SCONS

Build your software, better.

SCons 4.3.0

MAN page

The SCons Development Team

Version 4.3.02001 - 2021The SCons FoundationReleased Tue, 16 Nov 2021 19:09:21 +0000

Name

scons — a software construction tool

Synopsis

`scons` [*`options`*...] [*`name`*=*`val`*...] [*`targets`*...]

DESCRIPTION

scons orchestrates the construction of software (and other tangible products such as documentation files) by determining which component pieces must be built or rebuilt and invoking the necessary commands to build them. SCons offers many features to improve developer productivity such as parallel builds, caching of build artifacts, automatic dependency scanning, and a database of information about previous builds so details do not have to be recalculated each run.

scons requires Python 3.5 or later to run; there should be no other dependencies or requirements. *Support for Python 3.5 is deprecated since SCons 4.2 and will be dropped in a future release. The CPython project has retired 3.5: https://www.python.org/dev/peps/pep-0478.*

You set up an SCons build system by writing a script that describes things to build (*targets*), and, if necessary, the rules to build those files (*actions*). SCons comes with a collection of *Builder* methods which apply premade actions for building many common software components such as executable programs, object files and libraries, so that for many software projects, only the targets and input files (*sources*) need be specified in a call to a builder. **scons** thus can operate at a level of abstraction above that of pure files. For example if you specify a library target named "foo", **scons** keeps track of the actual operating system dependent filename (such as `libfoo.so` on a GNU/Linux system), and how to refer to that library in later construction steps that want to use it, so you don't have to specify that precise information yourself. **scons** can also scan automatically for dependency information, such as header files included by source code files, so this does not have to be specified manually.

When invoked, **scons** looks for a file named `SConstruct` in the current directory and reads the build configuration from that file (other names are allowed, see the section called “SConscript Files” for more information). The `SConstruct` file may specify subsidiary configuration files by calling the `SConscript` function. By convention, these subsidiary files are named `SConscript`, although any name may be used. As a result of this naming convention, the term *SConscript files* is used to refer generically to the complete set of configuration files for a project (including the `SConstruct` file), regardless of the actual file names or number of such files.

Before reading the SConscript files, **scons** looks for a directory named `site_scons` in various system directories and in the directory containing the `SConstruct` file or, if specified, the directory from the `--site-dir` option instead, and prepends the ones it finds to the Python module search path (`sys.path`), thus allowing modules in such directories to be imported in the normal Python way in SConscript files. For each found site directory, (1) if it contains a file `site_init.py` that file is evaluated, and (2) if it contains a directory `site_tools` the path to that directory is prepended to the default toolpath. See the `--site-dir` and `--no-site-dir` options for details on default paths and controlling the site directories.

SConscript files are written in the *Python* programming language, although it is normally not necessary to be a Python programmer to use **scons** effectively. SConscript files are invoked in a context that makes the facilities described in this manual page available in their local namespace without any special steps. Standard Python scripting capabilities such as flow control, data manipulation, and imported Python libraries are available to use to handle complicated build situations. Other Python files can be made a part of the build system, but they do not automatically have the SCons context and need to import it if they need access (described later).

scons reads and executes all of the included SConscript files *before* it begins building any targets. To make this clear, **scons** prints the following messages about what it is doing:

```
$ scons foo.out
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets  ...
cp foo.in foo.out
scons: done building targets.
$
```

The status messages (lines beginning with the `scons:` tag) may be suppressed using the `-Q` option.

scons does not automatically propagate the external environment used to execute **scons** to the commands used to build target files. This is so that builds will be guaranteed repeatable regardless of the environment variables set at the time **scons** is invoked. This also means that if the compiler or other commands that you want to use to build your target files are not in standard system locations, **scons** will not find them unless you explicitly include the locations into the execution environment by setting the path in the `ENV` construction variable in the internal construction environment:

```
import os
env = Environment(ENV={'PATH': os.environ['PATH']})
```

Similarly, if the commands use specific external environment variables that **scons** does not recognize, they can be propagated into the execution environment:

```
import os

env = Environment(
    ENV={
        'PATH': os.environ['PATH'],
        'ANDROID_HOME': os.environ['ANDROID_HOME'],
        'ANDROID_NDK_HOME': os.environ['ANDROID_NDK_HOME'],
    }
)
```

Or you may explicitly propagate the invoking user's complete external environment:

```
import os
env = Environment(ENV=os.environ.copy())
```

This comes at the expense of making your build dependent on the user's environment being set correctly, but it may be more convenient for many configurations. It should not cause problems if done in a build setup which tightly controls how the environment is set up before invoking **scons**, as in many continuous integration setups.

scons can scan known input file types automatically for dependency information (for example, `#include` preprocessor directives in C or C++ files) and will rebuild dependent files appropriately whenever any "included" input file changes. **scons** supports the ability to define new scanners to support additional input file types.

scons is normally executed in a top-level directory containing an `SConstruct` file. When **scons** is invoked, the command line (including the contents of the `SCONSFLAGS` environment variable, if set) is processed. Command-line options (see the section called “OPTIONS”) are consumed. Any variable argument assignments are collected, and remaining arguments are taken as targets to build.

Values of variables to be passed to the SConscript files may be specified on the command line:

```
scons debug=1
```

These variables are available through the `ARGUMENTS` dictionary, and can be used in the SConscript files to modify the build in any way:

```
if ARGUMENTS.get('debug', 0):
    env = Environment(CCFLAGS='-g')
else:
    env = Environment()
```

The command-line variable arguments are also available in the `ARGLIST` list, indexed by their order on the command line. This allows you to process them in order rather than by name, if necessary. Each `ARGLIST` entry is a tuple containing (*`argname`*, *`argvalue`*).

See the section called “Command-Line Construction Variables” for more information.

scons can maintain a cache of target (derived) files that can be shared between multiple builds. When derived-file caching is enabled in an SConscript file, any target files built by **scons** will be copied to the cache. If an up-to-date target file is found in the cache, it will be retrieved from the cache instead of being rebuilt locally. Caching behavior may be disabled and controlled in other ways by the `--cache-force`, `--cache-disable`, `--cache-readonly`, and `--cache-show` command-line options. The `--random` option is useful to prevent multiple builds from trying to update the cache simultaneously.

By default, **scons** searches for known programming tools on various systems and initializes itself based on what is found. On Windows systems which identify as *win32*, **scons** searches in order for the Microsoft Visual C++ tools, the MinGW tool chain, the Intel compiler tools, and the PharLap ETS compiler. On Windows system which identify as *cygwin* (that is, if **scons** is invoked from a cygwin shell), the order changes to prefer the GCC toolchain over the MSVC tools. On OS/2 systems, **scons** searches in order for the OS/2 compiler, the GCC tool chain, and the Microsoft Visual C++ tools, On SGI IRIX, IBM AIX, Hewlett Packard HP-UX, and Oracle Solaris systems, **scons** searches for the native compiler tools (MIPSpro, Visual Age, aCC, and Forte tools respectively) and the GCC tool chain. On all other platforms, including POSIX (Linux and UNIX) platforms, **scons** searches in order for the GCC tool chain, and the Intel compiler tools. These default values may be overridden by appropriate setting of construction variables.

Target Selection

SCons acts on the *selected targets*, whether the requested operation is build, no-exec or clean. Targets are selected as follows:

1. Targets specified on the command line. These may be files, directories, or phony targets defined using the `Alias` function. Directory targets are scanned by **scons** for any targets that may be found with a destination in or under that directory. The targets listed on the command line are made available in the `COMMAND_LINE_TARGETS` list.

2. If no targets are specified on the command line, **scons** will select those targets specified in the SConscript files via calls to the `Default` function. These are known as the *default targets*, and are made available in the `DEFAULT_TARGETS` list.

3. If no targets are selected by the previous steps, **scons** selects the current directory for scanning, unless command-line options which affect the target scan are detected (`-C`, `-D`, `-u`, `-U`). Since targets thus selected were not the result of user instructions, this target list is not made available for direct inspection; use the `--debug=explain` option if they need to be examined.

4. **scons** always adds to the selected targets any intermediate targets which are necessary to build the specified ones. For example, if constructing a shared library or dll from C source files, **scons** will also build the object files which will make up the library.

To ignore the default targets specified through calls to `Default` and instead build all target files in or below the current directory specify the current directory (`.`) as a command-line target:

```
scons .
```

To build all target files, including any files outside of the current directory, supply a command-line target of the root directory (on POSIX systems):

```
scons /
```

or the path name(s) of the volume(s) in which all the targets should be built (on Windows systems):

```
scons C:\ D:\
```

A subset of a hierarchical tree may be built by remaining at the top-level directory (where the `SConstruct` file lives) and specifying the subdirectory as the target to build:

```
scons src/subdir
```

or by changing directory and invoking scons with the `-u` option, which traverses up the directory hierarchy until it finds the `SConstruct` file, and then builds targets relatively to the current subdirectory (see also the related `-D` and `-U` options):

```
cd src/subdir
scons -u .
```

In all cases, more files may be built than are requested, as **scons** needs to make sure any dependent files are built.

Specifying "cleanup" targets in SConscript files is usually not necessary. The `-c` flag removes all selected targets:

```
scons -c .
```

to remove all target files in or under the current directory, or:

```
scons -c build export
```

to remove target files under `build` and `export`.

Additional files or directories to remove can be specified using the `Clean` function in the SConscript files. Conversely, targets that would normally be removed by the `-c` invocation can be retained by calling the `NoClean` function with those targets.

scons supports building multiple targets in parallel via a `-j` option that takes, as its argument, the number of simultaneous tasks that may be spawned:

```
scons -j 4
```

builds four targets in parallel, for example.

