source, env)

subst_kw(target, source, env)

class SCons.Action.ActionFactory (actfunc, strfunc, convert=<function ActionFactory.<lambda>>)
Bases: object
A factory class that will wrap up an arbitrary function as an SCons-executable Action object.
The real heavy lifting here is done by the ActionCaller class. We just collect the (positional and keyword) arguments that we’re called with and give them to the ActionCaller object we create, so it can hang onto them until it needs them.

class SCons.Action.CommandAction (cmd, **kw)
Bases: 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.

\begin{verbatim}
get_targets(env, executor)
    Returns the type of targets ($TARGETS, $CHANGED_TARGETS) used by this action.
\end{verbatim}

\begin{verbatim}
get_varlist(target, source, env, executor=None)
no_batch_key(env, target, source)
\end{verbatim}

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

\begin{verbatim}
print_cmd_line(s, target, source, env)
\end{verbatim}

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.

\begin{verbatim}
process(target, source, env, executor=None)
strfunction(target, source, env, executor=None)
\end{verbatim}

\begin{verbatim}
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. 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.
\end{verbatim}

\begin{verbatim}
get_varlist(target, source, env, executor=None)
no_batch_key(env, target, source)
presub_lines(env)
\end{verbatim}

\begin{verbatim}
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_contents(target, source, env)
\end{verbatim}
get_implicit_deps (target, source, env)

get_presig (target, source, env)
   Return the signature contents of this callable action.

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.

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

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)

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)

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

class SCons.Action.ListAction (actionlist)
   Bases: SCons.Action.ActionBase
   Class for lists of other actions.

batch_key (env, target, source)

genstring (target, source, env)

get_contents (target, source, env)

get_implicit_deps (target, source, env)

get_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)

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)

get_contents (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)

**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:

- [https://docs.python.org/2/library/inspect.html](https://docs.python.org/2/library/inspect.html)

For info on what each _co_var 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 Command Action 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](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 consistent content for a action class or an instance thereof

Parameters:
- obj Should be either an action class or an instance thereof

Returns:
- bytearray or bytes representing the obj suitable for generating a signature from.

SCons.Action._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)
A fiddlin’ little function that has an ‘import SCons.Environment’ which can’t 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.

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.

```python
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.
```

```python
adjust_suffix(suff)
```

```python
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).
```

```python
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 ‘BUILDERS’ variable of the construction environment and cache the result.
```

```python
get_src_suffix(env)
Get the first src_suffix in the list of src_suffixes.
```

```python
set_src_suffix(src_suffix)
```

```python
get_suffix(env, sources=[])
set_suffix(suffix)
```

```python
splitext(path, env=None)
```

```python
src_builder_sources(env, source, overwarn={})
```

```python
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.)
```

```python
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.
```

```python
class SCons.Builder.CallableSelector
Bases: SCons.Util.Selector
A callable dictionary that will, in turn, call the value it finds if it can.
```

```python
clear() → None. Remove all items from od.
copy() → a shallow copy of od
fromkeys() → Create a new ordered dictionary with keys from iterable and values set to value.
get() → 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()  
Move an existing element to the end (or beginning if last is false).  
Raise KeyError if the element does not exist.

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()  
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()  
Insert key with a value of default if key is not in the dictionary.  
Return the value for key if key is in the dictionary, else default.

update([E], **F) → None. Update D from dict/iterable E and F.  
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]  
If E is present and lacks a .keys()  
method, then does: for k, v in E: D[k] = v  
In either case, this is followed by: for k in F: D[k] = F[k]

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

class SCons.Builder.CompositeBuilder (builder, cmdgen)  
Bases: 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.

add_action(suffix, action)

get ()  
Retrieve the entire wrapped object

class SCons.Builder.DictCmdGenerator (dict=None, source_ext_match=1)  
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()  
Create a new ordered dictionary with keys from iterable and values set to value.

get ()  
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()  
Move an existing element to the end (or beginning if last is false).  
Raise KeyError if the element does not exist.

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()
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.

```python
setdefault()
```
Insert key with a value of default if key is not in the dictionary.
Return the value for key if key is in the dictionary, else default.

```python
src_suffixes()
```

```python
update([E], **F) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v
In either case, this is followed by: for k in F: D[k] = F[k]
```

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

```python
class SCons.Builder.DictEmitter
```
```python
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.

```python
clear() → None. Remove all items from od.
```

```python
copy() → a shallow copy of od
```

```python
fromkeys()
```
Create a new ordered dictionary with keys from iterable and values set to value.

```python
get()
```
Return the value for key if key is in the dictionary, else default.

```python
items() → a set-like object providing a view on D's items
```

```python
keys() → a set-like object providing a view on D's keys
```

```python
move_to_end()
```
Move an existing element to the end (or beginning if last is false).
Raise KeyError if the element does not exist.

```python
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.
```

```python
popitem()
```
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.

```python
setdefault()
```
Insert key with a value of default if key is not in the dictionary.
Return the value for key if key is in the dictionary, else default.

```python
update([E], **F) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v
In either case, this is followed by: for k in F: D[k] = F[k]
```

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

```python
class SCons.Builder.EmitterProxy(var)
```
```python
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.

__UserList__cast (other)
__abc_impl = <_abc_data object>
append (item)
  S.append(value) – append value to the end of the sequence
clear () \rightarrow None -- remove all items from S
copy ()
count (value) \rightarrow integer -- return number of occurrences of value
extend (other)
  S.extend(iterable) – extend sequence by appending elements from the iterable
index (value [, start [, stop]]) \rightarrow 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]) \rightarrow 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.OverrideWarner(dict)
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.

__MutableMapping__marker = <object object>
__abc_impl = <_abc_data object>
clear () \rightarrow None. Remove all items from D.
copy ()
classmethod fromkeys (iterable, value=None)
get (k [, d]) \rightarrow D[k] if k in D, else d. d defaults to None.
items () \rightarrow a set-like object providing a view on D’s items
keys () \rightarrow a set-like object providing a view on D’s keys
pop (k [, d]) \rightarrow 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(κ [, 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
```

```
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=[])
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**

SCons.Conftest

Autoconf-like configuration support; low level implementation of tests.

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)

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 libraries 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.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 wether the type has the given number of bytes. If not given, will automatically find the size.
- **Returns** – status int 0 if the check failed, or the found size of the type if the check succeeded.

SCons.Conftest._Have(context, key, have, comment=None)

Store result of a test in context.havedict and context.headerfilename.

Parameters:

- **key** - is a “HAVE_abc” name. It is turned into all CAPITALS and non-alphanumerics are replaced by an underscore.
- **have** - value as it should appear in the header file, include quotes when desired and escape special characters!
- **comment** is the C comment to add above the line defining the symbol (the comment is automatically put inside a /* */). If None, no comment is added.

The value of “have” can be:
• 1 - Feature is defined, add “#define key”.
• 0 - Feature is not defined, add “#undef key */”. Adding “undef” is what autoconf does. Not useful for the compiler, but it shows that the test was done.
• number - Feature is defined to this number “#define key have”. Doesn’t work for 0 or 1, use a string then.
• string - Feature is defined to this 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
• suffix = the suffix, including the leading dot
• msg = an error message

---

**SCons.Debug module**

SCons.Debug
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, filename=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 file argument persists across calls unless overridden.

Parameters:
• filename – 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)
SCons.Debug.caller_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.
fetchedLoggedInstances(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

SCons.Defaults

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.msvc.compiler.

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)
Bases: object
This is a callable class that can be used in place of other command generators if you don’t want them to do anything.
The __call__ method for this class simply returns the thing you instantiated it with.
Example usage: env['DO NOTHING'] = NullCmdGenerator env['LINKCOM'] = "$(DO NOTHING($LINK $SOURCES $TARGET))"
SCons.Defaults.SharedFlagChecker(source, target, env)
SCons.Defaults.SharedObjectEmitter(target, source, env)
SCons.Defaults.StaticObjectEmitter(target, source, env)

class SCons.Defaults.Variable_Method_Caller(variable, method)
Bases: object
A class for finding a construction variable on the stack and calling one of its methods.
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._libversionflags(env, version_var, flags_var)
SCons.Defaults._concat(prefix, list, suffix, env, f=<function <lambda>>, target=None, source=None)
Creates a new list from ‘list’ 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 concatenated ‘prefix’ and ‘suffix’ onto each element of the list.
SCons.Defaults._concat_ixes(prefix, list, suffix, env)
SCons Project API Documentation

Creating a new list from 'list' by concatenating the 'prefix' and 'suffix' arguments onto each element of the list. A trailing space on 'prefix' or leading space on 'suffix' will cause them to be put into separate list elements rather than being concatenated.

SCons.Defaults._defines (prefix, defs, suffix, env, c=<function _concat_ixes>)
A wrapper around _concat_ixes that turns a list or string into a list of C preprocessor command-line definitions.

SCons.Defaults._fetch_DefaultEnvironment (*args, **kw)
Returns the already-created default construction environment.

SCons.Defaults._stripixes (prefix, itms, suffix, stripprefixes, stripsuffixes, env, c=None)
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 (dest, mode)
SCons.Defaults.chmod_strfunc (dest, mode)
SCons.Defaults.copy_func (dest, src, symlinks=True)
If symlinks (is true), then a symbolic link will be shallow copied and recreated as a symbolic link; otherwise, copying a symbolic link will be equivalent to copying the symbolic link's final target regardless of symbolic link depth.

SCons.Defaults.delete_func (dest, must_exist=0)
SCons.Defaults.delete_strfunc (dest, must_exist=0)
SCons.Defaults.get_paths_str (dest)
SCons.Defaults.mkdir_func (dest)
SCons.Defaults.move_func (dest, src)
SCons.Defaults.processDefines (defs)
    process defines, resolving strings, lists, dictionaries, into a list of strings
SCons.Defaults.touch_func (dest)

SCons.Environment module

SCons.Environment
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.

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

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 existing construction variables in an Environment.

**AppendENVPath** (name, newpath, envname='ENV', sep=':', delete_existing=0)  
Append path elements to the path ‘name’ in the ‘ENV’ 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 0, a newpath which is already in the path will not be moved to the end (it will be left where it is).

**AppendUnique** (delete_existing=0, **kw**)  
Append values to existing construction variables in an Environment, if they’re not already there. If delete_existing is 1, removes existing values first, so values move to end.

**Builder** (**kw**)  
**CacheDir** (path)  
**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 in progs.

**Dictionary** (*args)  
Return construction variables from an 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** *(optional)* – specify the format to serialize to. “pretty” generates a pretty-printed string, “json” a JSON-formatted string. (Default value = None, equivalent to ‘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=’.’)
returns 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=1, dict=None)
Merge the dict in args into the construction variables of this env, or the passed-in dict. If args is not a dict, it is converted into a dict using ParseFlags. If unique is not set, the flags are appended rather than merged.

**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=1)
Use the specified function to parse the output of the command in order to modify the current environment. The ‘command’ can be a string or a list of strings representing a command and its arguments. ‘Function’ is an optional argument that takes the environment, the output of the command, and the unique flag. If no function is specified, MergeFlags, which treats the output as the result of a typical ‘X-config’ command (i.e. gtk-config), will merge the output into the appropriate variables.

**ParseDepends**(filename, must_exist=None, only_one=0)
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*)
Parse the set of flags and return a dict with the flags placed in the appropriate entry. The flags are treated as a typical set of command-line flags for a GNU-like toolchain and used to populate the entries in the dict immediately below. If one of the flag 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 existing construction variables in an Environment.

**PrependENVPath**(name, newpath, envname='ENV', sep=':', delete_existing=1)
Prepend path elements to the path ‘name’ in the ‘ENV’ 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 0, a newpath which is already in the path will not be moved to the front (it will be left where it is).

**PrependUnique**(delete_existing=0, **kw**)
Prepend values to existing construction variables in an Environment, if they’re not already there. If delete_existing is 1, 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, **kw)

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

VariantDir(variant_dir, src_dir, duplicate=1)

WhereIs(prog, path=None, pathext=None, reject=[])
Find prog in the path.

_canonicalize(path)

_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)

_copy2_from_cache(src, dst)

_copy_from_cache(src, dst)

_find_toolpath_dir(tp)

_gsm()

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

_update(dict)
Update an environment’s values directly, bypassing 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)

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).

g_get_src_sig_type ()

g_get_tgt_sig_type ()

gvars ()

has_key (key)
   Emulates the has_key() method of dictionaries.

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

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

gvars ()

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

subst (string, raw=0, target=None, source=None, conv=None, executor=None)
   Recursively interpolates construction variables from the Environment into the specified string, returning the expanded result. Construction variables are specified by a $ prefix in the string and begin with an initial underscore or alphabetic character followed by any number of underscores or alphanumeric characters. The construction variable names may be surrounded by curly braces to separate the name from trailing characters.

subst_kw (kw, raw=0, target=None, source=None)

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

subst_path (path, target=None, source=None)
   Substitute a path list, 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)
   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.BuilderDict (dict, 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.

MutableMapping__marker = <object object>
_abcs_impl = <_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 (object, method, name=None)
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.

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

class SCons.Environment.MethodWrapper (object, method, name=None)
Bases: object
A generic Wrapper class that associates a method (which can actually be any callable) with an object. As part of creating this MethodWrapper object an attribute with the specified (by default, the name of the supplied method) is added to the underlying object. When that new “method” is called, our __call__() method adds the object as the first argument, simulating the Python behavior of supplying “self” on method calls.
We hang on to the name by which the method was added to the underlying base class so that we can provide a method to “clone” ourselves onto a new underlying object being copied (without which we wouldn't need to save that info).
clone(new_object)
   Returns an object that re-binds the underlying “method” to the specified new object.

SCons.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

class SCons.Environment.OverrideEnvironment(subject, overrides={})
   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 existing construction variables in an Environment.

AppendENVPath(name, newpath, envname='ENV', sep=':', delete_existing=0)
   Append path elements to the path ‘name’ in the ‘ENV’ 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 0, a newpath which is already in the path will not be moved to the end (it will be left where it
   is).

AppendUnique(delete_existing=0,**kw)
   Append values to existing construction variables in an Environment, if they’re not already there. If
   delete_existing is 1, removes existing values first, so values move to end.

Builder(**kw)

CacheDir(path)

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 in progs.

**Parameters:**
- `progs (str or list)` — one or more command names to check for
**Returns:**
- `str` — first name from progs that can be found.

**Dictionary** `()`
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 (optional)` — specify the format to serialize to. “pretty” generates a pretty-printed string, “json” a JSON-formatted string. (Default value = None, equivalent to “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='.'**) returns 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=1, dict=None**)  

Merge the dict in args into the construction variables of this env, or the passed-in dict. If args is not a dict, it is converted into a dict using ParseFlags. If unique is not set, the flags are appended rather than merged.

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=1**)  

Use the specified function to parse the output of the command in order to modify the current environment. The command can be a string or a list of strings representing a command and its arguments. ‘Function’ is an optional argument that takes the environment, the output of the command, and the unique flag. If no function is specified, MergeFlags, which treats the output as the result of a typical ’X-config’ command (i.e. gtk-config), will merge the output into the appropriate variables.

ParseDepends(**filename, must_exist=None, only_one=0**)  

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'**)  

Parse the set of flags and return a dict with the flags placed in the appropriate entry. The flags are treated as a typical set of command-line flags for a GNU-like toolchain and used to populate the entries in the dict immediately below. If one of the flag 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 existing construction variables in an Environment.

PrependENVPath(name, newpath, envname='ENV', sep=':', delete_existing=1)
Prepend path elements to the path ‘name’ in the ‘ENV’ 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 0, a newpath which is already in the path will not be moved to the front (it will be left where it is).

PrependUnique(delete_existing=0, **kw)
Prepend values to existing construction variables in an Environment, if they’re not already there. If delete_existing is 1, 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=toolpath=None, **kw)
Value(value, built_value=None, name=None)

VariantDir(variant_dir, src_dir, duplicate=1)

WhereIs(prog, path=None, pathext=None, reject=[])  
Find prog in the path.

_canonicalize(path)

_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)

_copy2_from_cache(src, dst)

_copy_from_cache(src, dst)

_find_toolpath_dir(tp)

_gsm()

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

_update(dict)  
Update an environment's values directly, bypassing 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)

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()

has_key(key)
Emulates the has_key() method of dictionaries.

```
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).

```
subst(string, raw=0, target=None, source=None, conv=None, executor=None)
```
Recursively interpolates construction variables from the Environment into the specified string, returning the expanded result. Construction variables are specified by a $ prefix in the string and begin with an initial underscore or alphabetic character followed by any number of underscores or alphanumeric characters. The construction variable names may be surrounded by curly braces to separate the name from trailing characters.