OPTIONS

In general, **scons** supports the same command-line options as GNU Make and many of those supported by cons.

`-b`
: Ignored for compatibility with non-GNU versions of Make

`-c`, `--clean`, `--remove`
: Set *clean* mode. Clean up by removing the selected targets, well as any files or directories associated with a selected target through calls to the `Clean` function. Will not remove any targets which are marked for preservation through calls to the `NoClean` function.

  While clean mode removes targets rather than building them, work which is done directly in Python code in SConscript files will still be carried out. If it is important to avoid some such work from taking place in clean mode, it should be protected. An SConscript file can determine which mode is active by querying `GetOption`, as in the call `if GetOption("clean"):`

`--cache-debug=`*`file`*
: Write debug information about derived-file caching to the specified *`file`*. If *`file`* is a hyphen (`-`), the debug information is printed to the standard output. The printed messages describe what signature-file names are being looked for in, retrieved from, or written to the derived-file cache specified by `CacheDir`.

`--cache-disable`, `--no-cache`
: Disable derived-file caching. **scons** will neither retrieve files from the cache nor copy files to the cache. This option can be used to temporarily disable the cache without modifying the build scripts.

`--cache-force`, `--cache-populate`
: When using `CacheDir`, populate a derived-file cache by copying any already-existing, up-to-date derived files to the cache, in addition to files built by this invocation. This is useful to populate a new cache with all the current derived files, or to add to the cache any derived files recently built with caching disabled via the `--cache-disable` option.

`--cache-readonly`
: Use the derived-file cache, if enabled, to retrieve files, but do not not update the cache with any files actually built during this invocation.

`--cache-show`
: When using a derived-file cache show the command that would have been executed to build the file (or the corresponding `*COMSTR` contents if set) even if the file is retrieved from cache. Without this option, **scons** shows a cache retrieval message if the file is fetched from cache. This allows producing consistent output for build logs, regardless of whether a target file was rebuilt or retrieved from the cache.

`--config=`*`mode`*
: Control how the `Configure` call should use or generate the results of configuration tests. *`mode`* should be one of the following choices:

  auto
  : SCons will use its normal dependency mechanisms to decide if a test must be rebuilt or not. This saves time by not running the same configuration tests every time you invoke scons, but will overlook changes in system header files or external commands (such as compilers) if you don't specify those dependecies explicitly. This is the default behavior.

  force
  : If this mode is specified, all configuration tests will be re-run regardless of whether the cached results are out of date. This can be used to explicitly force the configuration tests to be updated in response to an otherwise unconfigured change in a system header file or compiler.

cache

If this mode is specified, no configuration tests will be rerun and all results will be taken from cache. **scons** will report an error if `--config=cache` is specified and a necessary test does not have any results in the cache.

`-C` *`directory`*, `--directory=`*`directory`*

Run as if **scons** was started in *`directory`* instead of the current working directory. That is, change directory before searching for the `SConstruct`, `Sconstruct`, `sconstruct`, `SConstruct.py`, `Sconstruct.py` or `sconstruct.py` file or doing anything else. When multiple `-C` options are given, each subsequent non-absolute `-C` *`directory`* is interpreted relative to the preceding one. This option is similar to using `-f` *`directory`*`/SConstruct`, but `-f` does not search for any of the predefined `SConstruct` names in the specified directory. See also options `-u`, `-U` and `-D` to change the `SConstruct` search behavior when this option is used.

`-D`

Works exactly the same way as the `-u` option except for the way default targets are handled. When this option is used and no targets are specified on the command line, all default targets are built, whether or not they are below the current directory.

`--debug=`*`type`*`[,`*`type`*`...]`

Debug the build process. *`type`* specifies the kind of debugging info to emit. Multiple types may be specified, separated by commas. The following types are recognized:

action-timestamps

Prints additional time profiling information. For each command, shows the absolute start and end times. This may be useful in debugging parallel builds. Implies the `--debug=time` option.

*Available since **scons** 3.1.*

count

Print how many objects are created of the various classes used internally by SCons before and after reading the SConscript files and before and after building targets. This is not supported when SCons is executed with the Python `-O` (optimized) option or when the SCons modules have been compiled with optimization (that is, when executing from `*.pyo` files).

duplicate

Print a line for each unlink/relink (or copy) of a variant file from its source file. Includes debugging info for unlinking stale variant files, as well as unlinking old targets before building them.

explain

Print an explanation of why **scons** is deciding to (re-)build the targets it selects for building.

findlibs

Instruct the scanner that searches for libraries to print a message about each potential library name it is searching for, and about the actual libraries it finds.

includes

Print the include tree after each top-level target is built. This is generally used to find out what files are included by the sources of a given derived file:

```
$ scons --debug=includes foo.o
```

memoizer

Prints a summary of hits and misses using the Memoizer, an internal subsystem that counts how often SCons uses cached values in memory instead of recomputing them each time they're needed.

memory

Prints how much memory SCons uses before and after reading the SConscript files and before and after building targets.

objects

Prints a list of the various objects of the various classes used internally by SCons.

pdb

Re-run **scons** under the control of the **pdb** Python debugger.

prepare

Print a line each time any target (internal or external) is prepared for building. **scons** prints this for each target it considers, even if that target is up to date (see also `--debug=explain`). This can help debug problems with targets that aren't being built; it shows whether **scons** is at least considering them or not.

presub

Print the raw command line used to build each target before the construction environment variables are substituted. Also shows which targets are being built by this command. Output looks something like this:

```
$ scons --debug=presub
Building myprog.o with action(s):
  $SHCC $SHCFLAGS $SHCCFLAGS $CPPFLAGS $_CPPINCFLAGS -c -o $TARGET $SOURCES
...
```

stacktrace

Prints an internal Python stack trace when encountering an otherwise unexplained error.

time

Prints various time profiling information:

- The time spent executing each individual build command
- The total build time (time SCons ran from beginning to end)
- The total time spent reading and executing SConscript files
- The total time SCons itself spent running (that is, not counting reading and executing SConscript files)
- The total time spent executing all build commands
- The elapsed wall-clock time spent executing those build commands
- The time spent processing each file passed to the `SConscript` function

(When **scons** is executed without the `-j` option, the elapsed wall-clock time will typically be slightly longer than the total time spent executing all the build commands, due to the SCons processing that takes place in between executing each command. When **scons** is executed *with* the `-j` option, and your build configuration allows good parallelization, the elapsed wall-clock time should be significantly smaller than the total time spent executing all the build commands, since multiple build commands and intervening SCons processing should take place in parallel.)

`--diskcheck=`*`type`*

Enable specific checks for whether or not there is a file on disk where the SCons configuration expects a directory (or vice versa) when searching for source and include files. *`type`* can be an available diskcheck type or the special tokens `all` or `none`. A comma-separated string can be used to select multiple checks. The default setting is `all`.

Current available checks are:

match
to check that files and directories on disk match SCons' expected configuration.

Disabling some or all of these checks can provide a performance boost for large configurations, or when the configuration will check for files and/or directories across networked or shared file systems, at the slight increased risk of an incorrect build or of not handling errors gracefully.

`--duplicate=`*`ORDER`*
There are three ways to duplicate files in a build tree: hard links, soft (symbolic) links and copies. The default policy is to prefer hard links to soft links to copies. You can specify a different policy with this option. *`ORDER`* must be one of *hard-soft-copy* (the default), *soft-hard-copy*, *hard-copy*, *soft-copy* or *copy*. SCons will attempt to duplicate files using the mechanisms in the specified order.

`--enable-virtualenv`
Import virtualenv-related variables to SCons.

`--experimental=`*`feature`*
Enable experimental features and/or tools. *`feature`* can be an available feature name or the special tokens `all` or `none`. A comma-separated string can be used to select multiple features. The default setting is `none`.

Current available features are: `ninja`.

Caution

No Support offered for any features or tools enabled by this flag.

*Available since **scons** 4.2.*

`-f` ***`file`*, `--file=`*`file`*, `--makefile=`*`file`*, `--sconstruct=`*`file`***
Use *`file`* as the initial SConscript file. Multiple `-f` options may be specified, in which case **scons** will read all of the specified files.

`-h`, `--help`
Print a local help message for this project, if one is defined in the SConscript files (see the `Help` function), plus a line that refers to the standard SCons help message. If no local help message is defined, prints the standard SCons help message (as for the `-H` option) plus help for any local options defined through `AddOption`. Exits after displaying the appropriate message.

Note that use of this option requires SCons to process the SConscript files, so syntax errors may cause the help message not to be displayed.

`--hash-chunksize=`*`KILOBYTES`*
Set the block size used when computing content signatures to *`KILOBYTES`*. This value determines the size of the chunks which are read in at once when computing signature hashes. Files below that size are fully stored in memory before performing the signature computation while bigger files are read in block-by-block. A huge block-size leads to high memory consumption while a very small block-size slows down the build considerably.

The default value is to use a chunk size of 64 kilobytes, which should be appropriate for most uses.

*Available since **scons** 4.2.*

`--hash-format=`*`ALGORITHM`*
Set the hashing algorithm used by SCons to *`ALGORITHM`*. This value determines the hashing algorithm used in generating content signatures or `CacheDir` keys.

The supported list of values are: md5, sha1, and sha256. However, the Python interpreter used to run SCons must have the corresponding support available in the `hashlib` module to use the specified algorithm.

Specifying this value changes the name of the SConsign database. For example, `--hash-format=sha256` will create a SConsign database with name `.sconsign_sha256.dblite`.

If this option is not specified, a the first supported hash format found is selected. Typically this is MD5, however, if you are on a FIPS-compliant system and using a version of Python less than 3.9, SHA1 or SHA256 will be chosen as the default. Python 3.9 and onwards clients will always default to MD5, even in FIPS mode, unless otherwise specified with the `--hash-format` option.

For MD5 databases (either explicitly specified with `--hash-format=md5` or defaulted), the SConsign database is`.sconsign.dblite`. The newer SHA1 and SHA256 selections meanwhile store their databases to `.sconsign_algorithmname.dblite`

*Available since **scons** 4.2.*

`-H`, `--help-options`
Print the standard help message about SCons command-line options and exit.

`-i`, `--ignore-errors`
Ignore all errors from commands executed to rebuild files.

`-I` *`directory`*, `--include-dir=`*`directory`*
Specifies a *`directory`* to search for imported Python modules. If several `-I` options are used, the directories are searched in the order specified.

`--ignore-virtualenv`
Suppress importing virtualenv-related variables to SCons.

`--implicit-cache`
Cache implicit dependencies. This causes **scons** to use the implicit (scanned) dependencies from the last time it was run instead of scanning the files for implicit dependencies. This can significantly speed up SCons, but with the following limitations:

scons will not detect changes to implicit dependency search paths (e.g. `CPPPATH`, `LIBPATH`) that would ordinarily cause different versions of same-named files to be used.

scons will miss changes in the implicit dependencies in cases where a new implicit dependency is added earlier in the implicit dependency search path (e.g. `CPPPATH`, `LIBPATH`) than a current implicit dependency with the same name.

`--implicit-deps-changed`
Forces SCons to ignore the cached implicit dependencies. This causes the implicit dependencies to be rescanned and recached. This implies `--implicit-cache`.

`--implicit-deps-unchanged`
Force SCons to ignore changes in the implicit dependencies. This causes cached implicit dependencies to always be used. This implies `--implicit-cache`.

`--install-sandbox=`*`sandbox_path`*
When using the `Install` builders, prepend *`sandbox_path`* to the installation paths such that all installed files will be placed under that directory. This option is unavailable if one of `Install`, `InstallAs` or `InstallVersionedLib` is not used in the SConscript files.

`--interactive`
Starts SCons in interactive mode. The SConscript files are read once and a `scons>>>` prompt is printed. Targets may now be rebuilt by typing commands at interactive prompt without having to re-read the SConscript files and re-initialize the dependency graph from scratch.

SCons interactive mode supports the following commands:

`build` *`[OPTIONS] [TARGETS] ...`*

Builds the specified *`TARGETS`* (and their dependencies) with the specified SCons command-line *`OPTIONS`*. **b** and **scons** are synonyms for **build**.

The following SCons command-line options affect the **build** command:

```
--cache-debug=FILE
--cache-disable, --no-cache
--cache-force, --cache-populate
--cache-readonly
--cache-show
--debug=TYPE
-i, --ignore-errors
-j N, --jobs=N
-k, --keep-going
-n, --no-exec, --just-print, --dry-run, --recon
-Q
-s, --silent, --quiet
--taskmastertrace=FILE
--tree=OPTIONS
```

Any other SCons command-line options that are specified do not cause errors but have no effect on the **build** command (mainly because they affect how the SConscript files are read, which only happens once at the beginning of interactive mode).

`clean` *`[OPTIONS] [TARGETS] ...`*

Cleans the specified *`TARGETS`* (and their dependencies) with the specified *`OPTIONS`*. **c** is a synonym. This command is itself a synonym for **`build --clean`**

`exit`

Exits SCons interactive mode. You can also exit by terminating input (**Ctrl**+**D** UNIX or Linux systems, (**Ctrl**+**Z** on Windows systems).

`help` *`[COMMAND]`*

Provides a help message about the commands available in SCons interactive mode. If *COMMAND* is specified, **h** and **?** are synonyms.

`shell` *`[COMMANDLINE]`*

Executes the specified *`COMMANDLINE`* in a subshell. If no *`COMMANDLINE`* is specified, executes the interactive command interpreter specified in the `SHELL` environment variable (on UNIX and Linux systems) or the `COMSPEC` environment variable (on Windows systems). **sh** and **!** are synonyms.

`version`

Prints SCons version information.

An empty line repeats the last typed command. Command-line editing can be used if the **readline** module is available.

```
$ scons --interactive
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons>>> build -n prog
scons>>> exit
```

`-j` *`N`*, `--jobs=`*`N`*

Specifies the maximum number of comcurrent jobs (commands) to run. If there is more than one `-j` option, the last one is effective.

`-k`, `--keep-going`

Continue as much as possible after an error. The target that failed and those that depend on it will not be remade, but other targets specified on the command line will still be processed.

`-m`

Ignored for compatibility with non-GNU versions of Make.

`--max-drift=`*`SECONDS`*

Set the maximum expected drift in the modification time of files to *`SECONDS`*. This value determines how long a file must be unmodified before its cached content signature will be used instead of calculating a new content signature (MD5 checksum) of the file's contents. The default value is 2 days, which means a file must have a modification time of at least two days ago in order to have its cached content signature used. A negative value means to never cache the content signature and to ignore the cached value if there already is one. A value of 0 means to always use the cached signature, no matter how old the file is.

`--md5-chunksize=`*`KILOBYTES`*

A deprecated synonym for `--hash-chunksize`.

*Deprecated since **scons** 4.2.*

`-n`, `--no-exec`, `--just-print`, `--dry-run`, `--recon`

Set *no execute* mode. Print the commands that would be executed to build any out-of-date target files, but do not execute the commands.

The output is a best effort, as SCons cannot always precisely determine what would be built. For example, if a file is generated by a builder action that is later used in the build, that file is not available to scan for dependencies on an unbuilt tree, or may contain out of date information in a built tree.

Work which is done directly in Python code in SConscript files, as opposed to work done by builder actions during the build phase, will still be carried out. If it is important to avoid some such work from taking place in no execute mode, it should be protected. An SConscript file can determine which mode is active by querying `GetOption`, as in the call `if GetOption("no_exec"):`

`--no-site-dir`

Prevents the automatic addition of the standard `site_scons` dirs to `sys.path`. Also prevents loading the `site_scons/site_init.py` modules if they exist, and prevents adding their `site_scons/site_tools` dirs to the toolpath.

`--package-type=`*`type`*

The type or types of package to create when using the `Package` builder. In the case of multiple types, *`type`* should be a comma-separated string; SCons will try to build for all of those packages. Note this option is only available if the `packaging` tool has been enabled.

`--profile=`*`file`*

Run SCons under the Python profiler and save the results in the specified *`file`*. The results may be analyzed using the Python pstats module.

`-q`, `--question`

Do not run any commands, or print anything. Just return an exit status that is zero if the specified targets are already up to date, non-zero otherwise.

`-Q`

Quiets SCons status messages about reading SConscript files, building targets and entering directories. Commands that are executed to rebuild target files are still printed.

`--random`

Build dependencies in a random order. This is useful when building multiple trees simultaneously with caching enabled, to prevent multiple builds from simultaneously trying to build or retrieve the same target files.

`-s`, `--silent`, `--quiet`

Silent. Do not print commands that are executed to rebuild target files. Also suppresses SCons status messages.

`-S`, `--no-keep-going`, `--stop`

Ignored for compatibility with GNU Make

`--site-dir=`*`dir`*

Uses the named *`dir`* as the site directory rather than the default `site_scons` directories. This directory will be prepended to `sys.path`, the module *`dir`*`/site_init.py` will be loaded if it exists, and *`dir`*`/site_tools` will be added to the default toolpath.

The default set of `site_scons` directories used when `--site-dir` is not specified depends on the system platform, as follows. Directories are examined in the order given, from most generic to most specific, so the last-executed `site_init.py` file is the most specific one (which gives it the chance to override everything else), and the directories are prepended to the paths, again so the last directory examined comes first in the resulting path.

Windows:

```
%ALLUSERSPROFILE/Application Data/scons/site_scons
%USERPROFILE%/Local Settings/Application Data/scons/site_scons
%APPDATA%/scons/site_scons
%HOME%/.scons/site_scons
./site_scons
```

Mac OS X:

```
/Library/Application Support/SCons/site_scons
/opt/local/share/scons/site_scons (for MacPorts)
/sw/share/scons/site_scons (for Fink)
$HOME/Library/Application Support/SCons/site_scons
$HOME/.scons/site_scons
./site_scons
```

Solaris:

```
/opt/sfw/scons/site_scons
/usr/share/scons/site_scons
$HOME/.scons/site_scons
./site_scons
```

Linux, HPUX, and other Posix-like systems:

```
/usr/share/scons/site_scons
$HOME/.scons/site_scons
```

```
./site_scons
```

`--stack-size=`*`KILOBYTES`*

Set the size stack used to run threads to *`KILOBYTES`*. This value determines the stack size of the threads used to run jobs. These threads execute the actions of the builders for the nodes that are out-of-date. This option has no effect unless the number of concurrent build jobs is larger than one (as set by `-j N` or `--jobs=N` on the command line or `SetOption` in a script).

Using a stack size that is too small may cause stack overflow errors. This usually shows up as segmentation faults that cause scons to abort before building anything. Using a stack size that is too large will cause scons to use more memory than required and may slow down the entire build process. The default value is to use a stack size of 256 kilobytes, which should be appropriate for most uses. You should not need to increase this value unless you encounter stack overflow errors.

`-t`, `--touch`

Ignored for compatibility with GNU Make. (Touching a file to make it appear up-to-date is unnecessary when using **scons**.)