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

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

```
subst_path(path, target=None, source=None)
```
Substitute a path list, 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)
```
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.SubstitutionEnvironment(**kw)
```

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.)
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)
Parse the set of flags and return a dict with the flags placed in the appropriate entry. The flags are treated as a typical set of command-line flags for a GNU-like toolchain and used to populate the entries in the dict immediately below. If one of the flag 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)
get(key, default=None)
Emulates the get() method of dictionaries.

gvars()

has_key(key)
Emulates the has_key() method of dictionaries.

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

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

lvars()

subst(string, raw=0, target=None, source=None, conv=None, executor=None)
Recursively interpolates construction variables from the Environment into the specified string, returning the expanded result. Construction variables are specified by a $ prefix in the string and begin with an initial underscore or alphabetic character followed by any number of underscores or alphanumeric characters. The construction variable names may be surrounded by curly braces to separate the name from trailing characters.

subst_kw(kw, raw=0, target=None, source=None)

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

subst_path(path, target=None, source=None)
Substitute a path list, 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)
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

SCons.Environment._del_SCANNERS (env, key)
    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 (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.

\section{SCons.Errors module}

This file contains the exception classes used to handle internal and user errors in SCons.

\begin{verbatim}
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.

    Information about the cause of the build error

    errstr
        a description of the error message

    status
        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
        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
        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.

    exc_info
\end{verbatim}

163
Info about exception that caused the build error. Set to (None, None, None) if this build error is not due to an exception.

**Information about the what caused the build error**

- **node**
  - the error occurred while building this target node(s)

- **executor**
  - the executor that caused the build to fail (might be None if the build failures is not due to the executor failing)

- **action**
  - the action that caused the build to fail (might be None if the build failures is not due to the an action failure)

- **command**
  - 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)

- **args**

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

**exception** `SCons.Errors.ExplicitExit` (node=None, status=None, *args)

- Bases: `Exception`

  ```python
  args
  ```

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

**exception** `SCons.Errors.InternalError`

- Bases: `Exception`

  ```python
  args
  ```

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

**exception** `SCons.Errors.MSVCError`

- Bases: `OSError`

  ```python
  args
  ```

  ```python
  characters_written
  ```

  ```python
  errno
  POSIX exception code
  ```

  ```python
  filename
  exception filename
  ```

  ```python
  filename2
  second exception filename
  ```

  ```python
  strerror
  exception strerror
  ```

  ```python
  with_traceback ()
  Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.
  ```
exception SCons.Errors.SConsEnvironmentError
    Bases: Exception

    args

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

exception SCons.Errors.StopError
    Bases: Exception

    args

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

exception SCons.Errors.UserError
    Bases: Exception

    args

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

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**

SCons.Executor

A module for executing actions with specific lists of target and source Nodes.

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 $N \times M$ 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()
   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.Null

SCons = <module 'SCons' from '/Users/bdbaddog/devel/scons/git/scons-2/SCons/__init__.py'>

_CacheDir = <SCons.CacheDir.CacheDir object>

_CacheDir_path = None

get_CacheDir()

class SCons.Executor.TSList(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.

__UserList__cast(other)

__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.
Raising 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(i, 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.TSObject(func):
    Bases: object
    A class that implements $TARGET or $SOURCE expansions by wrapping an Executor method.

SCons.Executor.execute_action_list(obj, target, kw)
    Actually execute the action list.

SCons.Executor.execute_actions_str(obj)

SCons.Executor.execute_nothing(obj, target, kw)

SCons.Executor.execute_null_str(obj)
    Use singleton pattern for Null Environments.

SCons.Executor.rfile(node)
    A function to return the results of a Node’s rfile() method, if it exists, and the Node itself otherwise (if it’s a Value Node, e.g.).

## SCons.Job module

SCons.Job
This module defines the Serial and Parallel classes that execute tasks to complete a build. The Jobs class provides a higher level interface to start, stop, and wait on jobs.

class SCons.Job.InterruptState:
    Bases: object
    set()

class SCons.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.Job.Parallel(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.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.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.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.

    _exc_info()
    _exc_info() -> (type, value, traceback)
    Return information about the most recent exception caught by an except clause in the current stack frame or in an older stack frame.

    _initialized = False

    _reset_internal_locks(is_alive)

    _set_ident()

    _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()

**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.

isAlive()
Return whether the thread is alive.
This method is deprecated, use is_alive() instead.

isDaemon()

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. The
module function enumerate() returns a list of all alive threads.

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.

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)

setName(name)

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.
Memoizer

A decorator-based implementation to count hits and misses of the computed values that various methods cache in memory.

Use of this module assumes that wrapped methods be coded to cache their values in a consistent way. In particular, it requires that the class uses a dictionary named "_memo" to store the cached values.

Here is an example of wrapping a method that returns a computed value, with no input parameters:

```python
@SCons.Memoize.CountMethodCall
def foo(self):
    try:
        return self._memo['foo']
    except KeyError:
        pass
    result = self.compute_foo_value()
    self._memo['foo'] = result
    return result
```

Here is an example of wrapping a method that will return different values based on one or more input arguments:

```python
def _bar_key(self, argument):
    return argument

@SCons.Memoize.CountDictCall(_bar_key)
def bar(self, argument):
    memo_key = argument
    try:
        memo_dict = self._memo['bar']
    except KeyError:
        memo_dict = {}
    self._memo['dict'] = memo_dict
    else:
        try:
            return memo_dict[memo_key]
        except KeyError:
            pass
    result = self.compute_bar_value(argument)
    memo_dict[memo_key] = result
    return result
```

Deciding what to cache is tricky, because different configurations can have radically different performance tradeoffs, and because the tradeoffs involved are often so non-obvious. Consequently, deciding whether or not to cache a given method will likely be more of an art than a science, but should still be based on available data from this module. Here are some VERY GENERAL guidelines about deciding whether or not to cache return values from a method that's being called a lot:

- The first question to ask is, "Can we change the calling code so this method isn't called so often?" Sometimes this can be done by changing the algorithm. Sometimes the caller should be memoized, not the method you're looking at.

The memoized function should be timed with multiple configurations to make sure it doesn't inadvertently slow down some other configuration.
– When memoizing values based on a dictionary key composed of input arguments, you don’t need to use all of the arguments if some of them don’t affect the return values.

class SCons.Memoize.CountDict (cls_name, method_name, keymaker)
Bases: 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 ()

SCons.Memoize.CountDictCall (keyfunc)
Decorator for counting memoizer hits/misses while accessing dictionary values with a key-generating function. Like CountMethodCall above, it wraps the given method fn and uses a CountDict object to keep track of the caching statistics. The dict-key function keyfunc has to get passed in the decorator call and gets stored in the CountDict instance. Wrapping gets enabled by calling EnableMemoization().

SCons.Memoize.CountMethodCall (fn)
Decorator for counting memoizer hits/misses while retrieving a simple value in a class method. It wraps the given method fn and uses a CountValue object to keep track of the caching statistics. Wrapping gets enabled by calling EnableMemoization().

class SCons.Memoize.CountValue (cls_name, method_name)
Bases: 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 ()

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 ()

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

SCons.Memoize.EnableMemoization ()

SCons.PathList module

SCons.PathList
A module for handling lists of directory paths (the sort of things that get set as CPPPATH, LIBPATH, etc.) with as much caching of data and efficiency as we can, while still keeping the evaluation delayed so that we Do the Right Thing (almost) regardless of how the variable is specified.
SCons.PathList.PathList (pathlist)

Returns the cached _PathList object for the specified pathlist, creating and caching a new object as necessary.

class SCons.PathList._PathList (pathlist)
Bases: object

An actual PathList object.

subst_path (env, target, source)

Performs construction variable substitution on a pre-digested PathList for a specific target and source.

SCons.PathList.node_conv (obj)

This is the “string conversion” routine that we have our substitutions use to return Nodes, not strings. This relies on the fact that an EntryProxy object has a get() method that returns the underlying Node that it wraps, which is a bit of architectural dependence that we might need to break or modify in the future in response to additional requirements.

### SCons.SConf module

SCons.SConf

Autoconf-like configuration support.

In other words, SConf allows to run tests on the build machine to detect capabilities of system and do some things based on result: generate config files, header files for C/C++, update variables in environment.

Tests on the build system can detect if compiler sees header files, if libraries are installed, if some command line options are supported etc.

SCons.SConf.CheckCC (context)

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:

```
def CheckCustom(context, ...):
    context.Message('Checking my weird test ... ') ret = myWeirdTestFunction(...) context.Result(ret)
```

Often, myWeirdTestFunction will be one of context.TryCompile/context.TryLink/context.TryRun. The results of those are cached, for they are only rebuild, if the dependencies have changed.

AppendLIBS (lib_name_list)

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)
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.

SCons.SConf.CheckLib(context, library=None, symbol='main', header=None, language=None, autoadd=1)
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=1)
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.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)

eception SCons.SConf.ConfigureCacheError(target)
Bases: SCons.SConf.SConfError
Raised when a use explicitely 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.

eception 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 Project API Documentation

SCons.SConf.NeedConfigHBuilder()
SCons.SConf(*args, **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_instance 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).

_createDir (node)

_shutdown ()
Private method. Reset to non-piped spawn

_startup()

Private method. Set up logstream, and set the environment variables necessary for a piped build

pspawn_wrapper (sh, escape, cmd, args, env)

Wrapper function for handling piped spawns.
This looks to the calling interface (in Action.py) like a "normal" spawn, but associates the call with the PSPAWN variable from the construction environment and with the streams to which we want the output logged. This gets slid into the construction environment as the SPAWN variable so Action.py doesn't have to know or care whether it's spawning a piped command or not.

class SCons.SConf.SConfBuildInfo
Bases: 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.

bact
bactsig
bdepants
bdepantsigs
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.

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, bdepants 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)

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.

_ABC_impl = <_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

eexecute ()
    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 ()

179
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:

```python
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**(method, node, description='node')

**exception** SCons.SConf.SConfError(msg)  
Bases: SCons.Errors.UserError
with_traceback ()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception SCons.SConf.SConfWarning
    Bases: SCons_WARNINGS.Warning

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

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 ()
        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._stringConfigH (target, source, env)
SCons.SConf._stringSource (target, source, env)
SCons.SConf.createIncludesFromHeaders (headers, leaveLast, include_quotes='""')

SCons.SConsign module

SCons.SConsign
Writing and reading information to the .sconsign file or files.

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.write()
```

---

**SCons.Subst module**

SCons.Subst

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_data object>

   capitalize()

   casefold()

   center(width, *args)

   count(value) → integer -- return number of occurrences of value

   encode(encoding=None, errors=None)

   endswith(suffix, start=0, end=9223372036854775807)
```
**escape** *(escape_func, quote_func=<function quote_spaces>)*

Escape the string with the supplied function. The function is expected to take an arbitrary string, then return it with all special characters escaped and ready for passing to the command interpreter. After calling this function, the next call to `str()` will return the escaped string.

**expandtabs** *(tabsize=8)*

**find** *(sub, start=0, end=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** *

**isdecimal** *

**isdigit** *

**isidentifier** *

**islower** *

**isnumeric** *

**isprintable** *

**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)*
replace(old, new, maxsplit=-1)

rfind(sub, start=0, end=9223372036854775807)

rindex(sub, start=0, end=9223372036854775807)

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=9223372036854775807)

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.

_add_new_word(x)

Add 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.

_add_to_current_word(x)

Add 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

expand (s, lvars, within_list)
Expand a single “token” as necessary, appending the expansion to the current result.
This handles expanding different types of things (strings, lists, callables) appropriately. It calls the wrapper
substitute() method to re-expand things as necessary, so that the results of expansions of side-by-side strings
still get re-evaluated separately, not smushed together.

expanded (s)
Determines if the string s requires further expansion.
Due to the implementation of ListSubber expand will call itself 2 additional times for an already expanded string.
This method is used to determine if a string is already fully expanded and if so exit the loop early to prevent
these recursive calls.

extend (other)
S.extend(iterable) – extend sequence by appending elements from the iterable

index (value [, start [, stop]]) → integer -- return first index of value.
Raises ValueError if the value is not present.
Supporting start and stop arguments is optional, but recommended.

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

literal (x)

next_line ()
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.NullSeq

    _instance

SCons.Subst.NullNodesList

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 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.

```python
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.

    __UserList__cast__(other)
    __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, item)
    S.insert(index, value) – insert value before index

    pop([, index]) → item -- remove and return item at index (default last).
    Raise IndexError if list is empty or index is out of range.

    remove(item)
    S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

    reverse()
    S.reverse() – reverse IN PLACE

    sort(*args, **kwds)

SCons.Subst._remove_list(list)
SCons.Subst._rm_list(list)
SCons.Subst.escape_list(mylist, escape_func)
    Escape a list of arguments by running the specified escape_func on every object in the list that has an escape() method.

SCons.Subst.quote_spaces(arg)
    Generic function for putting double quotes around any string that has white space in it.

SCons.Subst.raise_exception(exception, target, s)
SCons.Subst.scons_subst(strSubst, env, mode=1, target=None, source=None, gvars={}, lvars={}, conv=None)
    Expand a string or list containing construction variable substitutions.
    This is the work-horse function for substitutions in file names and the like. The companion scons_subst_list() function (below) handles separating command lines into lists of arguments, so see that function if that’s what you’re looking for.
```
SCons.Subst.scons_subst_list (strSubst, env, mode=1, target=None, source=None, gvars={}, lvars={}, conv=None)

Substitute construction variables in a string (or list or other object) and separate the arguments into a command list.
The companion scons_subst() function (above) handles basic substitutions within strings, so see that function instead if that’s what you’re looking for.

SCons.Subst.scons_subst_once (strSubst, env, key)

Perform single (non-recursive) substitution of a single construction variable keyword.
This is used when setting a variable when copying or overriding values in an Environment. We want to capture (expand) the old value before we override it, so people can do things like:

    env2 = env.Clone(CCFLAGS = '${CCFLAGS -g}')

We do this with some straightforward, brute-force code here…

SCons.Subst.subst_dict (target, source)

Create a dictionary for substitution of special construction variables.
This translates the following special arguments:

- **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.Taskmaster module**

**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

    _abc_impl = _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.

```python
exc_clear()
```
Clears any recorded exception. This also changes the "exception_raise" attribute to point to the appropriate do-nothing method.

```python
exc_info()
```
Returns info about a recorded exception.

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

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

```python
executed()
```
Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s callback methods. This may have been a do-nothing operation (to preserve build order), so we must check the node’s state before deciding whether it was “built”, in which case we call the appropriate Node method. In any event, we always call “visited()”, which will handle any post-visit actions that must take place regardless of whether or not the target was an actual built target or a source Node.

```python
executed_with_callbacks()
```
Called when the task has been successfully executed and the Taskmaster instance wants to call the Node’s callback methods. This may have been a do-nothing operation (to preserve build order), so we must check the node’s state before deciding whether it was “built”, in which case we call the appropriate Node method. In any event, we always call “visited()”, which will handle any post-visit actions that must take place regardless of whether or not the target was an actual built target or a source Node.

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

```python
fail_continue()
```
Explicit continue-the-build failure. This sets failure status on the target nodes and all of their dependent parent nodes. Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

```python
fail_stop()
```
Explicit stop-the-build failure. This sets failure status on the target nodes and all of their dependent parent nodes. Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

```python
failed()
```
Default action when a task fails: stop the build. Note: Although this function is normally invoked on nodes in the executing state, it might also be invoked on up-to-date nodes when using Configure().

```python
get_target()
```
Fetch the target being built or updated by this task.
**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(method, node, description='node')**

```class SCons.Taskmaster.OutOfDateTask (tm, targets, top, node)
Bases: SCons.Taskmaster.Task

_abc_impl = <_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(method, 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.

    _abc_impl = <_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.

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(method, 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:

```
  ^
 /|
+----------------+
  Next candidate
```

Now, when the Taskmaster examines the Node C’s child Node A, it finds that Node A is in the “pending” state.
Therefore, Node A is a pending child of node C.
Pending children indicate that the Taskmaster has potentially loop back through a cycle. We say potentially
because it could also occur when a DAG is evaluated in parallel. For example, consider the following graph:

```
Node A (Pending) --> Node B(Pending) --> Node C (Pending) --> ...
  ^
 /|
+--------> Node D (NoState) ------+
     /                          /
Next candidate /
```

The Taskmaster first evaluates the nodes A, B, and C and starts building some children of node C. Assuming,
that the maximum parallel level has not been reached, the Taskmaster will examine Node D. It will find that
Node C is a pending child of Node D.
In summary, evaluating a graph with a cycle will always involve a pending child at one point. A pending child
might indicate either a cycle or a diamond-shaped DAG. Only a fraction of the nodes ends-up being a “pending
child” of another node. This keeps the pending_children set small in practice.
We can differentiate between the two cases if we wait until the end of the build. At this point, all the pending
children nodes due to a diamond-shaped DAG will have been properly built (or will have failed to build). But, the
pending children involved in a cycle will still be in the pending state.
The taskmaster removes nodes from the pending_children set as soon as a pending_children node moves out
of the pending state. This also helps to keep the pending_children set small.
cleanup()
Check for dependency cycles.

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.

trace_message(message)

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.Util module

SCons.Util
Various utility functions go here.

SCons.Util.AddMethod(obj, function, name=None)
Adds either a bound method to an instance or the function itself (or an unbound method in Python 2) to a class. If
name is ommited the name of the specified function is used by default.
Example:

```python
a = A()
def f(self, x, y):
    self.z = x + y
AddMethod(f, A, "add")
a.add(2, 4)
print(a.z)
AddMethod(lambda self, i: self.l[i], a, "listIndex")
print(a.listIndex(5))
```

SCons.Util.AddPathIfNotExists(env_dict, key, path, sep=': ')
This function will take ‘key’ out of the dictionary ‘env_dict’, then add the path ‘path’ to that key if it is not already
there. This treats the value of env_dict[key] as if it has a similar format to the PATH variable…a list of paths
separated by tokens. The ‘path’ will get added to the list if it is not already there.
SCons.Util.AppendPath(oldpath, newpath, sep=':', delete_existing=1, canonicalize=None)
This appends new path elements to the given old path. Will only add any particular path once (leaving the last one
it encounters and ignoring the rest, to preserve path order), and will os.path.normpath and os.path.normcase all
paths to help assure this. This can also handle the case where the given old path variable is a list instead of a
string, in which case a list will be returned instead of a string.

Example
Old Path: “/foo/bar:/foo” New Path: “/biz/boom:/foo” Result: “/foo/bar:/biz/boom:/foo”
If delete_existing is 0, then adding a path that exists will not move it to the end; it will stay where it is in the list.
If canonicalize is not None, it is applied to each element of newpath before use.

class SCons.Util.CLVar(seq=[])
Bases: collections.UserList
A class for command-line construction variables.
This is a list that uses Split() to split an initial string along white-space arguments, and similarly to split any strings
that get added. This allows us to Do the Right Thing with Append() and Prepend() (as well as straight Python foo =
env['VAR'] + 'arg1 arg2') regardless of whether a user adds a list or a string to a command-line construction
variable.

__UserList__cast(other)

__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, item)
S.insert(index, value) – insert value before index

pop([], index)→ item -- remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

remove(item)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

reverse()
S.reverse() – reverse IN PLACE

sort(*args, **kwds)
class SCons.Util.Delegate(attribute)
Bases: object
A Python Descriptor class that delegates attribute fetches to an underlying wrapped subject of a Proxy. Typical
use:

class Foo(Proxy):
    __str__ = Delegate('__str__')
class SCons.Util.DisplayEngine
Bases: object

  print_it = True

  set_mode (mode)

SCons.Util_IDX (N)

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 ()

SCons.Util_MD5collect (signatures)
Collects a list of signatures into an aggregate signature. Signatures - a list of signatures returns - the aggregate signature

SCons.Util_MD5filesignature (fname, chunksize=65536)
Generate the md5 signature of a file

  Parameters:
  • fname – file to hash
  • chunksize – chunk size to read

  Returns: String of Hex digits representing the signature

SCons.Util_MD5signature (s)
Generate md5 signature of a string

  Parameters: s – either string or bytes. Normally should be bytes

  Returns: String of hex digits representing the signature

class SCons.Util.NodeList (initlist=None)
Bases: collections.UserList
This class is almost exactly like a regular list of Nodes (actually it can hold any object), with one important difference. If you try to get an attribute from this list, it will return that attribute from every item in the list. For example:

```python
>>> someList = NodeList([ '  foo  ', '  bar  ' ])
>>> someList.strip()
[ 'foo', 'bar' ]
```

_SUserList__cast (other)

__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.

```python
insert (i, item)
S.insert(index, value) – insert value before index
```

```python
pop ([i, index]) → item -- remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.
```

```python
remove (item)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.
```

```python
reverse ()
S.reverse() – reverse IN PLACE
```

```python
sort (*args, **kwds)
```

```python
class SCons.Util.Null (*args, **kwargs)
    Bases: object
    Null objects always and reliably “do nothing.”
```

```python
class SCons.Util.NullSeq (*args, **kwargs)
    Bases: SCons.Util.Null
    A Null object that can also be iterated over.
```

```python
class SCons.Util.PowerPath (*args, **kwargs)
    Bases: object
```

```python
exception SCons.Util.PlainWindowsError
    Bases: OSError
```

```python
args
characters_written
errno
    POSIX exception code
filename
    exception filename
filename2
    second exception filename
strerror
    exception strerror

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

```python
SCons.Util.PrependPath (oldpath, newpath, sep=':', delete_existing=1, canonicalize=None)
This prepends newpath elements to the given oldpath. Will only add any particular path once (leaving the first one it encounters and ignoring the rest, to preserve path order), and will os.path.normpath and os.path.normcase all paths to help assure this. This can also handle the case where the given old path variable is a list instead of a string, in which case a list will be returned instead of a string.
```

```python
Example
Old Path: “/foo/bar:/foo” New Path: “/biz/boom:/foo” Result: “/biz/boom:/foo:/foo/bar”
If delete_existing is 0, then adding a path that exists will not move it to the beginning; it will stay where it is in the list.
If canonicalize is not None, it is applied to each element of newpath before use.
```

```python
class SCons.Util.Proxy (subject)
    Bases: object
```

199
A simple generic Proxy class, forwarding all calls to subject. So, for the benefit of the python newbie, what does this really mean? Well, it means that you can take an object, let’s call it ‘objA’, and wrap it in this Proxy class, with a statement like this

```python
proxyObj = Proxy(objA),
```

Then, if in the future, you do something like this

```python
x = proxyObj.var1,
```

since Proxy does not have a ‘var1’ attribute (but presumably objA does), the request actually is equivalent to saying

```python
x = objA.var1
```

Inherit from this class to create a Proxy.

Note that, with new-style classes, 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 `__*__()` method names you want to pass on to the underlying Proxy object, and specifically delegate their calls like this:

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

`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)
SCons.Util.RenameFunction (function, name)
    Returns a function identical to the specified function, but with the specified name.
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
    fromkeys ()
        Create a new ordered dictionary with keys from iterable and values set to value.
    get ()
        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 ()
        Move an existing element to the end (or beginning if last is false).
        Raise KeyError if the element does not exist.
    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 ()
        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 ()

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

SCons.Util.Split (arg)

class SCons.Util.Unbuffered (file)
Bases: object
A proxy class that wraps a file object, flushing after every write, and delegating everything else to the wrapped object.

write (arg)

class SCons.Util.UniqueList (seq=[])
Bases: collections.UserList

__make_unique ()
__cast (other)
__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)
S.insert(index, value) – insert value before index

pop ([i, 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.Util.WhereIs (file, path=None, pathext=None, reject=None)

SCons.Util.WinError
alias of SCons.Util.PlainWindowsError
exception SCons.Util._NoError
    Bases: Exception

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

SCons.Util._semi_deepcopy_list (x)
SCons.Util._semi_deepcopy_tuple (x)
SCons.Util.adjustixes (fname, pre, suf, ensure_suffix=False)
SCons.Util.case_sensitive_suffixes (s1, s2)
SCons.Util.cmp (a, b)
    Define cmp because it's no longer available in python3 Works under python 2 as well
SCons.Util.containsAll (str, set)
    Check whether sequence str contains ALL of the items in set.
SCons.Util.containsAny (str, set)
    Check whether sequence str contains ANY of the items in set.
SCons.Util.containsOnly (str, set)
    Check whether sequence str contains ONLY items in set.
SCons.Util.dictify (keys, values, result={})
SCons.Util.do_flatten (sequence, result, instanceof=<built-in function isinstance>, StringTypes=\(<\text{class 'str'}>, \text{class 'collections.UserString'}\), SequenceTypes=\(<\text{class 'list'}>, \text{class 'tuple'}>, \text{class 'collections.UserList'}>, \text{class 'collections.abc.MappingView'}\))
SCons.Util.flatten (obj, instanceof=<built-in function isinstance>, StringTypes=\(<\text{class 'str'}>, \text{class 'collections.UserString'}\), SequenceTypes=\(<\text{class 'list'}>, \text{class 'tuple'}>, \text{class 'collections.UserList'}>, \text{class 'collections.abc.MappingView'}\), do_flatten=<function do_flatten>)
    Flatten a sequence to a non-nested list.
    Flatten() 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 Python would.
SCons.Util.flatten_sequence (sequence, instanceof=<built-in function isinstance>, StringTypes=\(<\text{class 'str'}>, \text{class 'collections.UserString'}\), SequenceTypes=\(<\text{class 'list'}>, \text{class 'tuple'}>, \text{class 'collections.UserList'}>, \text{class 'collections.abc.MappingView'}\), do_flatten=<function do_flatten>)
    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_env_bool (env, name, default=False)
    Convert a construction variable to bool.
    If the value of name in env is 'true', 'yes', 'y', 'on' (case insensitive) or anything convertible to int that yields non-zero then return True; if 'false', 'no', 'n', 'off' (case insensitive) or a number that converts to integer zero return False. Otherwise, return default.

Parameters:
    env – construction environment, or any dict-like object
    name – name of the variable
    default – value to return if name not in env or cannot be converted (default: False)

Returns:
    the "truthiness" of name

Return type:
    bool
SCons.Util.get_environment_var (varstr)
    Given a string, first determine if it looks like a reference to a single environment variable, like "$FOO" or "${FOO}". If so, return that variable with no decorations ("FOO"). If not, return None.
SCons.Util.get_native_path(path)
Transforms an absolute path into a native path for the system. Non-Cygwin version, just leave the path alone.

SCons.Util.get_os_env_bool(name, default=False)
Convert an environment variable to bool.
Conversion is the same as for get_env_bool().

SCons.Util.is_Dict(obj, isinstantiate=<built-in function isinstance>, DictTypes=(<class 'dict'>,<class 'collections.UserDict'>))

SCons.Util.is_List(obj, isinstantiate=<built-in function isinstance>, ListTypes=(<class 'list'>,<class 'collections.UserList'>))

SCons.Util.is_Scalar(obj, isinstantiate=<built-in function isinstance>, StringTypes=(<class 'str'>,<class 'collections.UserString'>), SequenceTypes=(<class 'list'>,<class 'collections.UserList'>,<class 'collections.abc.MappingView'>))

SCons.Util.is_Sequence(obj, isinstantiate=<built-in function isinstance>, SequenceTypes=(<class 'list'>,<class 'tuple'>,<class 'collections.UserList'>,<class 'collections.abc.MappingView'>))

SCons.Util.is_String(obj, isinstantiate=<built-in function isinstance>, StringTypes=(<class 'str'>,<class 'collections.UserString'>))

SCons.Util.is_Tuple(obj, isinstantiate=<built-in function isinstance>, tuple=<class 'tuple'>)

SCons.Util.logical_lines(physical_lines, joiner=<built-in method join of str object>)

SCons.Util.make_path_relative(path)
makes an absolute path name to a relative pathname.

SCons.Util.print_tree(root, child_func, prune=0, showtags=0, margin=[0], visited=None, lastChild=False, singleLineDraw=False)
Print a tree of nodes. This is like render_tree, except it prints lines directly instead of creating a string representation in memory, so that huge trees can be printed.

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 not prune, or in the whole tree if prune.
- singleLineDraw: use line-drawing characters rather than ASCII.

SCons.Util.render_tree(root, child_func, prune=0, margin=[0], visited=None)
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 not prune, or in the whole tree if prune.

SCons.Util.rightmost_separator(path, sep)

SCons.Util.semi_deepcopy(x)
SCons.Util.semi_deepcopy_dict(x, exclude=[])  
SCons.Util.silent_intern(x)  
    Perform sys.intern() on the passed argument and return the result. If the input is ineligible the original argument is returned and no exception is thrown.
SCons.Util.splitext(path)  
    Same as os.path.splitext() but faster.
SCons.Util.to_string(s, isinstance=<built-in function isinstance>, str=<class 'str'>, UserString=<class 'collections.UserString'>, BaseStringTypes=<class 'str'>)  
SCons.Util.to_string_for_signature(obj, to_string_for_subst=<function to_string_for_subst>, AttributeError=<class 'AttributeError'>)  
SCons.Util.to_string_for_subst(s, isinstance=<built-in function isinstance>, str=<class 'str'>, SequenceTypes=(<class 'list'>, <class 'tuple'>, <class 'collections.UserList'>, <class 'collections.abc.MappingView'>), UserString=<class 'collections.UserString'>)  
SCons.Util.to_bytes(s)  
SCons.Util.to_str(s)  
SCons.Util.unique(s)  
    Return a list of the elements in s, but without duplicates.
For example, unique([1, 2, 3, 1, 2, 3]) is some permutation of [1, 2, 3], unique("abcabc") some permutation of ["a", "b", "c"], and unique(((1, 2), (2, 3), (1, 2))) some permutation of [(1, 2), (2, 3), (1, 2)].
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*log2(N)) time.
If that's not possible either, the sequence elements must support equality-testing. Then unique() will usually work in quadratic time.
SCons.Util.uniquer(seq, idfun=None)  
SCons.Util.uniquer_hashables(seq)  
SCons.Util.updrive(path)  
    Make the drive letter (if any) upper case. This is useful because Windows is inconsistent on the case of the drive letter, which can cause inconsistencies when calculating command signatures.

SCons Warnings module

SCons.Warnings
This file implements the warnings framework for SCons.

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.Warning

    args

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

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

exception SCons.Warnings.DependencyWarning  
Bases: SCons.Warnings.Warning

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.Warning

args

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

exception SCons.Warnings.DevelopmentVersionWarning  
Bases: SCons.Warnings.WarningOnByDefault

args

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

exception SCons.Warnings.TargetNotBuiltWarning
    Bases: SCons.Warnings.Warning
    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.VisualCMissingWarning
    Bases: SCons.Warnings.WarningOnByDefault
Exception with traceback (tb) — set self.__traceback__ to tb and return self.

**exception** SCons.Warnings.VisualStudioMissingWarning
Bases: SCons.Warnings.Warning

Exception with traceback (tb) — set self.__traceback__ to tb and return self.

**exception** SCons.Warnings.VisualVersionMismatch
Bases: SCons.Warnings.WarningOnByDefault

Exception with traceback (tb) — set self.__traceback__ to tb and return self.

**exception** SCons.Warnings.Warning
Bases: SCons.Errors.UserError

Exception with traceback (tb) — set self.__traceback__ to tb and return self.

**exception** SCons.Warnings.WarningOnByDefault
Bases: SCons.Warnings.Warning

Exception with traceback (tb) — set self.__traceback__ to tb and return self.

SCons.Warnings.enableWarningClass (clazz)
Enables all warnings that are 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 in order to get an actual class name from the classes above, which we need to pass to the {enable,disable}WarningClass() functions. The supplied <warning-class> is split on hyphens, each element is capitalized, then smushed back together. Then the string “Warning” is appended to get the class name. 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 Warning class of all warnings.

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

SCons.Warnings.warn (clazz,*args)

SCons.Warnings.warningAsException (flag=1)
Turn warnings into exceptions. Returns the old value of the flag.

**SCons.cpp module**

SCons C Pre-Processor module
SCons Project API Documentation

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.

_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_included_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.

class SCons.cpp.PreProcessor(current='.', cpppath=(), dict={}, all=0, depth=-1)
    Bases: object
    The main workhorse class for handling C pre-processing.

    _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.

---

**SCons.dblite module**

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

**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 ()**

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.

texterror 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.

newline 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 newlines 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 `closed` 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`. 

SCons Project API Documentation

213
It is also possible to use a string or bytearray as a file for both reading and writing. For strings StringIO can be used like a file opened in a text mode, and for bytes a BytesIO can be used like a file opened in a binary mode.

**_os_chmod ()**

Change the access permissions of a file.

- **path**
  - Path to be modified. May always be specified as a str, bytes, or a path-like object. On some platforms, path may also be specified as an open file descriptor. If this functionality is unavailable, using it raises an exception.

- **mode**
  - Operating-system mode bitfield.

- **dir_fd**
  - If not None, it should be a file descriptor open to a directory, and path should be relative; path will then be relative to that directory.

- **follow_symlinks**
  - If False, and the last element of the path is a symbolic link, chmod will modify the symbolic link itself instead of the file the link points to.

It is an error to use **dir_fd** or **follow_symlinks** when specifying path as an open file descriptor.

**dir_fd** and **follow_symlinks** may not be implemented on your platform.

If they are unavailable, using them will raise a NotImplementedError.

**_os_chown ()**

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.

**_os_rename ()**

Rename a file or directory.

**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.
_os_unlink ()
Remove a file (same as remove()).

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.

*dir_fd may not be implemented on your platform.*
If it is unavailable, using it will raise a NotImplementedException.

*static* **_pickle_dump ()**
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 and
4. The default protocol is 3; a backward-incompatible protocol designed for Python 3.
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.

*static* **_pickle_protocol = 4**

*static* **_shutil_copyfile (dst, *, follow_symlinks=True)**
Copy data from src to dst.
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.

*static* **_time_time ()**
  *time() -> floating point number*
  Return the current time in seconds since the Epoch. Fractions of a second may be present if the system clock
  provides them.

*close ()*

*has_key (key)*

*iterkeys ()*

*keys ()*

*sync ()*

*SCons.dblite.open (file, flag=None, mode=438)*

**SCons.exitfuncs module**

SCons.exitfuncs
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 that we’d like to use in SCons but which only show up in later versions of Python than the early, old version(s) we still support.

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.

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
   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

scons.Node.Alias
   Alias nodes.
   This creates a hash of global Aliases (dummy targets).

class SCons.Node.Alias.Alias (name)

class Attrs
   Bases: object

   shared

   BuildInfo
      alias of AliasBuildInfo
Decider(function)

GetTag(key)
   Return a user-defined tag.

NodeInfo
   alias of AliasNodeInfo

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.

\texttt{alter\_targets()} \\
Return a list of alternate targets for this Node.

\texttt{always\_build} \\
\texttt{attributes} \\
\texttt{binfo} \\
\texttt{build()} \\
A “builder” for aliases.

\texttt{builder} \\
\texttt{builder\_set(builder)} \\
\texttt{built()} \\
Called just after this node is successfully built.

\texttt{cached} \\
\texttt{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 \texttt{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()

\texttt{changed\_since\_last\_build} \\
\texttt{children(scan=1)} \\
Return a list of the node’s direct children, minus those that are ignored by this node.

\texttt{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.


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

\texttt{clear\_memoized\_values()} \\
\texttt{convert()} \\
\texttt{del\_binfo()} \\
Delete the build info from this node.

\texttt{depends} \\
\texttt{depends\_set} \\
\texttt{disambiguate(must\_exist=\texttt{None})}
SCons.compat package