`--taskmastertrace=`*`file`*

Prints trace information to the specified *`file`* about how the internal Taskmaster object evaluates and controls the order in which Nodes are built. A file name of **-** may be used to specify the standard output.

`--tree=`*`type`*`[,`*`type`*`...]`

Prints a tree of the dependencies after each top-level target is built. This prints out some or all of the tree, in various formats, depending on the *`type`* specified:

all

Print the entire dependency tree after each top-level target is built. This prints out the complete dependency tree, including implicit dependencies and ignored dependencies.

derived

Restricts the tree output to only derived (target) files, not source files.

linedraw

Draw the tree output using Unicode line-drawing characters instead of plain ASCII text. This option acts as a modifier to the selected *`type`*(s). If specified alone, without any *`type`*, it behaves as if **all** had been specified.

*Available since **scons** 4.0.*

status

Prints status information for each displayed node.

prune

Prunes the tree to avoid repeating dependency information for nodes that have already been displayed. Any node that has already been displayed will have its name printed in **[square brackets]**, as an indication that the dependencies for that node can be found by searching for the relevant output higher up in the tree.

Multiple *`type`* choices may be specified, separated by commas:

```
# Prints only derived files, with status information:
scons --tree=derived,status

# Prints all dependencies of target, with status information
# and pruning dependencies of already-visited Nodes:
```

```
scons --tree=all,prune,status target
```

-u, --up, --search-up
Walks up the directory structure until an `SConstruct`, `Sconstruct`, `sconstruct`, `SConstruct.py`, `Sconstruct.py` or `sconstruct.py` file is found, and uses that as the top of the directory tree. If no targets are specified on the command line, only targets at or below the current directory will be built.

-U
Works exactly the same way as the `-u` option except for the way default targets are handled. When this option is used and no targets are specified on the command line, all default targets that are defined in the SConscript(s) in the current directory are built, regardless of what directory the resultant targets end up in.

-v, --version
Print the **scons** version, copyright information, list of authors, and any other relevant information. Then exit.

-w, --print-directory
Print a message containing the working directory before and after other processing.

--no-print-directory
Turn off -w, even if it was turned on implicitly.

--warn=*type*, --warn=no-*type*
Enable or disable (with the prefix "no-") warnings (`--warning` is a synonym). *`type`* specifies the type of warnings to be enabled or disabled:

all
All warnings.

cache-version
Warnings about the derived-file cache directory specified by `CacheDir` not using the latest configuration information. These warnings are enabled by default.

cache-write-error
Warnings about errors trying to write a copy of a built file to a specified derived-file cache specified by `CacheDir`. These warnings are disabled by default.

corrupt-sconsign
Warnings about unfamiliar signature data in `.sconsign` files. These warnings are enabled by default.

dependency
Warnings about dependencies. These warnings are disabled by default.

deprecated
Warnings about use of currently deprecated features. These warnings are enabled by default. Not all deprecation warnings can be disabled with the `--warn=no-deprecated` option as some deprecated features which are late in the deprecation cycle may have been designated as mandatory warnings, and these will still display. Warnings for certain deprecated features may also be enabled or disabled individually; see below.

duplicate-environment
Warnings about attempts to specify a build of a target with two different construction environments that use the same action. These warnings are enabled by default.

fortran-cxx-mix
Warnings about linking Fortran and C++ object files in a single executable, which can yield unpredictable behavior with some compilers.

future-deprecated
Warnings about features that will be deprecated in the future. Such warnings are disabled by default. Enabling future deprecation warnings is recommended for projects that redistribute SCons configurations for other users to build, so that the project can be warned as soon as possible about to-be-deprecated features that may require changes to the configuration.

link
Warnings about link steps.

misleading-keywords
Warnings about the use of two commonly misspelled keywords *`targets`* and *`sources`* to `Builder` calls. The correct spelling is the singular form, even though *`target`* and *`source`* can themselves refer to lists of names or nodes.

tool-qt-deprecated
Warnings about the `qt` tool being deprecated. These warnings are disabled by default for the first phase of deprecation. Enable to be reminded about use of this tool module. *Available since SCons 4.3.*

missing-sconscript
Warnings about missing SConscript files. These warnings are enabled by default.

no-object-count
Warnings about the `--debug=object` feature not working when **scons** is run with the Python `-O` option or from optimized Python (.pyo) modules.

no-parallel-support
Warnings about the version of Python not being able to support parallel builds when the `-j` option is used. These warnings are enabled by default.

python-version
Warnings about running SCons with a deprecated version of Python. These warnings are enabled by default.

reserved-variable
Warnings about attempts to set the reserved construction variable names `$CHANGED_SOURCES`, `$CHANGED_TARGETS`, `$TARGET`, `$TARGETS`, `$SOURCE`, `$SOURCES`, `$UNCHANGED_SOURCES` or `$UNCHANGED_TARGETS`. These warnings are disabled by default.

stack-size
Warnings about requests to set the stack size that could not be honored. These warnings are enabled by default.

target_not_build
Warnings about a build rule not building the expected targets. These warnings are disabled by default.

`-Y` *`repository`*, `--repository=`*`repository`*, `--srcdir=`*`repository`*
Search the specified *`repository`* for any input and target files not found in the local directory hierarchy. Multiple `-Y` options may be specified, in which case the repositories are searched in the order specified.

SCONSCRIPT FILE REFERENCE

SConscript Files

The build configuration is described by one or more files, known as SConscript files. There must be at least one file for a valid build (**scons** will quit if it does not find one). **scons** by default looks for this file by the name `SConstruct` in the directory from which you run **scons**, though if necessary, also looks for alternative file names `Sconstruct`, `sconstruct`, `SConstruct.py`, `Sconstruct.py` and `sconstruct.py` in that order. A different file name (which can include a pathname part) may be specified via the `-f` option. Except for the SConstruct file, these files are not searched for automatically; you add additional configuration files to the build by calling the `SConscript` function. This allows parts of the build to be conditionally included or excluded at run-time depending on how **scons** is invoked.

Each SConscript file in a build configuration is invoked independently in a separate context. This provides necessary isolation so that different parts of the build don't accidentally step on each other. You have to be explicit about sharing information, by using the `Export` function or the `exports` argument to the `SConscript` function, as well as the `Return` function in a called SConscript file, and comsume shared information by using the `Import` function.

The following sections describe the various SCons facilities that can be used in SConscript files. Quick links:

Construction Environments
Tools
Builder Methods
Methods and Functions to do Things
SConscript Variables
Construction Variables
Configure Contexts
Command-Line Construction Variables
Node Objects

Construction Environments

A *Construction Environment* is the basic means by which you communicate build information to SCons. A new construction environment is created using the `Environment` function:

```
env = Environment()
```

Construction environment attributes called *Construction Variables* may be set either by specifying them as keyword arguments when the object is created or by assigning them a value after the object is created. These two are nominally equivalent:

```
env = Environment(FOO='foo')
env['FOO'] = 'foo'
```

Note that certain settings which affect tool detection are referenced only when the tools are initializided, so you either need either to supply them as part of the call to `Environment`, or defer tool initialization. For example, initializing the Microsoft Visual C++ version you wish to use:

```
# initializes msvc to v14.1
env = Environment(MSVC_VERSION="14.1")

env = Environment()
# msvc tool was initialized to default, does not reinitialize
env['MSVC_VERSION'] = "14.1"

env = Environment(tools=[])
env['MSVC_VERSION'] = "14.1"
# msvc tool initialization was deferred, so will pick up new value
env.Tool('default')
```

As a convenience, construction variables may also be set or modified by the *`parse_flags`* keyword argument during object creation, which has the effect of the `env.MergeFlags` method being applied to the argument value after all other processing is completed. This is useful either if the exact content of the flags is unknown (for example, read from a control file) or if the flags need to be distributed to a number of construction variables. `env.ParseFlags` describes how these arguments are distributed to construction variables.

```
env = Environment(parse_flags='-Iinclude -DEBUG -lm')
```

This example adds 'include' to the `CPPPATH` construction variable, 'EBUG' to `CPPDEFINES`, and 'm' to `LIBS`.

An existing construction environment can be duplicated by calling the `env.Clone` method. Without arguments, it will be a copy with the same settings. Otherwise, `env.Clone` takes the same arguments as `Environment`, and uses the arguments to create a modified copy.

SCons provides a special construction environment called the *Default Environment*. The default environment is used only for global functions, that is, construction activities called without the context of a regular construction environment. See `DefaultEnvironment` for more information.

By default, a new construction environment is initialized with a set of builder methods and construction variables that are appropriate for the current platform. The optional *`platform`* keyword argument may be used to specify that the construction environment should be initialized for a different platform:

```
env = Environment(platform='cygwin')
```

Specifying a platform initializes the appropriate construction variables in the environment to use and generate file names with prefixes and suffixes appropriate for that platform.

Note that the `win32` platform adds the `SystemDrive` and `SystemRoot` variables from the user's external environment to the construction environment's `ENV` dictionary. This is so that any executed commands that use sockets to connect with other systems will work on Windows systems.

The *`platform`* argument may be a string value representing one of the pre-defined platforms (`aix`, `cygwin`, `darwin`, `hpux`, `irix`, `os2`, `posix`, `sunos` or `win32`), or it may be be a callable platform object returned by a call to `Platform` selecting a pre-defined platform, or it may be a user-supplied callable, in which case the `Environment` method will call it to update the new construction environment:

```
def my_platform(env):
    env['VAR'] = 'xyzzy'

env = Environment(platform=my_platform)
```

Note that supplying a non-default platform or custom fuction for initialization may bypass settings that should happen for the host system and should be used with care. It is most useful in the case where the platform is an alternative for the one that would be auto-detected, such as `platform="cygwin"` on a system which would otherwise identify as `win32`.