```python
eval
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.

```python
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.

```python
get_ninfo()
```

```python
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.

```python
get_state()
```

```python
get_stored_implicit()
```
Fetch the stored implicit dependencies

```python
get_stored_info()
```

```python
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.

```python
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.

```python
get_suffix()
```

```python
get_target_scanner()
```

```python
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 `__nonzero__` attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.

```python
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).

```python
ignore
```

```python
ignore_set
```

```python
implicit
```
implicit_set

includes

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 __nonzero__ 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().

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 (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

bact

bactsig

bdepends

bdependsigs

bimplicit
SCons.compat package

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(**kwargs)
    Bases: collections.UserDict

    Alias(name, **kw)

    _MutableMapping__marker = <object object>

    _abc_impl = <_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]]), *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


    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**

scons.Node.FS
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)

_abspath

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

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

cleared(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 now we 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

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)

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.

fs
    Reference to parent Node.FS object

get_abspath()
    Get the absolute path of the file.

get_binfo()
    Fetch a node’s build information.
    node - the node whose sources will be collected cache - alternate node to use for the signature cache returns - the build signature
    This no longer handles the recursive descent of the node’s children’s signatures. We expect that they’re already built and updated by someone else, if that’s what’s wanted.

get_build_env()
    Fetch the appropriate Environment to build this node.

get_build_scanner_path(scanner)
    Fetch the appropriate scanner path for this node.

get_builder(default_builder=None)
    Return the set builder, or a specified default value

get_cachedir_csig()
get_contents()
    Fetch the contents of the entry.

get_csig()

get_dir()

get_env()

get_env_scanner(env, kw={})

get_executor(create=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_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.

```python
get_suffix()
get_target_scanner()
get_tpath()
gmtime()
gsize()

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 `__nonzero__` 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_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 __nonzero__ 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

cache

clean

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.

**reentry ()**

**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**

**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
    alias of 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 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.

_Dir__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.

_Dir__resetDuplicate(node)

_Rfindalldirs_key(pathlist)

_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

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

_realt_key(other)

_save_str()

_sconsign

_specific_sources

_srcdir_find_file_key(filename)

tags

_tpath

addRepository(dir)

dependency(depend)
Adds dependencies.

ignore(depend)
Adds dependencies to ignore.

prerequisite(prerequisite)
Adds prerequisites

source(source)
Adds sources.

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

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()
SCons.compat package

```python
ccontsig
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
env_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.

```python
def getRepositories()
    Returns a list of repositories for this directory.

def get_abspath()
    Get the absolute path of the file.

def get_all_rdirs()

def 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.

def get_build_env()
    Fetch the appropriate Environment to build this node.

def get_build_scanner_path(scanner)
    Fetch the appropriate scanner path for this node.

def get_builder(default_builder=None)
    Return the set builder, or a specified default value

def get_cachedir_csig()

def get_contents()
    Return content signatures and names of all our children separated by new-lines. Ensure that the nodes are
    sorted.

def 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.

def get_dir()

def get_env()

def get_env_scanner(env, kw={})

def get_executor(create=1)
    Fetch the action executor for this node. Create one if there isn’t already one, and requested to do so.

def get_found_includes(env, scanner, path)
    Return 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).

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

def get_internal_path()
```
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_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 ()
Return the latest timestamp from among our children

get_tpath ()

gmtime ()

getsize ()

glob (pathname, ondisk=True, source=False, strings=False, exclude=None)
Returns a list of Nodes (or strings) matching a specified pathname pattern.
Pathname patterns follow UNIX shell semantics: * matches any-length strings of any characters, ? matches any
character, and [] can enclose lists or ranges of characters. Matches do not span directory separators.
The matches take into account Repositories, returning local Nodes if a corresponding entry exists in a
Repository (either an in-memory Node or something on disk).
By default, the glob() function matches entries that exist on-disk, in addition to in-memory Nodes. Setting the “ondisk” argument to False (or some other non-true value) causes the glob() function to only match in-memory Nodes. The default behavior is to return both the on-disk and in-memory Nodes.

The “source” argument, when true, specifies that corresponding source Nodes must be returned if you’re globbing in a build directory (initialized with VariantDir()). The default behavior is to return Nodes local to the VariantDir().

The “strings” argument, when true, returns the matches as strings, not Nodes. The strings are path names relative to this directory.

The “exclude” argument, if not None, must be a pattern or a list of patterns following the same UNIX shell semantics. Elements matching a least one pattern of this list will be excluded from the result.

The underlying algorithm is adapted from the glob.glob() function in the Python library (but heavily modified), and uses fnmatch() under the covers.

`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 __nonzero__ 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_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.

`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 `__nonzero__` 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 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.

entry()

tentry_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()
SCons.compat package

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).

srcreal

srcdir_duplicate (name)

srcreal_find_file (filename)

srcreal_list ()

srcnode ()
Dir has a special need for srcnode()…if we have a srcreal attribute set, then that is our srcnode.
SCons.compat package

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

bact
bactsig
bdepends
bdependssigs
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

    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(type, do, ignore)
    Bases: object

    set(list)

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)
SCons.compat package

```
_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 ()

_globl(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_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

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
tables	env	env_set(env, safe=0)
extecutor
executor_cleanup()
Let the executor clean up any cached information.
exists()
Does this node exist?
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
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

getchcachedir_csigg()

get_contents()
Fetch the contents of the entry. Returns the exact binary contents of the file.

getcstag()

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_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 __nonzero__ 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_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 __nonzero__ 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.
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

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()

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.

rexists ()
Does this node exist locally or in a repository?

rfile ()
We’re a generic Entry, but the caller is actually looking for a File at this point, so morph into one.
A Node.FS.Base object's string representation is its path name.

Scan this node's dependents for implicit dependencies.

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 the Node's always_build value.

Set the action executor for this node.

Set the Node's nocache value.

Set the Node's noclean value.

Set the Node's precious value.

Set the Node's precious value.

Set the source code builder for this node.

Set the state of the node.

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

waiting_s_e

wkids

class SCons.Node.FS.EntryProxy (subject)
    Bases: SCons.Util.Proxy

    __EntryProxy__get_abspath ()

    __EntryProxy__get_base_path ()
        Return the file’s directory and file name, with the suffix stripped.

    __EntryProxy__get_dir ()

    __EntryProxy__get_file ()

    __EntryProxy__get_filebase ()

    __EntryProxy__get_posix_path ()
        Return the path with / as the path separator, regardless of platform.

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

    __EntryProxy__get_rsrcnode ()

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

    __EntryProxy__get_srcnode ()

    __EntryProxy__get_suffix ()
_EntryProxy__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>, '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>, 'win32': <function EntryProxy.__get_windows_path>, 'windows': <function EntryProxy.__get_windows_path>}

def 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

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(pathname, ondisk=True, source=True, strings=False, exclude=None, cwd=None)

Globs

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=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.

```python
chdir(dir, change_os_dir=0)
```
Change the current working directory for lookups. If change_os_dir is true, we will also change the “real” cwd to match.

```python
chmod(path, mode)
copy(src, dst)
copy2(src, dst)
exists(path)
get_max_drift()
get_root(drive)
```
Returns the root directory for the specified drive, creating it if necessary.

```python
getcwd()
getmtime(path)
getsize(path)
isdir(path)
isfile(path)
islink(path)
link(src, dst)
listdir(path)
lstat(path)
makedirs(path)
mkdir(path)
open(path)
readlink(file)
rename(old, new)
set_SConstruct_dir(dir)
set_max_drift(max_drift)
stat(path)
```
symlink\((src, dst)\)

unlink\((path)\)

\texttt{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 \texttt{SCons.Node.FS.File} (name, directory, fs)
Bases: \texttt{SCons.Node.FS.Base}
A class for files in a file system.

class \texttt{Attrs}
Bases: \texttt{object}

shared

\texttt{BuildInfo}
alias of \texttt{FileBuildInfo}

\texttt{Decider}(function)

\texttt{Dir}(name, create=True)
Create a directory node named ‘name’ relative to the directory of this file.

\texttt{Dirs}(pathlist)
Create a list of directories relative to the SConscript directory of this file.

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

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

\texttt{GetTag}(key)
Return a user-defined tag.

\texttt{NodeInfo}
alias of \texttt{FileNodeInfo}

\texttt{RDirs}(pathlist)
Search for a list of directories in the Repository list.

\texttt{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.

\texttt{Tag}(key, value)
Add a user-defined tag.

\_File\_dmap\_cache = {}

\_File\_dmap\_sig\_cache = {}

\_Rfindalldirs\_key(pathlist)

\_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) –
      - buildinfo from node being considered (binfo) –
   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)
   Return a list of corresponding csigs from previous build in order of the node/files in children.
   Parameters:
      - self (self) –
      - Dictionary of file -> csig (dmap) –
   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
SCons.compat package

_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
   :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](https://github.com/SCons/scons/issues/2980)
   Parameters:
   - dependency *(self)* –
   - target *(target)* –
   - The NodeInfo object loaded from previous builds .sconsign *(prev_ni)* –
   - Node instance. Check this node for file existence/timestamp *(node)* – if
     specified.
   Returns: Boolean - Indicates if node(File) has changed.

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)

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

to_entries
	env
	env_set(env, safe=0)

evaluator

evaluator_cleanup()
    Let the evaluator clean up any cached information.

exists()
    Does this node exist?

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.

get_abspath()
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

getcachedir_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.

gcachedir_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.

gcontent_hash ()
Compute and return the MD5 hash for this file.

gcontents ()
Fetch the contents of the entry.

gcontents_sig ()
A helper method for getcachedir_bsig.
It computes and returns the signature for this node's contents.

gcsig ()
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

gdir ()

gen ()

genv ()

genv_scanner (env, kw={})

gen_executor (create=1)
Fetch the action executor for this node. Create one if there isn't already one, and requested to do so.

gen_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.

gen_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()
    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_size()

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()
    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()

get_tpath()
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 __nonzero__ 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.

ignore
ignore_set
implicit
implicit_set
includes

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.

```
md5_chunksize = 64
```

```
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 __nonzero__ 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
```

```
ew_binfo()
```

```
new_ninfo()
```

```
ninfo
```

```
ocache
```

```
oclean
```

```
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(*) 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…
released_target_info

remove()
    Remove this file.

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

rentry()

repositories

reset_executor()
    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.
SCons.compat package

```python
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

csrcnode()
    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
```
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.

```python
def 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."""
```

```python
def 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
```

```python
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."""
```

```python
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

```python
with_traceback()
    """Exception.with_traceback(tb) – set self.__traceback__ to tb and return self."""
```
class SCons.Node.FS.FileFinder
    Bases: object

    _find_file_key(filename, paths, verbose=None)

    filedir_lookup(p, fd=None)
        A helper method for find_file() that looks up a directory for a file we’re trying to find. This only
        creates the Dir Node if it exists on-disk, since if the directory doesn’t exist we know we won’t find
        any files in it... :-)
        It would be more compact to just use this as a nested function with a default keyword argument (see
        the commented-out version below), but that doesn’t work unless you have nested scopes, so we define it
        here just so this work under Python 1.5.2.

    find_file(filename, paths, verbose=None)
        Find a node corresponding to either a derived file or a file that exists already.
        Only the first file found is returned, and none is returned if no file is found.
        filename: A filename to find
        paths: A list of directory path nodes
        returns The node created from the found file.

class SCons.Node.FS.FileNodeInfo

    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.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?
SCons.compat package

    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)

    mkdir(path)

    open(path)

    readlink(file)

    rename(old, new)

    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 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 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.

.Dir__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.

.Dir__resetDuplicate(node)

.Rfindalldirs_key(pathlist)

_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_isDerived

_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

_def_path

_local

_lookupDict

_lookup_abs(p, klass, create=1)

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.

_def

_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)
Deaths the number of nodes added to our waiting parents list: 1 if we add a unique waiting parent, 0 if not.
(Notes 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
```

```
built (*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
```

```
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
e

env_set(env, safe=0)

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

exists()
Does this node exists?
**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()**
Get the absolute path of the file.

**get_all_rdirs()**

**get_binfo()**
Fetch a node’s build information.

**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).

```python
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.

```python
get_internal_path()
```

```python
get_labspath()
```

Get the absolute path of the file.

```python
get_ninfo()
```

```python
get_path(dir=None)
```

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

```python
get_path_elements()
```

```python
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.

```python
get_state()
```

```python
get_stored_implicit()
```

Fetch the stored implicit dependencies

```python
get_stored_info()
```

```python
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.

```python
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.

```python
get_suffix()
```

```python
get_target_scanner()
```

```python
get_text_contents()
```

We already emit things in text, so just return the binary version.

```python
get_timestamp()
```

Return the latest timestamp from among our children

```python
get_tpath()
```
getmtime ()

getsize ()

glob (pathname, ondisk=True, source=False, strings=False, exclude=None)
Returns a list of Nodes (or strings) matching a specified pathname pattern.
Pathname patterns follow UNIX shell semantics: * matches any-length strings of any characters, ? matches any
character, and [] can enclose lists or ranges of characters. Matches do not span directory separators.
The matches take into account Repositories, returning local Nodes if a corresponding entry exists in a
Repository (either an in-memory Node or something on disk).
By default, the glob() function matches entries that exist on-disk, in addition to in-memory Nodes. Setting the
“ondisk” argument to False (or some other non-true value) causes the glob() function to only match in-memory
Nodes. The default behavior is to return both the on-disk and in-memory Nodes.
The “source” argument, when true, specifies that corresponding source Nodes must be returned if you’re
globbing in a build directory (initialized with VariantDir()). The default behavior is to return Nodes local to the
VariantDir().
The “strings” argument, when true, returns the matches as strings, not Nodes. The strings are path names
relative to this directory.
The “exclude” argument, if not None, must be a pattern or a list of patterns following the same UNIX shell
semantics. Elements matching a least one pattern of this list will be excluded from the result.
The underlying algorithm is adapted from the glob.glob() function in the Python library (but heavily modified), and
uses fnmatch() under the covers.

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 __nonzero__ 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_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 __nonzero__ 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

cache

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.

**rexists ()**

Does this node exist locally or in a repository?

**rfile ()**

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

Scan this node's dependents for implicit dependencies.

A directory does not get scanned.

Return the .sconsign file info for this directory.

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 the Node's always_build value.

Set the action executor for this node.

Set the Node's always_build value.

Set the Node's nocache value.

Set the Node's noclean value.

Set the Node's precious value.

Set the Node's precious value.

Set the source code builder for this node.

Set the Node's precious value.

Set the Node's precious value.

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

```
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
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)
    Invalidate the memoized values of all Nodes (files or directories) that are associated with the given entries. Has
    been added to clean 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 ()
    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 (list)
SCons.Node.FS._set_duplicate (duplicate)

SCons.Node.Python module

scons.Node.Python

Python nodes.

class SCons.Node.Python.Value (value, built_value=None, name=None)
   A class for Python variables, typically passed on the command line or generated by a script, but not from a file or
   some other source.

class Attrs
   Bases: object

   shared

   BuildInfo
      alias of ValueBuildInfo
Decider (function)

GetTag (key)
   Return a user-defined tag.