The optional *`tools`* and *`toolpath`* keyword arguments affect the way tools available to the environment are initialized. See the section called “Tools” for details.

The optional *`variables`* keyword argument allows passing a Variables object which will be used in the initialization of the construction environment See the section called “Command-Line Construction Variables” for details.

Tools

SCons has a large number of predefined tools (more properly, *tool specifications*) which are used to help initialize the construction environment. An SCons tool is only responsible for setup. For example, if the `SConscript` file declares the need to construct an object file from a C-language source file by calling the `Object` builder, then a tool representing an available C compiler needs to have run first, to set up that builder and all the construction variables it needs in the associated construction environment; the tool itself is not called in the process of the build. Normally this happens invisibly: **scons** has per-platform lists of default tools, and it runs through those tools, calling the ones which are actually applicable, skipping those where necessary programs are not installed on the build system, or other preconditions are not met.

A specific set of tools with which to initialize an environment when creating it may be specified using the optional keyword argument *`tools`*, which takes a list of tool names. This is useful to override the defaults, to specify non-default built-in tools, and to supply added tools:

```
env = Environment(tools=['msvc', 'lex'])
```

Tools can also be directly called by using the `Tool` method (see below).

The *`tools`* argument overrides the default tool list, it does not add to it, so be sure to include all the tools you need. For example if you are building a c/c++ program you must specify a tool for at least a compiler and a linker, as in `tools=['clang', 'link']`. The tool name `'default'` can be used to retain the default list.

If no *`tools`* argument is specified, or if *`tools`* includes `'default'`, then **scons** will auto-detect usable tools, using the execution environment value of `PATH` (that is, *`env`*`['ENV']['PATH']` - the external evironment `PATH` from `os.environ` is *not* used) for looking up any backing programs, and the platform name in effect to determine the default tools for that platform. Changing the `PATH` variable after the construction environment is constructed will not cause the tools to be re-detected.

Additional tools can be added to a project either by placing them in a `site_tools` subdirectory of a site directory, or in a custom location specified to **scons** by giving the *`toolpath`* keyword argument. *`toolpath`* also takes a list as its value:

```
env = Environment(tools=['default', 'foo'], toolpath=['tools'])
```

This looks for a tool specification module `foo.py` in directory `tools` and in the standard locations, as well as using the ordinary default tools for the platform.

Directories specified via *`toolpath`* are prepended to the existing tool path. The default tool path is any `site_tools` directories, so tools in a specified *`toolpath`* take priority, followed by tools in a `site_tools` directory, followed by built-in tools. For example, adding a tool specification module `gcc.py` to the toolpath directory would override the built-in `gcc` tool. The tool path is stored in the environment and will be used by subsequent calls to the `Tool` method, as well as by `env.Clone`.

```
base = Environment(toolpath=['custom_path'])
derived = base.Clone(tools=['custom_tool'])
derived.CustomBuilder()
```

A tool specification module must include two functions:

`generate(`*`env, **kwargs`*`)`

Modifies the environment referenced by *`env`* to set up variables so that the facilities represented by the tool can be executed. It may use any keyword arguments that the user supplies in *`kwargs`* to vary its initialization.

`exists(`*`env`*`)`

Return `True` if the tool can be called in the context of *`env`*. Usually this means looking up one or more known programs using the `PATH` from the supplied *`env`*, but the tool can make the "exists" decision in any way it chooses.

Note

At the moment, user-added tools do not automatically have their `exists` function called. As a result, it is recommended that the `generate` function be defensively coded - that is, do not rely on any necessary existence checks already having been performed. This is expected to be a temporary limitation, and the `exists` function should still be provided.

The elements of the *`tools`* list may also be functions or callable objects, in which case the `Environment` method will call those objects to update the new construction environment (see `Tool` for more details):

```
def my_tool(env):
    env['XYZZY'] = 'xyzzy'

env = Environment(tools=[my_tool])
```

The individual elements of the *`tools`* list may also themselves be lists or tuples of the form (`toolname`, `kw_dict`). SCons searches for the *`toolname`* specification file as described above, and passes *`kw_dict`*, which must be a dictionary, as keyword arguments to the tool's `generate` function. The `generate` function can use the arguments to modify the tool's behavior by setting up the environment in different ways or otherwise changing its initialization.

```
# in tools/my_tool.py:
def generate(env, **kwargs):
  # Sets MY_TOOL to the value of keyword 'arg1' '1' if not supplied
  env['MY_TOOL'] = kwargs.get('arg1', '1')

def exists(env):
  return True

# in SConstruct:
env = Environment(tools=['default', ('my_tool', {'arg1': 'abc'})],
                  toolpath=['tools'])
```

The tool specification (`my_tool` in the example) can use the `PLATFORM` variable from the construction environment it is passed to customize the tool for different platforms.

Tools can be "nested" - that is, they can be located within a subdirectory in the toolpath. A nested tool name uses a dot to represent a directory separator

```
# namespaced builder
env = Environment(ENV=os.environ.copy(), tools=['SubDir1.SubDir2.SomeTool'])
env.SomeTool(targets, sources)

# Search Paths
# SCons\Tool\SubDir1\SubDir2\SomeTool.py
# SCons\Tool\SubDir1\SubDir2\SomeTool\__init__.py
# .\site_scons\site_tools\SubDir1\SubDir2\SomeTool.py
# .\site_scons\site_tools\SubDir1\SubDir2\SomeTool\__init__.py
```

SCons supports the following tool specifications out of the box:

386asm
Sets construction variables for the 386ASM assembler for the Phar Lap ETS embedded operating system.

Sets: `$AS`, `$ASCOM`, `$ASFLAGS`, `$ASPPCOM`, `$ASPPFLAGS`.

Uses: `$CC`, `$CPPFLAGS`, `$_CPPDEFFLAGS`, `$_CPPINCFLAGS`.

aixc++

Sets construction variables for the IMB xlc / Visual Age C++ compiler.

Sets: `$CXX`, `$CXXVERSION`, `$SHCXX`, `$SHOBJSUFFIX`.

aixcc

Sets construction variables for the IBM xlc / Visual Age C compiler.

Sets: `$CC`, `$CCVERSION`, `$SHCC`.

aixf77

Sets construction variables for the IBM Visual Age f77 Fortran compiler.

Sets: `$F77`, `$SHF77`.

aixlink

Sets construction variables for the IBM Visual Age linker.

Sets: `$LINKFLAGS`, `$SHLIBSUFFIX`, `$SHLINKFLAGS`.

applelink

Sets construction variables for the Apple linker (similar to the GNU linker).

Sets: `$APPLELINK_COMPATIBILITY_VERSION`, `$APPLELINK_CURRENT_VERSION`, `$APPLELINK_NO_COMPATIBILITY_VERSION`, `$APPLELINK_NO_CURRENT_VERSION`, `$FRAMEWORKPATHPREFIX`, `$LDMODULECOM`, `$LDMODULEFLAGS`, `$LDMODULEPREFIX`, `$LDMODULESUFFIX`, `$LINKCOM`, `$SHLINKCOM`, `$SHLINKFLAGS`, `$_APPLELINK_COMPATIBILITY_VERSION`, `$_APPLELINK_CURRENT_VERSION`, `$_FRAMEWORKPATH`, `$_FRAMEWORKS`.

Uses: `$FRAMEWORKSFLAGS`.

ar

Sets construction variables for the ar library archiver.

Sets: `$AR`, `$ARCOM`, `$ARFLAGS`, `$LIBPREFIX`, `$LIBSUFFIX`, `$RANLIB`, `$RANLIBCOM`, `$RANLIBFLAGS`.

as

Sets construction variables for the as assembler.

Sets: `$AS`, `$ASCOM`, `$ASFLAGS`, `$ASPPCOM`, `$ASPPFLAGS`.

Uses: `$CC`, `$CPPFLAGS`, `$_CPPDEFFLAGS`, `$_CPPINCFLAGS`.

bcc32

Sets construction variables for the bcc32 compiler.

Sets: `$CC`, `$CCCOM`, `$CCFLAGS`, `$CFILESUFFIX`, `$CFLAGS`, `$CPPDEFPREFIX`, `$CPPDEFSUFFIX`, `$INCPREFIX`, `$INCSUFFIX`, `$SHCC`, `$SHCCCOM`, `$SHCCFLAGS`, `$SHCFLAGS`, `$SHOBJSUFFIX`.

Uses: `$_CPPDEFFLAGS`, `$_CPPINCFLAGS`.

cc

Sets construction variables for generic POSIX C compilers.

Sets: `$CC`, `$CCCOM`, `$CCFLAGS`, `$CFILESUFFIX`, `$CFLAGS`, `$CPPDEFPREFIX`, `$CPPDEFSUFFIX`, `$FRAMEWORKPATH`, `$FRAMEWORKS`, `$INCPREFIX`, `$INCSUFFIX`, `$SHCC`, `$SHCCCOM`, `$SHCCFLAGS`, `$SHCFLAGS`, `$SHOBJSUFFIX`.

Uses: `$CCCOMSTR`, `$PLATFORM`, `$SHCCCOMSTR`.

clang

Set construction variables for the Clang C compiler.

Sets: `$CC`, `$CCVERSION`, `$SHCCFLAGS`.

clangxx

Set construction variables for the Clang C++ compiler.