NodeInfo
   alias of ValueNodeInfo

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)
SCons.compat package

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

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
```
SCons.compat package

```python
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 contents for signature calculations. :return: bytes

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.
```
SCons.compat package

`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()`
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 __nonzero__ 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-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-scons-script()

Returns true if this node is an scons Script

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 __nonzero__ attributes on instances of our Builder Proxy class(es), generating a bazillion extra calls and slowing things down immensely.

new-binfo()

new-ninfo()

ninfo

no-cache

no-clean

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

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
    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

bact

bactsig

bdepends

bdependsgs

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.


    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.

**Module contents**

SCons.Node

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 NodelInfo instance for this node (signature information that's specific to the type of Node) and direct attributes for the generic build stuff we have to track: sources, explicit dependencies, implicit dependencies, and action information.

bact
SCons.compat package

```
bactsig
bdepends
bdependsigs
bimplicit
bimplicitssigs
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 BuildInfoBase

Decider(function)

GetTag(key)
    Return a user-defined tag.

NodeInfo
    alias of 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_reexists

_func_target_from_source

_get_scanner(env, initial_scanner, root_node_scanner, kw)
```
SCons.compat package

_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

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

evaluator_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()
    Fetch the contents of the entry.

getc sig()

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 __nonzero__ 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_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 __nonzero__ 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
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.

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

__UserList__cast (other)
__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, item)
S.insert(index, value) – insert value before index

pop ([, index]) → item -- remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

remove (item)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

reverse ()
S.reverse() – reverse IN PLACE

sort (*args, **kwds)

class SCons.Node.Walker (node, kids_func=<function get_children>, cycle_func=<function ignore_cycle>, eval_func=<function do_nothing>)
Bases: `object`
An iterator for walking a Node tree.
This is depth-first, children are visited before the parent. The Walker object can be initialized with any node, and returns the next node on the descent with each `get_next()` call. `get the children of a node instead of calling ‘children’`. ‘cycle_func’ is an optional function that will be called when a cycle is detected.
This class does not get caught in node cycles caused, for example, by C header file include loops.

```python
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.
```

```python
is_done()
```

```python
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.
```

```python
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.
```

```python
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.
```

```python
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.reexists_base(node)
```
SCons.Platform package

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)

**Submodules**

**SCons.Platform.aix module**

SCons.Platform.aix
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.cygwin module**

SCons.Platform.cygwin
Platform-specific initialization for Cygwin systems.
There normally shouldn’t be any need to import this module directly. It will usually be imported through the generic SCons.Platform.Platform() selection method.
SCons.Platform.cygwin.generate(env)

**SCons.Platform.darwin module**

SCons.Platform.darwin
Platform-specific initialization for Mac OS X systems.
There normally shouldn’t be any need to import this module directly. It will usually be imported through the generic SCons.Platform.Platform() selection method.
SCons.Platform.darwin.generate(env)

**SCons.Platform.hpux module**

SCons.Platform.hpux
Platform-specific initialization for HP-UX systems.
There normally shouldn’t be any need to import this module directly. It will usually be imported through the generic SCons.Platform.Platform() selection method.
SCons.Platform.hpux.generate(env)

**SCons.Platform.irix module**

SCons.Platform.irix
Platform-specific initialization for SGI IRIX systems.
SCons.Platform package

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

SCons.Platform.irix.generate(env)

**SCons.Platform.mingw module**

SCons.Platform.mingw

Platform-specific initialization for the MinGW system.

**SCons.Platform.os2 module**

SCons.Platform.os2

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**

SCons.Platform.posix

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**

SCons.Platform.sunos

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**

SCons.Platform.virtualenv

Support for 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**

SCons.Platform.win32
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.

```python
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()
    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
SCons platform selection.

This 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.

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

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

```python
class SCons.Platform.PlatformSpec(name, generate)
```
Bases: `object`

```python
class SCons.Platform.TempFileMunge(cmd, cmdstr=None)
```
Bases: `object`

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

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

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

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

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

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

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

```python
_SCons.Scanner package
_SCons.Scanner.C module
```

SCons.Scanner.C

This module implements the dependency scanner for C/C++ code.

```python
_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.).
Return a prototype Scanner instance for scanning source files that use the C pre-processor.

### SCons.Scanner.C.SConsCPPConditionalScanner

**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.

- **_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, **kw)
    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.
SCons.Scanner package

__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.

doiimport(t)
Default handling of a #import line.

doinclude(t)
Default handling of a #include line.

doinclude_next(t)
Default handling of a #include line.

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

doundef(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.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)

SCons.Scanner.D module

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

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

    _recurse_all_nodes(nodes)
SCons.Scanner package

_recursion_no_nodes(nodes)

add_scanner(skey, scanner)

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

find_includes(include, source_dir, path)

find_includes_names(node)

get_skeys(env=None)

path(env, dir=None, target=None, source=None)

scan(node, path=())

select(node)

sort_key(include)

SCons.Scanner.D.DScanner()
    Return a prototype Scanner instance for scanning D source files

SCons.Scanner.D.Dir module

SCons.Scanner.D.DirEntryScanner(**kw)
    Return a prototype Scanner instance for “scanning” directory Nodes for their in-memory entries

SCons.Scanner.D.DirScanner(**kw)
    Return a prototype Scanner instance for scanning directories for on-disk files

SCons.Scanner.D.do_not_scan(k)

SCons.Scanner.D.only_dirs(nodes)

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

SCons.Scanner.D.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

SCons.Scanner.Fortran

This module implements the dependency scanner for Fortran code.

class SCons.Scanner.Fortran.F90Scanner(name, suffixes, path_variable, use_regex, incl_regex, def_regex, *args, **kw)
    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.

_recursion_all_nodes(nodes)
SCons.Scanner package

```python
_defer_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, env, path=())

select(node)

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**

SCons.Scanner.IDL

This module implements the dependency scanner for IDL (Interface Definition Language) files.

SCons.Scanner.IDL.IDLScan()

Return a prototype Scanner instance for scanning IDL source files

**SCons.Scanner.LaTeX module**

SCons.Scanner.LaTeX

This module implements the dependency scanner for LaTeX code.

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

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

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

    The order is:
    ['.eps', '.ps'] for latex ['.png', '.pdf', '.jpg', '.tif']
```
Another difference is that the search path is determined by the type of the file being searched: env['TEXINPUTS'] for "input" and "include" keywords env['TEXINPUTS'] for "includegraphics" keyword env['TEXINPUTS'] for "listinputlisting" keyword env['BIBINPUTS'] for "bibliography" keyword env['BSTINPUTS'] for "bibliographystyle" keyword env['INDEXSTYLE'] for "makeindex" keyword, no scanning support needed just allows user to set it if needed.

FIXME: also look for the class or style in document[\texttt{class|style}]{} FIXME: also look for the argument of \texttt{bibliographystyle}{},

\begin{verbatim}
  _latex_names(include_type, filename)
  _recurse_all_nodes(nodes)
  _recurse_no_nodes(nodes)
  add_scanner(skey, scanner)
  add_skey(skey)
    Add a skey to the list of skeys
  canonical_text(text)
    Standardize an input TeX-file contents.
    Currently:
      \begin{itemize}
        \item removes comments, unwrapping comment-wrapped lines.
      \end{itemize}
  env_variables = ['TEXINPUTS', 'BIBINPUTS', 'BSTINPUTS', 'INDEXSTYLE']
  find_include(include, source_dir, path)
  get_skeys(env=None)
  keyword_paths = {'addbibresource': 'BIBINPUTS', 'addglobalbib': 'BIBINPUTS', 'addsectionbib': 'BIBINPUTS', 'bibliography': 'BIBINPUTS', 'bibliographystyle': 'BSTINPUTS', 'include': 'TEXINPUTS', 'includegraphics': 'TEXINPUTS', 'input': 'TEXINPUTS', 'lstinputlisting': 'TEXINPUTS', 'makeindex': 'INDEXSTYLE', 'usepackage': 'TEXINPUTS'}
  path(env, dir=None, target=None, source=None)
  scan(node, subdir='.'
    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']
\end{verbatim}

SCons.Scanner.LaTeX._null
alias of SCons.Scanner.LaTeX._Null

SCons.Scanner package
SCons.Scanner package

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

**SCons.Scanner.Prog module**

SCons.Scanner.Prog.ProgramScanner(**kw)
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=())
This scanner 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**

SCons.Scanner.RC
This module implements the 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**

SCons.Scanner.SWIG
This module implements the dependency scanner for SWIG code.

SCons.Scanner.SWIG.SWIGScanner()

**Module contents**

SCons.Scanner
The Scanner package for the SCons software construction utility.

```python
class SCons.Scanner.Base (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
    The base class for dependency scanners. This implements straightforward, single-pass scanning of a single file.

    _recurse_all_nodes(nodes)

    _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.Classic (name, suffixes, path_variable, regex, *args, **kw)
    Bases: SCons.Scanner.Current
```

311
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_key() methods), the regular expression passed to the constructor must return the name of the include file in group 0.

```python
_recurse_all_nodes(nodes)
_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)
sort_key(include)
```

class SCons.Scanner.ClassicCPP(name, suffixes, path_variable, regex, *args, **kw)
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
_recurse_all_nodes(nodes)
_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)
sort_key(include)
```

class SCons.Scanner.Current(*args, **kw)
Bases: SCons.Scanner.Base
SCons.Scanner package

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).

_recurse_all_nodes(nodes)
_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.FindPathDirs(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.

SCons.Scanner.Scanner(function, *args, **kw)
  Public interface factory function for creating different types of Scanners based on the different types of “functions” that may be supplied.
  TODO: Deprecate this some day. We’ve moved the functionality inside the Base 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.Selector(dict, *args, **kw)
  Bases: SCons.Scanner.Base
  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 Base 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.)

_recurse_all_nodes(nodes)
_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._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)

    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.

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:'
SCons.Script package

```python
use_rawinput = 1
SCons.Script.Interactive.interact(fs, parser, options, targets, target_top)
```

## SCons.Script.Main module

SCons.Script

This file implements 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.

    __abc_impl = __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 (method, node, description='node')

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

```python
__abc_impl = <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()

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(method, node, description='node')

class SCons.Script.Main.CountStats

do_append(label)

do_nothing(*args, **kw)

do_print()

enable(outfp)

class SCons.Script.Main.FakeOptionParser
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.QuestionTask(tm, targets, top, node)
Bases: SCons.Taskmaster.AlwaysTask
An SCons task for the -q (question) option.

_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:

```python
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()**

```
method, 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()**

```
(node)
```
SCons.Script package

```python
get_derived_children(node)
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=None)
    Load all of the predefined site_scons dir. Order is significant; we load them in order from most generic (machine-wide) to most specific (topdir). The verbose argument is only for testing.
SCons.Script.Main._load_site_scons_dir(topdir, site_dir_name=None)
    Load the site directory under topdir.
    If a site dir name is supplied use it, else use default “site_scons” Prepend site dir to sys.path. If a “site_tools” subdir exists, prepend to toolpath. Import “site_init.py” from site dir if it exists.
SCons.Script.Main._main(parser)
SCons.Script.Main._scons_internal_error()
    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.fetch_win32_parallel_msg()
SCons.Script.Main.find_deepest_user_frame(tb)
    Find the deepest stack frame that is not part of SCons.
    Input is a “pre-processed” stack trace in the form returned by traceback.extract_tb() or traceback.extract_stack()
SCons.Script.Main.main()
SCons.Script.Main.path_string(label, module)
SCons.Script.Main.python_version_deprecated(version=sys.version_info(major=3, minor=7, micro=8, releaselevel='final', serial=0))
SCons.Script.Main.python_version_string()
SCons.Script.Main.python_version_unsupported(version=sys.version_info(major=3, minor=7, micro=8, 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)
SCons.Script package

Returns an options parser object initialized with the standard SCons options.

class SCons.Script.SConsOptions.SConsIndentedHelpFormatter (indent_increment=2,
max_help_position=24, width=None, short_first=1)
    Bases: optparseIndentedHelpFormatter

NO_DEFAULT_VALUE = 'none'

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

dedent ()

expand_default (option)

format_description (description)

format_epilog (epilog)

format_heading (heading)
    This translates any heading of “options” or “Options” into “SCons Options.” Unfortunately, we have to do this
    here, because those titles are hard-coded in the optparse calls.

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

format_option_strings (option)
    Return a comma-separated list of option strings & metavariables.

format_usage (usage)

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 = [<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>}

_check_action ()
_check_callback ()
_check_choice ()
_check_const ()
_check_dest ()
_check_nargs ()
_check_nargsOptional ()
_check_opt_strings (opts)
_check_type ()
_set_attrs (attrs)
_set_opt_strings (opts)
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 ()
SCons.Script package

```python
/share_option_mappings(parser)

add_option(Option)
    add_option(opt_str, ..., kwarg=val, ...)

add_options(option_list)

destroy()
    see OptionParser.destroy().

format_description(formatter)

format_help(formatter)
    Format an option group's help text, outdenting the title 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)
```

```python
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, 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.

_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 a SetOption() 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, ...)

add_option_group(*args, **kwargs)

add_options(option_list)

check_values(values : Values, args : [string])
-> (values : Values, args : [string])
Check that the supplied option values and leftover arguments are valid. Returns the option values and leftover arguments (possibly adjusted, possibly completely new – whatever you like). Default implementation just returns the passed-in values; subclasses may override as desired.

destroy()
Declare that you are done with this OptionParser. This cleans up reference cycles so the OptionParser (and all objects referenced by it) can be garbage-collected promptly. After calling destroy(), the OptionParser is unusable.

disable_interspersed_args()
Set parsing to stop on the first non-option. Use this if you have a command processor which runs another command that has options of its own and you want to make sure these options don’t get confused.

enable_interspersed_args()
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior. See also disable_interspersed_args() and the class documentation description of the attribute allow_interspersed_args.

error(msg : string)
Print a usage message incorporating ‘msg’ to stderr and exit. If you override this in a subclass, it should not return – it should either exit or raise an exception.

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(opt_str)

get_option_group(opt_str)

get_prog_name()

get_usage()

get_version()

has_option(opt_str)

parse_args(args=None, values=None)

    parse_args(args : [string] = sys.argv[1:], values : Values = None)
    -> (values : Values, args : [string])

Parse the command-line options found in ‘args’ (default: sys.argv[1:]) Any errors result in a call to ‘error()’, which by default prints the usage message to stderr and calls sys.exit() with an error message. On success returns a pair (values, args) where ‘values’ is a Values instance (with all your option values) and ‘args’ is the list of arguments left over after parsing options.

preserve_unknown_options = False

print_help(file : file = stdout)

Print an extended help message, listing all options and any help text provided with them, to ‘file’ (default stdout).

print_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.

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).
SCons.Script package

```python
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 or not 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).

    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.

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

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

    ensure_value(attr, value)

    read_file(filename, mode='careful')

    read_module(modname, mode='careful')

    set_option(name, value)
    Sets an option from an SConscript file.

    settable = ['clean', 'diskcheck', 'duplicate', 'help', 'implicit_cache', 'max_drift', 'md5_chunksize', 'no_exec', 'num_jobs', 'random', 'stack_size', 'warn', 'silent', 'no_progress']

SCons.Script.SConsOptions.diskcheck_convert(value)
```
This module defines the Python API provided to SConscript and SConstruct files.