Sets: `$CXX`, `$CXXVERSION`, `$SHCXXFLAGS`, `$SHOBJSUFFIX`, `$STATIC_AND_SHARED_OBJECTS_ARE_THE_SAME`.

compilation_db

Sets up `CompilationDatabase` builder which generates a clang tooling compatible compilation database.

Sets: `$COMPILATIONDB_COMSTR`, `$COMPILATIONDB_PATH_FILTER`, `$COMPILATIONDB_USE_ABSPATH`.

cvf

Sets construction variables for the Compaq Visual Fortran compiler.

Sets: `$FORTRAN`, `$FORTRANCOM`, `$FORTRANMODDIR`, `$FORTRANMODDIRPREFIX`, `$FORTRANMODDIRSUFFIX`, `$FORTRANPPCOM`, `$OBJSUFFIX`, `$SHFORTRANCOM`, `$SHFORTRANPPCOM`.

Uses: `$CPPFLAGS`, `$FORTRANFLAGS`, `$SHFORTRANFLAGS`, `$_CPPDEFFLAGS`, `$_FORTRANINCFLAGS`, `$_FORTRANMODFLAG`.

cXX

Sets construction variables for generic POSIX C++ compilers.

Sets: `$CPPDEFPREFIX`, `$CPPDEFSUFFIX`, `$CXX`, `$CXXCOM`, `$CXXFILESUFFIX`, `$CXXFLAGS`, `$INCPREFIX`, `$INCSUFFIX`, `$OBJSUFFIX`, `$SHCXX`, `$SHCXXCOM`, `$SHCXXFLAGS`, `$SHOBJSUFFIX`.

Uses: `$CXXCOMSTR`, `$SHCXXCOMSTR`.

cyglink

Set construction variables for cygwin linker/loader.

Sets: `$IMPLIBPREFIX`, `$IMPLIBSUFFIX`, `$LDMODULEVERSIONFLAGS`, `$LINKFLAGS`, `$RPATHPREFIX`, `$RPATHSUFFIX`, `$SHLIBPREFIX`, `$SHLIBSUFFIX`, `$SHLIBVERSIONFLAGS`, `$SHLINKCOM`, `$SHLINKFLAGS`, `$_LDMODULEVERSIONFLAGS`, `$_SHLIBVERSIONFLAGS`.

default

Sets construction variables for a default list of Tool modules. Use **default** in the tools list to retain the original defaults, since the *`tools`* parameter is treated as a literal statement of the tools to be made available in that construction environment, not an addition.

The list of tools selected by default is not static, but is dependent both on the platform and on the software installed on the platform. Some tools will not initialize if an underlying command is not found, and some tools are selected from a list of choices on a first-found basis. The finished tool list can be examined by inspecting the `$TOOLS` construction variable in the construction environment.

On all platforms, the tools from the following list are selected if their respective conditions are met: filesystem;, wix, `lex`, `yacc`, `rpcgen`, `swig`, `jar`, `javac`, `javah`, `rmic`, `dvipdf`, `dvips`, `gs`, `tex`, `latex`, `pdflatex`, `pdftex`, `tar`, `zip`, `textfile`.

On Linux systems, the default tools list selects (first-found): a C compiler from `gcc`, `intelc`, `icc`, `cc`; a C ++ compiler from `g++`, `intelc`, `icc`, `cXX`; an assembler from `gas`, `nasm`, `masm`; a linker from `gnulink`, `ilink`; a Fortran compiler from `gfortran`, `g77`, `ifort`, `ifl`, `f95`, `f90`, `f77`; and a static archiver `ar`. It also selects all found from the list `m4` rpm.

On Windows systems, the default tools list selects (first-found): a C compiler from `msvc`, `mingw`, `gcc`, `intelc`, `icl`, `icc`, `cc`, `bcc32`; a C++ compiler from `msvc`, `intelc`, `icc`, `g++`, `cXX`, `bcc32`; an assembler from `masm`, `nasm`, `gas`, `386asm`; a linker from `mslink`, `gnulink`, `ilink`, `linkloc`, `ilink32`; a Fortran compiler from `gfortran`, `g77`, `ifl`, `cvf`, `f95`, `f90`, `fortran`; and a static archiver from `mslib`, `ar`, `tlib`; It also selects all found from the list `msvs`, `midl`.

On MacOS systems, the default tools list selects (first-found): a C compiler from `gcc`, `cc`; a C++ compiler from `g++`, `cXX`; an assembler `as`; a linker from `applelink`, `gnulink`; a Fortran compiler from `gfortran`, `f95`, `f90`, `g77`; and a static archiver `ar`. It also selects all found from the list `m4`, rpm.

Default lists for other platforms can be found by examining the **scons** source code (see `SCons/Tool/__init__.py`).

dmd

Sets construction variables for D language compiler DMD.

Sets: `$DC`, `$DCOM`, `$DDEBUG`, `$DDEBUGPREFIX`, `$DDEBUGSUFFIX`, `$DFILESUFFIX`, `$DFLAGPREFIX`, `$DFLAGS`, `$DFLAGSUFFIX`, `$DINCPREFIX`, `$DINCSUFFIX`, `$DLIB`, `$DLIBCOM`, `$DLIBDIRPREFIX`, `$DLIBDIRSUFFIX`, `$DLIBFLAGPREFIX`, `$DLIBFLAGSUFFIX`, `$DLIBLINKPREFIX`, `$DLIBLINKSUFFIX`, `$DLINK`, `$DLINKCOM`, `$DLINKFLAGPREFIX`, `$DLINKFLAGS`, `$DLINKFLAGSUFFIX`, `$DPATH`, `$DRPATHPREFIX`, `$DRPATHSUFFIX`, `$DVERPREFIX`, `$DVERSIONS`, `$DVERSUFFIX`, `$SHDC`, `$SHDCOM`, `$SHDLIBVERSIONFLAGS`, `$SHDLINK`, `$SHDLINKCOM`, `$SHDLINKFLAGS`.

docbook

This tool tries to make working with Docbook in SCons a little easier. It provides several toolchains for creating different output formats, like HTML or PDF. Contained in the package is a distribution of the Docbook XSL stylesheets as of version 1.76.1. As long as you don't specify your own stylesheets for customization, these official versions are picked as default...which should reduce the inevitable setup hassles for you.

Implicit dependencies to images and XIncludes are detected automatically if you meet the HTML requirements. The additional stylesheet `utils/xmldepend.xsl` by Paul DuBois is used for this purpose.

Note, that there is no support for XML catalog resolving offered! This tool calls the XSLT processors and PDF renderers with the stylesheets you specified, that's it. The rest lies in your hands and you still have to know what you're doing when resolving names via a catalog.

For activating the tool "docbook", you have to add its name to the Environment constructor, like this

```
env = Environment(tools=['docbook'])
```

On its startup, the `docbook` tool tries to find a required `xsltproc` processor, and a PDF renderer, e.g. fop. So make sure that these are added to your system's environment `PATH` and can be called directly without specifying their full path.

For the most basic processing of Docbook to HTML, you need to have installed

- the Python `lxml` binding to `libxml2`, or

- a standalone XSLT processor, currently detected are xsltproc, saxon, saxon-xslt and xalan.

Rendering to PDF requires you to have one of the applications fop or xep installed.

Creating a HTML or PDF document is very simple and straightforward. Say

```
env = Environment(tools=['docbook'])
env.DocbookHtml('manual.html', 'manual.xml')
env.DocbookPdf('manual.pdf', 'manual.xml')
```

to get both outputs from your XML source `manual.xml`. As a shortcut, you can give the stem of the filenames alone, like this:

```
env = Environment(tools=['docbook'])
env.DocbookHtml('manual')
env.DocbookPdf('manual')
```

and get the same result. Target and source lists are also supported:

```
env = Environment(tools=['docbook'])
env.DocbookHtml(['manual.html','reference.html'], ['manual.xml','reference.xml'])
```

or even

```
env = Environment(tools=['docbook'])
env.DocbookHtml(['manual','reference'])
```

Important

Whenever you leave out the list of sources, you may not specify a file extension! The Tool uses the given names as file stems, and adds the suffixes for target and source files accordingly.

The rules given above are valid for the Builders `DocbookHtml`, `DocbookPdf`, `DocbookEpub`, `DocbookSlidesPdf` and `DocbookXInclude`. For the `DocbookMan` transformation you can specify a target name, but the actual output names are automatically set from the `refname` entries in your XML source.

The Builders `DocbookHtmlChunked`, `DocbookHtmlhelp` and `DocbookSlidesHtml` are special, in that:

1. they create a large set of files, where the exact names and their number depend on the content of the source file, and
2. the main target is always named `index.html`, i.e. the output name for the XSL transformation is not picked up by the stylesheets.