```python
SCons.Script.SConscript().BuildDefaultGlobals()

Create a dictionary containing all the default globals for SConstruct and SConscript files.
```

```python
SCons.Script.SConscript().Configure(*args, **kw)
```

```python
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).
```

```python
class SCons.Script.SConscript.Frame(fs, exports, sconscript):
    Bases: object
    A frame on the SConstruct/SConscript call stack
```

```python
SCons.Script.SConscript.Return(*vars, **kw)
```

```python
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.
```

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

```python
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.
```

```python
AddPostAction(files, action)
AddPreAction(files, action)
```

```python
Alias(target, source=[], action=None, **kw)
```

```python
AlwaysBuild(*targets)
```

```python
Append(**kw)
    Append values to existing construction variables in an Environment.
```

```python
AppendENVPath(name, newpath, envname='ENV', sep=':', delete_existing=0)
    Append path elements to the path 'name' in the 'ENV' 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 0, a newpath which is already in the path will not be moved to the end (it will be left where it
    is).
```

```python
AppendUnique(delete_existing=0, **kw)
    Append values to existing construction variables in an Environment, if they're not already there. If
    delete_existing is 1, removes existing values first, so values move to end.
```

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

```python
CacheDir(path)
```
SCons.Script package

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’s depend on ‘dependency’.

Detect(progs)
Return the first available program in progs.

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

Returns str: first name from progs that can be found.

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 (optional) – specify the format to serialize to. “pretty” generates a pretty-printed string, “json” a JSON-formatted string. (Default value = None, equivalent to “pretty”)

EnsurePythonVersion(major, minor)
Exit abnormally if the Python version is not late enough.

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

Exit(value=0)
SCons.Script package

```
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='. ')
returns a list of all source files.

Flatten(sequence)

GetBuildPath(files)

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=1, dict=None)
Merge the dict in args into the construction variables of this env, or the passed-in dict. If args is not a dict, it is converted into a dict using ParseFlags. If unique is not set, the flags are appended rather than merged.

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=1)
```
Use the specified function to parse the output of the command in order to modify the current environment. The 'command' can be a string or a list of strings representing a command and its arguments. 'Function' is an optional argument that takes the environment, the output of the command, and the unique flag. If no function is specified, MergeFlags, which treats the output as the result of a typical 'X-config' command (i.e. gtk-config), will merge the output into the appropriate variables.

**ParseDepends (filename, must_exist=None, only_one=0)**
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**)
Parse the set of flags and return a dict with the flags placed in the appropriate entry. The flags are treated as a typical set of command-line flags for a GNU-like toolchain and used to populate the entries in the dict immediately below. If one of the flag 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 existing construction variables in an Environment.

**PrependENVPath (name, newpath, envname='ENV', sep=':', delete_existing=1)**
Prepend path elements to the path ‘name’ in the ‘ENV’ 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 0, a newpath which is already in the path will not be moved to the front (it will be left where it is).

**PrependUnique (delete_existing=0, **kw**)
Prepend values to existing construction variables in an Environment, if they're not already there. If delete_existing is 1, 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.

**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.
SCons.Script package

**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.

SConscriptChdir(flag)

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, **kw)

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

VariantDir(variant_dir, src_dir, duplicate=1)

WhereIs(prog, path=None, pathext=None, reject=[])  
Find prog in the path.

_canonicalize(path)

_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)
SCons.Script package

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

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

_copy2_from_cache(src, dst)

_copy_from_cache(src, dst)

_exceeds_version(major, minor, v_major, v_minor)
    Return 1 if 'major' and 'minor' are greater than the version in 'v_major' and 'v_minor', and 0 otherwise.

_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.

_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(dict)
    Update an environment's values directly, bypassing 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)

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()

has_key(key)
    Emulates the has_key() method of dictionaries.

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).

subst (string, raw=0, target=None, source=None, conv=None, executor=None)
Recursively interpolates construction variables from the Environment into the specified string, returning the expanded result. Construction variables are specified by a $ prefix in the string and begin with an initial underscore or alphabetic character followed by any number of underscores or alphanumeric characters. The construction variable names may be surrounded by curly braces to separate the name from trailing characters.

subst_kw (kw, raw=0, target=None, source=None)

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

subst_path (path, target=None, source=None)
Substitute a path list, 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)
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.

exception SCons.Script.SConscript.SConscriptReturn
Bases: Exception

args

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

SCons.Script.SConscript.SConscript_exception (file=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>)
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. On first warning, print a deprecation message.
**Parameters:**
- `f` *(str)* – path of missing configuration file
- `must_exist` *(bool)* – raise exception if file does not exist

**Raises:**
- `UserError` if 'must_exist' is True or if global `SCons.Script._no_missing_sconsript` is True.

### Module contents

**SCons.Script**

This file implements 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.

```python
SCons.Script.HelpFunction(text, append=False)
```

**class** `SCons.Script.TargetList` *(initlist=None)*

**Bases:** `collections.UserList`

- `__cast` *(other)*
- `__impl` = `<_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** *(i[, 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._AddTargets (tlist)
SCons.Script._Get_Default_Targets (d, fs)
SCons.Script._Set_Default_Targets (env, tlist)
SCons.Script._Set_Default_TargetsHAS_Been_Called (d, fs)
SCons.Script._Set_Default_TargetsHAS_Not_Been_Called (d, fs)
SCons.Script.set_missing_sconscript_error (flag=1)
    Set behavior on missing file in SConscript() call. Returns previous value

SCons.Tool package

Module contents

SCons.Tool
SCons tool selection.
This 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 definition.

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.CreateLibSymlinks (env, symlinks)
    Physically creates symlinks. The symlinks argument must be a list in form [(link, linktarget), …], where link and
linktarget are SCons nodes.

SCons.Tool.EmitLibSymlinks (env, symlinks, libnode, **kw)
    Used by emitters to handle (shared/versioned) library symlinks

SCons.Tool.FindAllTools (tools, env)
SCons.Tool.FindTool (tools, env)
SCons.Tool.Initializers (env)
SCons.Tool.LibSymlinksActionFunction (target, source, env)
SCons.Tool.LibSymlinksStrFun (target, source, env, *args)
SCons.Tool.StringizeLibSymlinks (symlinks)
    Converts list with pairs of nodes to list with pairs of node paths (strings). Used mainly for debugging.

class SCons.Tool.Tool (name, toolpath=None, **kw)
    Bases: object

    _load_dotted_module_py2 (short_name, full_name, searchpaths=None)
    _tool_module ()
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.

get_builder (env)
Returns the appropriate real Builder for this method name after having the associated ToolInitializer object apply
the appropriate Tool module.

class SCons.Tool._ImpLibInfoSupport
Bases: object

get_lib_noversionsymlinks (env, *args, **kw)
get_lib_prefix (env, *args, **kw)
get_lib_suffix (env, *args, **kw)
get_lib_version (env, *args, **kw)

libtype

class SCons.Tool._LdModInfoSupport
Bases: object

get_lib_noversionsymlinks (env, *args, **kw)
get_lib_prefix (env, *args, **kw)
get_lib_suffix (env, *args, **kw)
get_lib_version (env, *args, **kw)

libtype

class SCons.Tool._LibInfoGeneratorBase (libtype, infoname)
Bases: object
Generator base class for library-related info such as suffixes for versioned libraries, symlink maps, sonames etc. It
handles commonities of SharedLibrary and LoadableModule

__support_classes = {'ImpLib': <class 'SCons.Tool._ImpLibInfoSupport'>, 'LdMod': <class 'SCons.Tool._LdModInfoSupport'>, 'ShLib': <class 'SCons.Tool._ShLibInfoSupport'>}
SCons.Tool package

generate_versioned_lib_info(env, args, result=None, **kw)
get_lib_noversionsymlinks(env, *args, **kw)
get_lib_prefix(env, *args, **kw)
get_lib_suffix(env, *args, **kw)
get_lib_version(env, *args, **kw)
get_versioned_lib_info_generator(**kw)

@property libtype
class SCons.Tool._LibNameGenerator(libtype)
Bases: SCons.Tool._LibInfoGeneratorBase
Generates “unmangled” library name from a library file node. Generally, it’s thought to revert modifications done by
prefix/suffix generators (_LibPrefixGenerator/_LibSuffixGenerator) used by a library builder. For example, on gnulink the suffix generator used by
SharedLibrary builder appends $SHLIBVERSION to $SHLIBSUFFIX producing node name which ends
with “$SHLIBSUFFIX.$SHLIBVERSION”. Correspondingly, the implementation of _LibNameGenerator replaces
“$SHLIBSUFFIX.$SHLIBVERSION” with “$SHLIBSUFFIX” in the node’s basename. So that, if $SHLIBSUFFIX is “.so”, $SHLIBVERSION is “0.1.2” and the node path is “/foo/bar/libfoo.so.0.1.2”, the _LibNameGenerator shall
return “libfoo.so”. Other link tools may implement it’s own way of library name unmangling.

_support_classes = {'ImpLib': <class 'SCons.Tool._ImpLibInfoSupport'>, 'LdMod': <class 'SCons.Tool._LdModInfoSupport'>, 'ShLib': <class 'SCons.Tool._ShLibInfoSupport'>}

generate_versioned_lib_info(env, args, result=None, **kw)
get_lib_noversionsymlinks(env, *args, **kw)
get_lib_prefix(env, *args, **kw)
get_lib_suffix(env, *args, **kw)
get_lib_version(env, *args, **kw)
get_versioned_lib_info_generator(**kw)

property libtype
class SCons.Tool._LibPrefixGenerator(libtype)
Bases: SCons.Tool._LibInfoGeneratorBase
Library prefix generator, used as target_prefix in SharedLibrary and LoadableModule builders

_support_classes = {'ImpLib': <class 'SCons.Tool._ImpLibInfoSupport'>, 'LdMod': <class 'SCons.Tool._LdModInfoSupport'>, 'ShLib': <class 'SCons.Tool._ShLibInfoSupport'>}

generate_versioned_lib_info(env, args, result=None, **kw)
get_lib_noversionsymlinks(env, *args, **kw)
get_lib_prefix(env, *args, **kw)
get_lib_suffix(env, *args, **kw)
get_lib_version(env, *args, **kw)
get_versioned_lib_info_generator(**kw)
property libtype

class SCons.Tool._LibSonameGenerator (libtype)
Bases: SCons.Tool._LibInfoGeneratorBase
Library soname generator. Returns library soname (e.g. libfoo.so.0) for a given node (e.g. /foo/bar/libfoo.so.0.1.2)

_support_classes = {'ImpLib': <class 'SCons.Tool._ImpLibInfoSupport'>, 'LdMod': <class 'SCons.Tool._LdModInfoSupport'>, 'ShLib': <class 'SCons.Tool._ShLibInfoSupport'>}

generate_versioned_lib_info (env, args, result=None, **kw)
get_lib_noversionsymlinks (env, *args, **kw)
get_lib_prefix (env, *args, **kw)
get_lib_suffix (env, *args, **kw)
get_lib_version (env, *args, **kw)
get_versioned_lib_info_generator (**kw)

property libtype

class SCons.Tool._LibSuffixGenerator (libtype)
Bases: SCons.Tool._LibInfoGeneratorBase
Library suffix generator, used as target_suffix in SharedLibrary and LoadableModule builders

_support_classes = {'ImpLib': <class 'SCons.Tool._ImpLibInfoSupport'>, 'LdMod': <class 'SCons.Tool._LdModInfoSupport'>, 'ShLib': <class 'SCons.Tool._ShLibInfoSupport'>}

generate_versioned_lib_info (env, args, result=None, **kw)
get_lib_noversionsymlinks (env, *args, **kw)
get_lib_prefix (env, *args, **kw)
get_lib_suffix (env, *args, **kw)
get_lib_version (env, *args, **kw)
get_versioned_lib_info_generator (**kw)

property libtype

class SCons.Tool._LibSymlinkGenerator (libtype)
Bases: SCons.Tool._LibInfoGeneratorBase
Library symlink map generator. It generates a list of symlinks that should be created by SharedLibrary or LoadableModule builders

_support_classes = {'ImpLib': <class 'SCons.Tool._ImpLibInfoSupport'>, 'LdMod': <class 'SCons.Tool._LdModInfoSupport'>, 'ShLib': <class 'SCons.Tool._ShLibInfoSupport'>}

generate_versioned_lib_info (env, args, result=None, **kw)
get_lib_noversionsymlinks (env, *args, **kw)
get_lib_prefix (env, *args, **kw)
get_lib_suffix (env, *args, **kw)
get_lib_version (env, *args, **kw)
SCons.Tool package

get_versioned_lib_info_generator(**kw)

property libtype

class SCons.Tool._ShLibInfoSupport
    Bases: object

    get_lib_noversionsymlinks (env, *args, **kw)
    get_lib_prefix (env, *args, **kw)
    get_lib_suffix (env, *args, **kw)
    get_lib_version (env, *args, **kw)

    property libtype

SCons.Tool._call_env_subst (env, string, *args, **kw)

SCons.Tool._call_linker_cb (env, callback, args, result=None)
    Returns the result of env['LINKCALLBACKS'][callback](args) if env['LINKCALLBACKS'] is a dictionary and
    env['LINKCALLBACKS'][callback] is callable. If these conditions are not met, return the value provided as the
    *result argument. This function is mainly used for generating library info such as versioned suffixes, symlink maps,
    sonames etc. by delegating the core job to callbacks configured by current linker tool.

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)
    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.

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)
    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.

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)
    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.

SCons.Tool.tool_list (platform, env)
SCons.Variables package

Submodules

SCons.Variables.BoolVariable module

SCons.Variables.BoolVariable

This file defines the option type for SCons implementing true/false values.

Usage example:

```python
opts = Variables()
opts.Add(BoolVariable('embedded', 'build for an embedded system', 0))
...
if env['embedded'] == 1:
...
```


The input parameters describe a boolean option, thus they are returned with the correct converter and validator appended. The ‘help’ text will be appended by ‘(yes|no) to show the valid valued. The result is usable for input to opts.Add().

SCons.Variables.EnumVariable module

SCons.Variables.EnumVariable

This file defines the option type for SCons allowing only specified input-values.

Usage example:

```python
opts = Variables()
opts.Add(EnumVariable('debug', 'debug output and symbols', '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)

The input parameters describe an option with only certain values allowed. They are returned with an appropriate converter and validator appended. The result is usable for input to Variables.Add().

‘key’ and ‘default’ are the values to be passed on to Variables.Add().

‘help’ will be appended by the allowed values automatically.

‘allowed_values’ is a list of strings, which are allowed as values for this option.

The ‘map’-dictionary may be used for converting the input value into canonical values (e.g. for aliases).

‘ignorecase’ defines the behaviour of the validator:

- If ignorecase == 0, the validator/converter are case-sensitive.
- If ignorecase == 1, the validator/converter are case-insensitive.
- If ignorecase == 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

SCons.Variables.ListVariable

This file defines the option type for SCons implementing ‘lists’.

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.
SCons.Variables package

Usage example:

```python
list_of_libs = Split('x11 gl qt ical')

opts = Variables()
opts.Add(ListVariable('shared',
    'libraries to build as shared libraries',
    '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={})

The input parameters describe a ‘package list’ option, thus they are returned with the correct converter and validator appended. The result is usable for input to opts.Add().

A ‘package list’ option may either be ‘all’, ‘none’ or a list of package names (separated by space).

SCons.Variables.PackageVariable module

SCons.Variables.PackageVariable

This file defines the option type for SCons implementing ‘package activation’.

To be used whenever a ‘package’ may be enabled/disabled and the package path may be specified.

Usage example:

**Examples:**

x11=no (disables X11 support) x11=yes (will search for the package installation dir) x11=/usr/local/X11 (will check this path for existence)

To replace autoconf’s –with-xxx=yyy

```python
opts = Variables()
opts.Add(PackageVariable('x11',
    'use X11 installed here (yes = search some places',
    'yes'))
... if env['x11'] == True:
    dir = ... search X11 in some standard places ...
env['x11'] = dir
if env['x11']:
    ... build with x11 ...
```

SCons.Variables.PackageVariable.PackageVariable (key, help, default, searchfunc=None)

The input parameters describe a ‘package list’ option, thus they are returned with the correct converter and validator appended. The result is usable for input to opts.Add().

A ‘package list’ option may either be ‘all’, ‘none’ or a list of package names (separated by space).

SCons.Variables.PathVariable module

SCons.Variables.PathVariable

This file defines an option type for SCons implementing path settings.

To be used whenever a user-specified path override should be allowed.

**Arguments to PathVariable are:**

- option-name = name of this option on the command line (e.g. “prefix”)
- option-help = help string for option
- option-dflt = default value for this option
- validator = [optional] validator for option value. Predefined validators 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). The key is the name of the option, the val is the path specified for the option, and the env is the env to which the Options have been added.

Usage example:

```
Examples:
    prefix=/usr/local

    opts = Variables()
    opts = Variables()
    opts.Add(PathVariable('qtdir',
        'where the root of Qt is installed',
        qtdir, PathIsDir))
    opts.Add(PathVariable('qt_includes',
        'where the Qt includes are installed',
        '${qtdir}/includes', PathIsDirCreate))
    opts.Add(PathVariable('qt_libraries',
        'where the Qt library is installed',
        '${qtdir}/lib')
```

Module contents

SCons.Variables

This file defines the Variables class that is used to add 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)

    Add (key, help='', default=None, validator=None, converter=None, **kw)
    Add an option.
    
    Parameters:
    * key – the name of the variable, or a list or tuple of arguments
    * help – optional help text for the options (Default value = '')
    * default – optional default value for option (Default value = None)
    * validator – optional function called to validate the option’s value (Default value = None)
    * converter – optional function to be called to convert the option’s value before putting it in the environment. (Default value = None)
    * **kw – keyword args, unused.

    AddVariables (*optlist)
    Add 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=[])

GenerateHelpText (env, sort=None)
    Generate the help text for the options.
    env - an environment that is used to get the current values
        of the options.
    cmp - Either a function as follows: The specific sort function should take two arguments and return -1, 0
        or 1
        or a boolean to indicate if it should be sorted.

Save (filename, env)
    Saves all the options in the given file. This file can then be used to load the options next run. This can be used to
    create an option cache file.
    filename - Name of the file to save into env - the environment get the option values from

UnknownVariables ()
    Returns any options in the specified arguments lists that were not known, declared options in this object.

Update (env, args=None)
    Update an environment with the option variables.
    env - the environment to update.

_do_add (key, help='', default=None, validator=None, converter=None)

format = '%s: %s default: %s actual: %s'

format_ = '%s: %s default: %s actual: %s aliases: %s'

instance = None

keys ()
    Returns the keywords for the options
add_to_waiting_s_e() (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)

(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)

AddBatchExecutor() (in module SCons.Executor)
AddMethod() (in module SCons.Util)
(SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Environment.SubstitutionEnvironment method)
(SCons.Script.SConscript.SConsEnvironment method)

AddOption() (in module SCons.Script.Main)
AddPathIfNotExists() (in module SCons.Util)
AddPostAction() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Script.SConscript.SConsEnvironment method)

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

addRepository() (SCons.Node.FS.Dir method)
(SCons.Node.FS.RootDir method)

(SCons.Node.FS.Base attribute)
(SCons.Node.FS.Dir attribute)
(SCons.Node.FS.Entry attribute)
(SCons.Node.FS.File attribute)
(SCons.Node.FS.RootDir attribute)
(SCons.Node.Node attribute)

ALWAYS_TYPED_ACTIONS
(SCons.Script.SConsOptions.SConsOption attribute)
AlwaysBuild() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Script.SConscript.SConsEnvironment method)

AlwaysTask (class in SCons.Taskmaster)

Annotate() (in module SCons.Node)
annotate() (in module SCons.Script.SConscript)
append() (SCons.Builder.ListEmitter method)
Append() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
append() (SCons.Executor.TSList method)
Append() (SCons.Script.SConscript.SConsEnvironment method)
append() (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)
AppendENVPath() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Script.SConscript.SConsEnvironment method)

AppendLIBS() (SCons.SConf.CheckContext method)

AppendPath() (in module SCons.Util)
AppendUnique() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Script.SConscript.SConsEnvironment method)
apply_tools() (in module SCons.Environment)
(SCons.Tool.ToolInitializer method)
ArchDefinition (class in SCons.Platform.win32)
arg2nodes() (SCons.Platform.win32)
(SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Environment.SubstitutionEnvironment method)
(SCons.Script.SConscript.SConsEnvironment method)

args (SCons.Errors.BuildError attribute)
(SCons.Errors.ExplicitExit attribute)
(SCons.Errors.InternalError attribute)
(SCons.Errors.MSVCError attribute)
(SCons.Errors.SConsEnvironmentError attribute)
(SCons.Errors.StopError attribute)
(SCons.Errors.UserError attribute)
(SCons.Node.FS.EntryProxyAttributeError attribute)
(SCons.Node.FS.FileBuildInfoFileToCsigMappingError attribute)
(SCons.SConf.ConfigureCacheError attribute)
(SCons.SConf.ConfigureDryRunError attribute)
(SCons.SConf.SConfError attribute)
(SCons.SConf.SConfWarning attribute)
(SCons.Script.Main.SConsPrintHelpException attribute)
(SCons.Script.SConscript.SConscriptReturn attribute)
(SCons.Util._NoError attribute)
(SCons.Util.PlainWindowsError attribute)
(SCons.Warnings.CacheVersionWarning attribute)
(SCons.Warnings.CacheWriteErrorWarning attribute)
(SCons.Warnings.CorrectSConsignWarning attribute)
(SCons.Warnings.DependencyWarning attribute)
(SCons.Warnings.DeprecatedDebugOptionsWarning attribute)
(SCons.Warnings.DeprecatedMissingSConscriptWarning attribute)
(SCons.Warnings.DeprecatedOptionsWarning attribute)
(SCons.Warnings.DeprecatedSourceCodeWarning attribute)
(SCons.Warnings.DeprecatedWarning attribute)
(SCons.Warnings.DevelopmentVersionWarning attribute)
(SCons.Warnings.DuplicateEnvironmentWarning attribute)
(SCons.Warnings.FortranCxxMixWarning attribute)
(SCons.Warnings.FutureDeprecatedWarning attribute)
(SCons.Warnings.FutureReservedVariableWarning attribute)
(SCons.Warnings.LinkWarning attribute)
(SCons.Warnings.MandatoryDeprecatedWarning attribute)
(SCons.Warnings.MisleadingKeywordsWarning attribute)
(SCons.Warnings.MissingSConscriptWarning attribute)
(SCons.Warnings.NoObjectCountWarning attribute)
do_else() (SCons.cpp.DumbPreProcessor method)
(SCons.cpp.PreProcessor method)
(SCons.Scanner.C.SConsCPPConditionalScanner method)
(SCons.Scanner.C.SConsCPPScanner method)

do endif() (SCons.cpp.DumbPreProcessor method)
(SCons.cpp.PreProcessor method)
(SCons.Scanner.C.SConsCPPConditionalScanner method)
(SCons.Scanner.C.SConsCPPScanner method)

do_eof() (SCons.cpp.DumbPreProcessor method)
(SCons.cpp.PreProcessor method)
(SCons.Scanner.C.SConsCPPConditionalScanner method)
(SCons.Scanner.C.SConsCPPScanner method)

do_if() (SCons.cpp.DumbPreProcessor method)
(SCons.cpp.PreProcessor method)
(SCons.Scanner.C.SConsCPPConditionalScanner method)
(SCons.Scanner.C.SConsCPPScanner method)

do ifndef() (SCons.cpp.DumbPreProcessor method)
(SCons.cpp.PreProcessor method)
(SCons.Scanner.C.SConsCPPConditionalScanner method)
(SCons.Scanner.C.SConsCPPScanner method)

do_import() (SCons.cpp.DumbPreProcessor method)
(SCons.cpp.PreProcessor method)
(SCons.Scanner.C.SConsCPPConditionalScanner method)
(SCons.Scanner.C.SConsCPPScanner method)

do_include() (SCons.cpp.DumbPreProcessor method)
(SCons.cpp.PreProcessor method)
(SCons.Scanner.C.SConsCPPConditionalScanner method)
(SCons.Scanner.C.SConsCPPScanner method)

do_include_next() (SCons.cpp.DumbPreProcessor method)
(SCons.cpp.PreProcessor method)
(SCons.Scanner.C.SConsCPPConditionalScanner method)
(SCons.Scanner.C.SConsCPPScanner method)

do_not_scan() (in module SCons.Scanner.Dir)

do_not_set_entry() (SCons.SConsign.Dir method)
(SCons.SConsign.DB method)
(SCons.SConsign.Dir method)
(SCons.SConsign.DirFile method)

do_not_store_info() (SCons.SConsign.Dir method)
(SCons.SConsign.DB method)
(SCons.SConsign.Dir method)
(SCons.SConsign.DirFile method)

do_nothing() (in module SCons.Node)
(SCons.cpp.DumbPreProcessor method)
(SCons.cpp.PreProcessor method)
(SCons.Scanner.C.SConsCPPConditionalScanner method)
(SCons.Scanner.C.SConsCPPScanner method)
(SCons.Script.Main.CountStats method)
(SCons.Script.Main.MemStats method)
(SCons.Script.Main.Stats method)

do_nothing_node() (in module SCons.Node)

do_print() (SCons.Script.Main.CountStats method)
(SCons.Script.Main.MemStats method)

do_shell() (SCons.Script.Interactive.SConsInteractiveCmd method)

do_UNDEF() (SCons.cpp.DumbPreProcessor method)
(SCons.cpp.PreProcessor method)
(SCons.Scanner.C.SConsCPPConditionalScanner method)
(SCons.Scanner.C.SConsCPPScanner method)

do_version() (SCons.Script.Interactive.SConsInteractiveCmd method)

doc_header (SCons.Script.Interactive.SConsInteractiveCmd attribute)
dump_caller_counts() (in module SCons.Debug)
dump_stats() (in module SCons.Taskmaster)
dumpLoggedInstances() (in module SCons.Debug)
duplicate (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)
Dump() (in module SCons.Memoize)
    (SCons.Environment.Base method)
    (SCons.Environment.OverrideEnvironment method)
    (SCons.Script.SConscript.SConsEnvironment method)
duplicate() (in module SCons.Scanner.D)
DumbPreProcessor (class in SCons.cpp)
dump() (in module SCons.Memoize)
    (SCons.Environment.Base method)
    (SCons.Environment.OverrideEnvironment method)
    (SCons.Script.SConscript.SConsEnvironment method)
dump_caller_counts() (in module SCons.Debug)
dump_stats() (in module SCons.Taskmaster)
dumpLoggedInstances() (in module SCons.Debug)
duplicate (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)
DuplicateEnvironmentWarning

E

EmitLibSymlinks() (in module SCons.Tool)
EmitterProxy (class in SCons.Builder)
emptyline()
    (SCons.Script.Interactive.SConsInteractiveCmd method)
enable() (SCons.Script.Main.CountStats method)
    (SCons.Script.Main.MemStats method)
    (SCons.Script.Main.Stats method)
enable_interspersed_args()
    (SCons.Script.SConsOptions.SConsOptionParser method)
EnableMemoization() (in module SCons.Memoize)
enableWarningClass() (in module SCons.Warnings)
encode() (SCons.Subst.CmdStringHolder method)
endswith() (SCons.Subst.CmdStringHolder method)
ensure_value()
    (SCons.Script.SConsOptions.SConsValues method)
EnsurePythonVersion()
    (SCons.Script.SConscript.SConsEnvironment method)
EnsureSConsVersion()
    (SCons.Script.SConscript.SConsEnvironment method)
entries (SCons.Node.FS.Dir attribute)
    (SCons.Node.FS.Entry attribute)
    (SCons.Node.FS.File attribute)
executed_without_callbacks()
(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)

Executor (class in SCons.Executor)
executor (SCons.Errors.BuildError 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)

(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)

exists_always() (in module SCons.Node)
exists_base() (in module SCons.Node)
exists_entry() (in module SCons.Node)
exists_file() (in module SCons.Node)
exists_none() (in module SCons.Node)

Exit() (SCons.Script.SConscript.SConsEnvironment method)

exit() (SCons.Script.SConsOptions.SConsOptionParser method)
exitstatus (SCons.Errors.BuildError attribute)
expand() (SCons.Subst.ListSubber method)
(expand_default() (SCons.Script.SConsOptions.SConsIndentedHelpFormatter method)
expand_prog_name() (SCons.Script.SConsOptions.SConsOptionParser method)
expanded() (SCons.Subst.ListSubber method)
expandtabs() (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)

ExplicitExit
Export() (SCons.Script.SConscript.SConsEnvironment method)

extend() (SCons.Builder.ListEmitter method)
(SCons.Executor.TSList 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)

F90Scanner (class in SCons.Scanner.Fortran)
fail_continue() (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)

fail_stop() (SCons.SConf.SConfBuildTask method)
(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)
FutureDeprecationWarning
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)
generate_versioned_lib_info() (SCons.Tool._LibInfoGeneratorBase method)
    (SCons.Tool._LibNameGenerator method)
    (SCons.Tool._LibPrefixGenerator method)
    (SCons.Tool._LibSonameGenerator method)
    (SCons.Tool._LibSuffixGenerator method)
    (SCons.Tool._LibSymlinkGenerator method)
GenerateHelpText() (SCons.Variables.Variables method)
genstring() (SCons.Action._ActionAction method)
    (SCons.Action.CommandGeneratorAction 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)
getName() (SCons.Job.Worker method)
GetOption() (in module SCons.Script.Main)
   (SCons.Script.SConscript.SConsEnvironment method)
getRepositories() (SCons.Node.FS.Dir method)
   (SCons.Node.FS.RootDir method)
getsize() (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)
   (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.RootDir method)
getvalue() (SCons.SConf.Streamer method)
Glob() (SCons.Environment.Base method)
   (SCons.Environment.OverrideEnvironment method)
glob() (SCons.Node.FS.Dir method)
Glob() (SCons.Node.FS.FS method)
glob() (SCons.Node.FS.RootDir method)
Glob() (SCons.Script.SConscript.SConsEnvironment method)
gvars() (SCons.Environment.Base method)
   (SCons.Environment.OverrideEnvironment method)
   (SCons.Environment.SubstitutionEnvironment method)
   (SCons.Script.SConscript.SConsEnvironment method)
has_explicit_builder() (SCons.Node.Alias.Alias method)
has_glob_magic() (in module SCons.Node.FS)
has_key() (SCons.dblite.dblite method)
   (SCons.Environment.Base method)
   (SCons.Environment.OverrideEnvironment method)
   (SCons.Environment.SubstitutionEnvironment method)
   (SCons.Script.SConscript.SConsEnvironment method)
has_option()
   (SCons.Script.SConsOptions.SConsOptionGroup method)
   (SCons.Script.SConsOptions.SConsOptionParser method)
has_src_builder() (SCons.Node.FS.File method)
Help() (SCons.Script.SConscript.SConsEnvironment method)
HelpFunction() (in module SCons.Script)
hit_ratio() (SCons.CacheDir.CacheDir property)
ident() (SCons.Job.Worker property)
identchars (SCons.Script.Interactive.SConsInteractiveCmd attribute)
IDLScan() (in module SCons.Scanner.IDL)
IDX() (in module SCons.Util)
   (SCons.Node.FS.Base attribute)
   (SCons.Node.FS.Dir attribute)
   (SCons.Node.FS.Entry attribute)
   (SCons.Node.FS.File attribute)
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.FS method)
   (SCons.Node.FS.LocalFS method)
   (SCons.Node.FS.RootDir method)
islink() (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)
isslower() (SCons.Subst.CmdStringHolder method)
isnumeric() (SCons.Subst.CmdStringHolder method)
isspace() (SCons.Subst.CmdStringHolder method)
istitle() (SCons.Subst.CmdStringHolder method)
isupper() (SCons.Subst.CmdStringHolder method)
items() (SCons.Builder.CallableSelector 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.Script.SConscript.SConsEnvironment method)
    (SCons.Util.Selector method)
    (SCons.Variables.Variables method)
iterkeys() (SCons.dblite.dblite method)

J

Jobs (class in SCons.Job)
join() (SCons.Job.Worker method)
    (SCons.Subst.Cmd method)

K

key() (SCons.Memoize.CountDict method)
    (SCons.Memoize.Counter method)
    (SCons.Memoize.CountValue method)
keys() (SCons.Builder.CallableSelector method)
    (SCons.Builder.DictCmdGenerator method)
    (SCons.Builder.DictEmitter method)
    (SCons.Builder.OverrideWarner method)
    (SCons.dblite.dblite method)

L

lastcmd
    (SCons.Script.Interactive.SConsInteractiveCmd attribute)
LaTeX (class in SCons.Scanner.LaTeX)
LaTeXScanner() (in module SCons.Scanner.LaTeX)
LazyAction (class in SCons.Action)
LibSymlinksActionFunction() (in module SCons.Tool)
LibSymlinksStrFun() (in module SCons.Tool)
libtype() (SCons.Tool._ImpLibInfoSupport property)
    (SCons.Tool._LdModInfoSupport property)
    (SCons.Tool._LibInfoGeneratorBase property)
    (SCons.Tool._LibNameGenerator property)
    (SCons.Tool._LibPrefixGenerator property)
    (SCons.Tool._LibSonameGenerator property)
    (SCons.Tool._LibSuffixGenerator property)
    (SCons.Tool._LibSymlinkGenerator property)
    (SCons.Tool._ShLibInfoSupport property)
link() (SCons.Node.FS.Dir method)
    (SCons.Node.FS.FS method)
    (SCons.Node.FS.LocalFS method)
    (SCons.Node.FS.RootDir method)
linked (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)
keyword_paths (SCons.Scanner.LaTeX.LaTeX attribute)
match_splitext() (in module SCons.Builder)
md5_chunksize (SCons.Node.FS.File attribute)
MD5collect() (in module SCons.Util)
MD5signature() (in module SCons.Util)
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)
   (SCons.Node.NodeInfoBase method)
   (SCons.SConf.SConfBuildInfo method)
   (SCons.SConsign.Base method)
   (SCons.SConsign.DB method)
   (SCons.SConsign.Dir method)
   (SCons.SConsign.DirFile method)
MergeFlags() (SCons.Environment.Base method)
   (SCons.Environment.OverrideEnvironment method)
   (SCons.Environment.SubstitutionEnvironment method)
   (SCons.Script.SConscript.SConscriptEnvironment method)
Message() (SCons.SConf.CheckContext method)
MethodWrapper (class in SCons.Environment)
misc_header
   (SCons.Script.Interactive.SConsInteractiveCmd attribute)
MisleadingKeywordsWarning
misses() (SCons.CacheDir.CacheDir property)
   (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.Python method)
MissingSConscriptWarning
mkdir() (SCons.Node.FS.FS method)
   (SCons.Node.FS.LocalFS method)
mkdir_func() (in module SCons.Defaults)
MkdirFunc() (in module SCons.Node.FS)
modify_env_var() (in module SCons.Scanner.LaTeX)
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)
MSVCErro
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)
N
name (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)
name() (SCons.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)
NoObjectCountWarning
NoParallelSupportWarning
NoSlotsPyPy (class in SCons.compat)
NoSubstitutionProxy() (in module SCons.Environment)
Null (class in SCons.Executor)
  (class in SCons.Util)
NullCmdGenerator (class in SCons.Defaults)
NullEnvironment (class in SCons.Executor)
nullify() (SCons.Executor.Executor method)
NullNodeList (class in SCons.Subst)
NullNodesList (in module SCons.Subst)
NullSeq (class in SCons.Util)

O

on_disk_entries (SCons.Node.FS.Dir attribute)
  (SCons.Node.FS.Entry attribute)
  (SCons.Node.FS.File attribute)
  (SCons.Node.FS.RootDir attribute)
oncmd() (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)
on_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)
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.Base 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.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)
PlainWindowsError
Platform() (in module SCons.Platform)
  (SCons.Environment.Base method)
  (SCons.Environment.OverrideEnvironment method)
  (SCons.Script.SConscript.SConsEnvironment method)
Pseudo() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Script.SConscript.SConsEnvironment method)

pspawn_wrapper() (SCons.SConf.SConfBase method)
push() (SCons.CacheDir.CacheDir method)
push_if_forced() (SCons.CacheDir.CacheDir 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)

put() (SCons.Job.ThreadPool method)

PyPackageDir() (SCons.Environment.Base method)
(SCons.Environment.OverrideEnvironment method)
(SCons.Node.FS.FS method)
(SCons.Script.SConscript.SConsEnvironment method)

python_version_deprecated() (in module SCons.Script.Main)
python_version_string() (in module SCons.Script.Main)
python_version_unsupported() (in module SCons.Script.Main)

PythonVersionWarning

Q

QuestionTask (class in SCons.Script.Main)
quote_spaces() (in module SCons.Subst)

R

raise_exception() (in module SCons.Subst)
RCScan() (in module SCons.Scanner.RC)
rdir() (SCons.Node.FS.Dir method)
(SCons.Node.FS.RootDir method)
RDirs() (SCons.Node.FS.Base method)
(SCons.Node.FS.Dir method)
(SCons.Node.FS.Entry method)
(SCons.Node.FS.File method)
(SCons.Node.FS.RootDir method)

read_file() (SCons.cpp.DumbPreProcessor method)
(SCons.cpp.PreProcessor method)
(SCons.Scanner.C.SConsCPPConditionalScanner method)
(SCons.Scanner.C.SConsCPPScanner method)
(SCons.Script.SConsOptions.SConsValues method)

read_module()
(SCons.Script.SConsOptions.SConsValues method)
readlines() (SCons.Util.LogicalLines method)
readlink() (SCons.Node.FS.FS method)
(SCons.Node.FS.LocalFS method)
recure_nodes() (SCons.Scanner.C.SConsCPPConditionalScannerWrapper method)
(SCons.Scanner.C.SConsCPPScannerWrapper 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)

RegError (in module SCons.Util)
RegGetValue() (in module SCons.Util)
register() (in module SCons.exitfuncs)
RegOpenKeyEx() (in module SCons.Util)
rel_path() (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)

released_target_info (SCons.Node.FS.Dir attribute)
<table>
<thead>
<tr>
<th>Documentation and Textual Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SCons.Job.Worker method)</td>
</tr>
<tr>
<td>RunProg() (SCons.SConf.CheckContext method)</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>save() (SCons.cpp.DumbPreProcessor method)</td>
</tr>
<tr>
<td>(SCons.cpp.PreProcessor method)</td>
</tr>
<tr>
<td>(SCons.Scanner.C.SConsCPPConditionalScanner method)</td>
</tr>
<tr>
<td>(SCons.Scanner.C.SConsCPPScanner method)</td>
</tr>
<tr>
<td>Save() (SCons.Variables.Variables method)</td>
</tr>
<tr>
<td>save_strings() (in module SCons.Node.FS)</td>
</tr>
<tr>
<td>sbuilder (SCons.Node.FS.Base attribute)</td>
</tr>
<tr>
<td>(SCons.Node.FS.Dir attribute)</td>
</tr>
<tr>
<td>(SCons.Node.FS.Entry attribute)</td>
</tr>
<tr>
<td>(SCons.Node.FS.File attribute)</td>
</tr>
<tr>
<td>(SCons.Node.FS.RootDir attribute)</td>
</tr>
<tr>
<td>scan() (in module SCons.Scanner.Prog)</td>
</tr>
<tr>
<td>(SCons.Executor.Executor method)</td>
</tr>
<tr>
<td>(SCons.Node.FS.Base method)</td>
</tr>
<tr>
<td>(SCons.Node.FS.Dir method)</td>
</tr>
<tr>
<td>(SCons.Node.FS.Entry method)</td>
</tr>
<tr>
<td>(SCons.Node.FS.File method)</td>
</tr>
<tr>
<td>(SCons.Node.FS.RootDir method)</td>
</tr>
<tr>
<td>(SCons.Scanner.Classic method)</td>
</tr>
<tr>
<td>(SCons.Scanner.ClassicCPP method)</td>
</tr>
<tr>
<td>(SCons.Scanner.D.D method)</td>
</tr>
<tr>
<td>(SCons.Scanner.Fortran.F90Scanner method)</td>
</tr>
<tr>
<td>(SCons.Scanner.LaTeX.LaTeX method)</td>
</tr>
<tr>
<td>scan_in_memory() (in module SCons.Scanner.Dir)</td>
</tr>
<tr>
<td>scan_on_disk() (in module SCons.Scanner.Dir)</td>
</tr>
<tr>
<td>scan_recurs() (SCons.Scanner.LaTeX.LaTeX method)</td>
</tr>
<tr>
<td>scan_sources() (SCons.Executor.Executor method)</td>
</tr>
<tr>
<td>scan_targets() (SCons.Executor.Executor method)</td>
</tr>
<tr>
<td>Scanner() (in module SCons.Scanner)</td>
</tr>
<tr>
<td>(SCons.Environment.Base method)</td>
</tr>
<tr>
<td>(SCons.Environment.OverrideEnvironment method)</td>
</tr>
<tr>
<td>(SCons.Script.SConscript.SConsEnvironment method)</td>
</tr>
<tr>
<td>(SCons.Node.FS.Base method)</td>
</tr>
<tr>
<td>(SCons.Node.FS.Dir method)</td>
</tr>
<tr>
<td>(SCons.Node.FS.Entry method)</td>
</tr>
<tr>
<td>(SCons.Node.FS.File method)</td>
</tr>
<tr>
<td>(SCons.Node.FS.RootDir method)</td>
</tr>
<tr>
<td>scanner_map_delete() (SCons.Environment.Base method)</td>
</tr>
<tr>
<td>(SCons.Environment.OverrideEnvironment method)</td>
</tr>
<tr>
<td>(SCons.Script.SConscript.SConsEnvironment method)</td>
</tr>
<tr>
<td>scanner_paths (SCons.Node.FS.Dir attribute)</td>
</tr>
<tr>
<td>(SCons.Node.FS.Entry attribute)</td>
</tr>
<tr>
<td>(SCons.Node.FS.File attribute)</td>
</tr>
<tr>
<td>(SCons.Node.FS.RootDir attribute)</td>
</tr>
<tr>
<td>SConf() (in module SCons.SConf)</td>
</tr>
<tr>
<td>SConfBase (class in SCons.SConf)</td>
</tr>
<tr>
<td>SConfBase.TestWrapper (class in SCons.SConf)</td>
</tr>
<tr>
<td>SConfBuildInfo (class in SCons.SConf)</td>
</tr>
<tr>
<td>SConfBuildTask (class in SCons.SConf)</td>
</tr>
<tr>
<td>SConfError</td>
</tr>
<tr>
<td>SConfWarning</td>
</tr>
<tr>
<td>SCons (module)</td>
</tr>
<tr>
<td>(SCons.Executor_NullEnvironment attribute)</td>
</tr>
<tr>
<td>SCons.Action (module)</td>
</tr>
<tr>
<td>SCons.Builder (module)</td>
</tr>
<tr>
<td>SCons.CacheDir (module)</td>
</tr>
<tr>
<td>SCons.compat (module)</td>
</tr>
<tr>
<td>SCons.Conftest (module)</td>
</tr>
<tr>
<td>SCons.cpp (module)</td>
</tr>
<tr>
<td>SCons.dblite (module)</td>
</tr>
<tr>
<td>SCons.Debug (module)</td>
</tr>
<tr>
<td>SCons.Defaults (module)</td>
</tr>
<tr>
<td>SCons.Environment (module)</td>
</tr>
<tr>
<td>SCons.Errors (module)</td>
</tr>
<tr>
<td>SCons.Executor (module)</td>
</tr>
<tr>
<td>SCons.exitfuncs (module)</td>
</tr>
<tr>
<td>SCons.Job (module)</td>
</tr>
<tr>
<td>SCons.Memoize (module)</td>
</tr>
<tr>
<td>SCons.Node (module)</td>
</tr>
<tr>
<td>Function/Method</td>
</tr>
<tr>
<td>------------------------------------------------------</td>
</tr>
<tr>
<td>SetDefault()</td>
</tr>
<tr>
<td>setdefault()</td>
</tr>
<tr>
<td>SetDefault()</td>
</tr>
<tr>
<td>setdefault()</td>
</tr>
<tr>
<td>setLIBS()</td>
</tr>
<tr>
<td>setName()</td>
</tr>
<tr>
<td>SetOption()</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>SetProgressDisplay()</td>
</tr>
<tr>
<td>settable</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>SharedFlagChecker()</td>
</tr>
<tr>
<td>SharedObjectEmitter()</td>
</tr>
<tr>
<td>show()</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>sources</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>spawn()</td>
</tr>
<tr>
<td>spawnve()</td>
</tr>
<tr>
<td>SpecialAttrWrapper</td>
</tr>
<tr>
<td>spinner()</td>
</tr>
<tr>
<td>Split()</td>
</tr>
</tbody>
</table>
two_arg_commands  (SCons.Scanner.LaTeX.LaTeX attribute)

TYPE_CHECKER
(SCons.Script.SConsOptions.SConsOption attribute)

TYPED_ACTIONS
(SCons.Script.SConsOptions.SConsOption attribute)

TYPES  (SCons.Script.SConsOptions.SConsOption attribute)

Unbuffered (class in SCons.Util)

undoc_header
(SCons.Script.Interactive.SConsInteractiveCmd attribute)

unique() (in module SCons.Util)

UniqueList (class in SCons.Util)

uniquer() (in module SCons.Util)

uniquer_hashables() (in module SCons.Util)

UnknownVariables()  (SCons.Variables.Variables method)

unlink() (SCons.Node.FS.FS method)

    (SCons.Node.FS.LocalFS method)

UnlinkFunc() (in module SCons.Node.FS)

up() (SCons.Node.FS.Dir method)

    (SCons.Node.FS.RootDir method)

update() (SCons.Builder.CallableSelector method)

    (SCons.Builder.DictCmdGenerator method)

    (SCons.Builder.DictEmitter method)

    (SCons.Builder.OverrideWarner method)

    (SCons.Environment.BuilderDict method)

    (SCons.Node.Alias.AliasNameSpace method)


    (SCons.Node.FS.DirNodeInfo method)

    (SCons.Node.FS.FileNodeInfo method)

    (SCons.Node.NodeInfoBase method)


    (SCons.Util.Selector method)

Update() (SCons.Variables.Variables method)

updrive() (in module SCons.Util)

upper() (SCons.Subst.CmdStringHolder method)

use_rawinput
(SCons.Script.Interactive.SConsInteractiveCmd attribute)

UserError

Value (class in SCons.Node.Python)

Value()  (SCons.Environment.Base method)

    (SCons.Environment.OverrideEnvironment method)

    (SCons.Script.SConscript.SConsEnvironment method)

Value.Attrs (class in SCons.Node.Python)

ValueBuildInfo (class in SCons.Node.Python)

ValueNodeInfo (class in SCons.Node.Python)

values (SCons.Script.Main.FakeOptionParser attribute)

values() (SCons.Builder.CallableSelector attribute)

    (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.Script.SConscript.SConsEnvironment method)

    (SCons.Util.Selector method)

ValueWithMemo() (in module SCons.Node.Python)

Variable_Method_Caller (class in SCons.Defaults)

Variables (class in SCons.Variables)

Variables() (in module SCons.Script)

variant_dir_target_climb()  (SCons.Node.FS.FS method)

variant_dirs (SCons.Node.FS.Dir attribute)

    (SCons.Node.FS.Entry attribute)

    (SCons.Node.FS.File attribute)

    (SCons.Node.FS.RootDir attribute)

VariantDir() (SCons.Environment.Base method)

    (SCons.Environment.OverrideEnvironment method)

    (SCons.Node.FS.FS method)

    (SCons.Script.SConscript.SConsEnvironment method)

version_string() (in module SCons.Script.Main)

Virtualenv() (in module SCons.Platform.virtualenv)

(SCons.Warnings.MissingSConscriptWarning method)
(SCons.Warnings.NoObjectCountWarning method)
(SCons.Warnings.NoParallelSupportWarning method)
(SCons.Warnings.PythonVersionWarning method)
(SCons.Warnings.ReservedVariableWarning method)
(SCons.Warnings.StackSizeWarning method)
(SCons.WarningsTargetException method)
(SCons.Warnings.TaskmasterNeedsExecuteWarning method)
(SCons.Warnings.VisualCMissingWarning method)
(SCons.Warnings.VisualStudioMissingWarning method)
(SCons.Warnings.VisualStudioMismatch method)
(SCons.Warnings.Warning method)
(SCons.Warnings.WarningOnByDefault 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)

Worker (class in SCons.Job)
write() (in module SCons.SConsign)
(SCons.SConf.Streamer method)
(SCons.SConsign.DB method)
(SCons.SConsign.DirFile method)
(SCons.Script.Main.Progressor method)
(SCons.Util.Unbuffered method)

writelines() (SCons.SConf.Streamer method)

z

zfill() (SCons.Subst.CmdStringHolder method)
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.Job
SCons.Memoize
SCons.Node
SCons.Node.Alias
SCons.Node.FS
SCons.Node.Python
SCons.PathList
SCons.Platform
SCons.Platform.aix
SCons.Platform.cygwin
SCons.Platform.darwin
SCons.Platform.hpux
SCons.Platform.irix
SCons.Platform.mingw
SCons.Platform.os2
SCons.Platform.posix
SCons.Platform.sunos
SCons.Platform.virtualenv
SCons.Platform.win32
SCons.Scanner
SCons.Scanner.C
SCons.Scanner.D
SCons.Scanner.Dir
SCons.Scanner.Fortran
SCons.Scanner.IDL
SCons.Scanner.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.Tool
SCons.Util
SCons.Variables
SCons.Variables.BoolVariable
SCons.Variables.EnumVariable
SCons.Variables.ListVariable
SCons.Variables.PackageVariable
SCons.Variables.PathVariable
SCons.Warnings