As a result, there is simply no use in specifying a target HTML name. So the basic syntax for these builders is always:

```
env = Environment(tools=['docbook'])
env.DocbookHtmlhelp('manual')
```

If you want to use a specific XSL file, you can set the additional *`xsl`* parameter to your Builder call as follows:

```
env.DocbookHtml('other.html', 'manual.xml', xsl='html.xsl')
```

Since this may get tedious if you always use the same local naming for your customized XSL files, e.g. `html.xsl` for HTML and `pdf.xsl` for PDF output, a set of variables for setting the default XSL name is provided. These are:

```
DOCBOOK_DEFAULT_XSL_HTML
DOCBOOK_DEFAULT_XSL_HTMLCHUNKED
DOCBOOK_DEFAULT_XSL_HTMLHELP
DOCBOOK_DEFAULT_XSL_PDF
DOCBOOK_DEFAULT_XSL_EPUB
DOCBOOK_DEFAULT_XSL_MAN
DOCBOOK_DEFAULT_XSL_SLIDESPDF
DOCBOOK_DEFAULT_XSL_SLIDESHTML
```

and you can set them when constructing your environment:

```
env = Environment(
    tools=['docbook'],
    DOCBOOK_DEFAULT_XSL_HTML='html.xsl',
    DOCBOOK_DEFAULT_XSL_PDF='pdf.xsl',
)
env.DocbookHtml('manual')  # now uses html.xsl
```

Sets: `$DOCBOOK_DEFAULT_XSL_EPUB`, `$DOCBOOK_DEFAULT_XSL_HTML`, `$DOCBOOK_DEFAULT_XSL_HTMLCHUNKED`, `$DOCBOOK_DEFAULT_XSL_HTMLHELP`, `$DOCBOOK_DEFAULT_XSL_MAN`, `$DOCBOOK_DEFAULT_XSL_PDF`, `$DOCBOOK_DEFAULT_XSL_SLIDESHTML`, `$DOCBOOK_DEFAULT_XSL_SLIDESPDF`, `$DOCBOOK_FOP`, `$DOCBOOK_FOPCOM`, `$DOCBOOK_FOPFLAGS`, `$DOCBOOK_XMLLINT`, `$DOCBOOK_XMLLINTCOM`, `$DOCBOOK_XMLLINTFLAGS`, `$DOCBOOK_XSLTPROC`, `$DOCBOOK_XSLTPROCCOM`, `$DOCBOOK_XSLTPROCFLAGS`, `$DOCBOOK_XSLTPROCPARAMS`.

Uses: `$DOCBOOK_FOPCOMSTR`, `$DOCBOOK_XMLLINTCOMSTR`, `$DOCBOOK_XSLTPROCCOMSTR`.

dvi

Attaches the `DVI` builder to the construction environment.

dvipdf

Sets construction variables for the dvipdf utility.

Sets: `$DVIPDF`, `$DVIPDFCOM`, `$DVIPDFFLAGS`.

Uses: `$DVIPDFCOMSTR`.

dvips

Sets construction variables for the dvips utility.

Sets: `$DVIPS`, `$DVIPSFLAGS`, `$PSCOM`, `$PSPREFIX`, `$PSSUFFIX`.

Uses: `$PSCOMSTR`.

f03

Set construction variables for generic POSIX Fortran 03 compilers.

Sets: `$F03`, `$F03COM`, `$F03FLAGS`, `$F03PPCOM`, `$SHF03`, `$SHF03COM`, `$SHF03FLAGS`, `$SHF03PPCOM`, `$_F03INCFLAGS`.

Uses: `$F03COMSTR`, `$F03PPCOMSTR`, `$SHF03COMSTR`, `$SHF03PPCOMSTR`.

f08

Set construction variables for generic POSIX Fortran 08 compilers.

Sets: `$F08`, `$F08COM`, `$F08FLAGS`, `$F08PPCOM`, `$SHF08`, `$SHF08COM`, `$SHF08FLAGS`, `$SHF08PPCOM`, `$_F08INCFLAGS`.

Uses: `$F08COMSTR`, `$F08PPCOMSTR`, `$SHF08COMSTR`, `$SHF08PPCOMSTR`.

f77

Set construction variables for generic POSIX Fortran 77 compilers.

Sets: `$F77`, `$F77COM`, `$F77FILESUFFIXES`, `$F77FLAGS`, `$F77PPCOM`, `$F77PPFILESUFFIXES`, `$FORTRAN`, `$FORTRANCOM`, `$FORTRANFLAGS`, `$SHF77`, `$SHF77COM`, `$SHF77FLAGS`, `$SHF77PPCOM`, `$SHFORTRAN`, `$SHFORTRANCOM`, `$SHFORTRANFLAGS`, `$SHFORTRANPPCOM`, `$_F77INCFLAGS`.

Uses: `$F77COMSTR`, `$F77PPCOMSTR`, `$FORTRANCOMSTR`, `$FORTRANPPCOMSTR`, `$SHF77COMSTR`, `$SHF77PPCOMSTR`, `$SHFORTRANCOMSTR`, `$SHFORTRANPPCOMSTR`.

f90

Set construction variables for generic POSIX Fortran 90 compilers.

Sets: `$F90`, `$F90COM`, `$F90FLAGS`, `$F90PPCOM`, `$SHF90`, `$SHF90COM`, `$SHF90FLAGS`, `$SHF90PPCOM`, `$_F90INCFLAGS`.

Uses: `$F90COMSTR`, `$F90PPCOMSTR`, `$SHF90COMSTR`, `$SHF90PPCOMSTR`.

f95

Set construction variables for generic POSIX Fortran 95 compilers.

Sets: `$F95`, `$F95COM`, `$F95FLAGS`, `$F95PPCOM`, `$SHF95`, `$SHF95COM`, `$SHF95FLAGS`, `$SHF95PPCOM`, `$_F95INCFLAGS`.

Uses: `$F95COMSTR`, `$F95PPCOMSTR`, `$SHF95COMSTR`, `$SHF95PPCOMSTR`.

fortran

Set construction variables for generic POSIX Fortran compilers.

Sets: `$FORTRAN`, `$FORTRANCOM`, `$FORTRANFLAGS`, `$SHFORTRAN`, `$SHFORTRANCOM`, `$SHFORTRANFLAGS`, `$SHFORTRANPPCOM`.

Uses: `$FORTRANCOMSTR`, `$FORTRANPPCOMSTR`, `$SHFORTRANCOMSTR`, `$SHFORTRANPPCOMSTR`.

g++

Set construction variables for the g++ C++ compiler.

Sets: `$CXX`, `$CXXVERSION`, `$SHCXXFLAGS`, `$SHOBJSUFFIX`.

g77

Set construction variables for the g77 Fortran compiler. Calls the `f77` Tool module to set variables.

gas

Sets construction variables for the gas assembler. Calls the `as` tool.

Sets: `$AS`.

gcc

Set construction variables for the gcc C compiler.

Sets: `$CC`, `$CCVERSION`, `$SHCCFLAGS`.

gdc

Sets construction variables for the D language compiler GDC.

Sets: `$DC`, `$DCOM`, `$DDEBUG`, `$DDEBUGPREFIX`, `$DDEBUGSUFFIX`, `$DFILESUFFIX`, `$DFLAGPREFIX`, `$DFLAGS`, `$DFLAGSUFFIX`, `$DINCPREFIX`, `$DINCSUFFIX`, `$DLIB`, `$DLIBCOM`, `$DLIBDIRPREFIX`, `$DLIBDIRSUFFIX`, `$DLIBFLAGPREFIX`, `$DLIBFLAGSUFFIX`, `$DLIBLINKPREFIX`, `$DLIBLINKSUFFIX`, `$DLINK`, `$DLINKCOM`, `$DLINKFLAGPREFIX`, `$DLINKFLAGS`, `$DLINKFLAGSUFFIX`, `$DPATH`, `$DRPATHPREFIX`, `$DRPATHSUFFIX`, `$DVERPREFIX`, `$DVERSIONS`, `$DVERSUFFIX`, `$SHDC`, `$SHDCOM`, `$SHDLIBVERSIONFLAGS`, `$SHDLINK`, `$SHDLINKCOM`, `$SHDLINKFLAGS`.

gettext

This is actually a toolset, which supports internationalization and localization of software being constructed with SCons. The toolset loads following tools:

- `xgettext` - to extract internationalized messages from source code to `POT` file(s),
- `msginit` - may be optionally used to initialize `PO` files,
- `msgmerge` - to update `PO` files, that already contain translated messages,
- `msgfmt` - to compile textual `PO` file to binary installable `MO` file.

When you enable `gettext`, it internally loads all abovementioned tools, so you're encouraged to see their individual documentation.

Each of the above tools provides its own builder(s) which may be used to perform particular activities related to software internationalization. You may be however interested in *top-level* `Translate` builder.

To use `gettext` tools add `'gettext'` tool to your environment:

```
env = Environment( tools = ['default', 'gettext'] )
```

gfortran

Sets construction variables for the GNU F95/F2003 GNU compiler.

Sets: `$F77`, `$F90`, `$F95`, `$FORTRAN`, `$SHF77`, `$SHF77FLAGS`, `$SHF90`, `$SHF90FLAGS`, `$SHF95`, `$SHF95FLAGS`, `$SHFORTRAN`, `$SHFORTRANFLAGS`.

gnulink

Set construction variables for GNU linker/loader.

Sets: `$LDMODULEVERSIONFLAGS`, `$RPATHPREFIX`, `$RPATHSUFFIX`, `$SHLIBVERSIONFLAGS`, `$SHLINKFLAGS`, `$_LDMODULESONAME`, `$_SHLIBSONAME`.

gs

This Tool sets the required construction variables for working with the Ghostscript software. It also registers an appropriate Action with the `PDF` Builder, such that the conversion from PS/EPS to PDF happens automatically for the TeX/LaTeX toolchain. Finally, it adds an explicit `Gs` Builder for Ghostscript to the environment.

Sets: `$GS`, `$GSCOM`, `$GSFLAGS`.

Uses: `$GSCOMSTR`.

hpc++

Set construction variables for the compilers aCC on HP/UX systems.

hpcc

Set construction variables for aCC compilers on HP/UX systems. Calls the `cXX` tool for additional variables.

Sets: `$CXX`, `$CXXVERSION`, `$SHCXXFLAGS`.

hplink

Sets construction variables for the linker on HP/UX systems.

Sets: `$LINKFLAGS`, `$SHLIBSUFFIX`, `$SHLINKFLAGS`.

icc

Sets construction variables for the icc compiler on OS/2 systems.

Sets: `$CC`, `$CCCOM`, `$CFILESUFFIX`, `$CPPDEFPREFIX`, `$CPPDEFSUFFIX`, `$CXXCOM`, `$CXXFILESUFFIX`, `$INCPREFIX`, `$INCSUFFIX`.

Uses: `$CCFLAGS`, `$CFLAGS`, `$CPPFLAGS`, `$_CPPDEFFLAGS`, `$_CPPINCFLAGS`.

icl

Sets construction variables for the Intel C/C++ compiler. Calls the `intelc` Tool module to set its variables.

ifl

Sets construction variables for the Intel Fortran compiler.

Sets: `$FORTRAN`, `$FORTRANCOM`, `$FORTRANPPCOM`, `$SHFORTRANCOM`, `$SHFORTRANPPCOM`.

Uses: `$CPPFLAGS`, `$FORTRANFLAGS`, `$_CPPDEFFLAGS`, `$_FORTRANINCFLAGS`.

ifort

Sets construction variables for newer versions of the Intel Fortran compiler for Linux.

Sets: `$F77`, `$F90`, `$F95`, `$FORTRAN`, `$SHF77`, `$SHF77FLAGS`, `$SHF90`, `$SHF90FLAGS`, `$SHF95`, `$SHF95FLAGS`, `$SHFORTRAN`, `$SHFORTRANFLAGS`.

ilink

Sets construction variables for the ilink linker on OS/2 systems.

Sets: `$LIBDIRPREFIX`, `$LIBDIRSUFFIX`, `$LIBLINKPREFIX`, `$LIBLINKSUFFIX`, `$LINK`, `$LINKCOM`, `$LINKFLAGS`.

ilink32

Sets construction variables for the Borland ilink32 linker.

Sets: `$LIBDIRPREFIX`, `$LIBDIRSUFFIX`, `$LIBLINKPREFIX`, `$LIBLINKSUFFIX`, `$LINK`, `$LINKCOM`, `$LINKFLAGS`.

install

Sets construction variables for file and directory installation.

Sets: `$INSTALL`, `$INSTALLSTR`.

intelc

Sets construction variables for the Intel C/C++ compiler (Linux and Windows, version 7 and later). Calls the `gcc` or `msvc` (on Linux and Windows, respectively) tool to set underlying variables.

Sets: `$AR`, `$CC`, `$CXX`, `$INTEL_C_COMPILER_VERSION`, `$LINK`.

jar

Sets construction variables for the jar utility.

Sets: `$JAR`, `$JARCOM`, `$JARFLAGS`, `$JARSUFFIX`.

Uses: `$JARCOMSTR`.

javac

Sets construction variables for the javac compiler.

Sets: `$JAVABOOTCLASSPATH`, `$JAVAC`, `$JAVACCOM`, `$JAVACFLAGS`, `$JAVACLASSPATH`, `$JAVACLASSSUFFIX`, `$JAVAINCLUDES`, `$JAVASOURCEPATH`, `$JAVASUFFIX`.

Uses: `$JAVACCOMSTR`.

javah

Sets construction variables for the javah tool.

Sets: `$JAVACLASSSUFFIX`, `$JAVAH`, `$JAVAHCOM`, `$JAVAHFLAGS`.

Uses: `$JAVACLASSPATH`, `$JAVAHCOMSTR`.

latex

Sets construction variables for the latex utility.

Sets: `$LATEX`, `$LATEXCOM`, `$LATEXFLAGS`.

Uses: `$LATEXCOMSTR`.

ldc

Sets construction variables for the D language compiler LDC2.

Sets: `$DC`, `$DCOM`, `$DDEBUG`, `$DDEBUGPREFIX`, `$DDEBUGSUFFIX`, `$DFILESUFFIX`, `$DFLAGPREFIX`, `$DFLAGS`, `$DFLAGSUFFIX`, `$DINCPREFIX`, `$DINCSUFFIX`, `$DLIB`, `$DLIBCOM`, `$DLIBDIRPREFIX`, `$DLIBDIRSUFFIX`, `$DLIBFLAGPREFIX`, `$DLIBFLAGSUFFIX`, `$DLIBLINKPREFIX`, `$DLIBLINKSUFFIX`, `$DLINK`, `$DLINKCOM`, `$DLINKFLAGPREFIX`, `$DLINKFLAGS`, `$DLINKFLAGSUFFIX`, `$DPATH`, `$DRPATHPREFIX`, `$DRPATHSUFFIX`, `$DVERPREFIX`, `$DVERSIONS`, `$DVERSUFFIX`, `$SHDC`, `$SHDCOM`, `$SHDLIBVERSIONFLAGS`, `$SHDLINK`, `$SHDLINKCOM`, `$SHDLINKFLAGS`.

lex

Sets construction variables for the lex lexical analyser.

Sets: `$LEX`, `$LEXCOM`, `$LEXFLAGS`, `$LEXUNISTD`.

Uses: `$LEXCOMSTR`.

link

Sets construction variables for generic POSIX linkers. This is a "smart" linker tool which selects a compiler to complete the linking based on the types of source files.

Sets: `$LDMODULE`, `$LDMODULECOM`, `$LDMODULEFLAGS`, `$LDMODULENOVERSIONSYMLINKS`, `$LDMODULEPREFIX`, `$LDMODULESUFFIX`, `$LDMODULEVERSION`, `$LDMODULEVERSIONFLAGS`, `$LIBDIRPREFIX`, `$LIBDIRSUFFIX`, `$LIBLINKPREFIX`, `$LIBLINKSUFFIX`, `$LINK`, `$LINKCOM`, `$LINKFLAGS`, `$SHLIBSUFFIX`, `$SHLINK`, `$SHLINKCOM`, `$SHLINKFLAGS`, `$__LDMODULEVERSIONFLAGS`, `$__SHLIBVERSIONFLAGS`.

Uses: `$LDMODULECOMSTR`, `$LINKCOMSTR`, `$SHLINKCOMSTR`.

linkloc

Sets construction variables for the LinkLoc linker for the Phar Lap ETS embedded operating system.

Sets: `$LIBDIRPREFIX`, `$LIBDIRSUFFIX`, `$LIBLINKPREFIX`, `$LIBLINKSUFFIX`, `$LINK`, `$LINKCOM`, `$LINKFLAGS`, `$SHLINK`, `$SHLINKCOM`, `$SHLINKFLAGS`.

Uses: `$LINKCOMSTR`, `$SHLINKCOMSTR`.

m4

Sets construction variables for the m4 macro processor.

Sets: `$M4`, `$M4COM`, `$M4FLAGS`.

Uses: `$M4COMSTR`.

masm

Sets construction variables for the Microsoft assembler.

Sets: `$AS`, `$ASCOM`, `$ASFLAGS`, `$ASPPCOM`, `$ASPPFLAGS`.

Uses: `$ASCOMSTR`, `$ASPPCOMSTR`, `$CPPFLAGS`, `$_CPPDEFFLAGS`, `$_CPPINCFLAGS`.

midl

Sets construction variables for the Microsoft IDL compiler.

Sets: `$MIDL`, `$MIDLCOM`, `$MIDLFLAGS`.

Uses: `$MIDLCOMSTR`.

mingw

Sets construction variables for MinGW (Minimal Gnu on Windows).

Sets: `$AS`, `$CC`, `$CXX`, `$LDMODULECOM`, `$LIBPREFIX`, `$LIBSUFFIX`, `$OBJSUFFIX`, `$RC`, `$RCCOM`, `$RCFLAGS`, `$RCINCFLAGS`, `$RCINCPREFIX`, `$RCINCSUFFIX`, `$SHCCFLAGS`, `$SHCXXFLAGS`, `$SHLINKCOM`, `$SHLINKFLAGS`, `$SHOBJSUFFIX`, `$WINDOWSDEFPREFIX`, `$WINDOWSDEFSUFFIX`.

Uses: `$RCCOMSTR`, `$SHLINKCOMSTR`.

msgfmt

This scons tool is a part of scons `gettext` toolset. It provides scons interface to **msgfmt(1)** command, which generates binary message catalog (`MO`) from a textual translation description (`PO`).

Sets: `$MOSUFFIX`, `$MSGFMT`, `$MSGFMTCOM`, `$MSGFMTCOMSTR`, `$MSGFMTFLAGS`, `$POSUFFIX`.

Uses: `$LINGUAS_FILE`.

msginit

This scons tool is a part of scons `gettext` toolset. It provides scons interface to **msginit(1)** program, which creates new `PO` file, initializing the meta information with values from user's environment (or options).

Sets: `$MSGINIT`, `$MSGINITCOM`, `$MSGINITCOMSTR`, `$MSGINITFLAGS`, `$POAUTOINIT`, `$POCREATE_ALIAS`, `$POSUFFIX`, `$POTSUFFIX`, `$_MSGINITLOCALE`.

Uses: `$LINGUAS_FILE`, `$POAUTOINIT`, `$POTDOMAIN`.

msgmerge

This scons tool is a part of scons `gettext` toolset. It provides scons interface to **msgmerge(1)** command, which merges two Uniform style `.po` files together.

Sets: `$MSGMERGE`, `$MSGMERGECOM`, `$MSGMERGECOMSTR`, `$MSGMERGEFLAGS`, `$POSUFFIX`, `$POTSUFFIX`, `$POUPDATE_ALIAS`.

Uses: `$LINGUAS_FILE`, `$POAUTOINIT`, `$POTDOMAIN`.

mslib

Sets construction variables for the Microsoft mslib library archiver.

Sets: `$AR`, `$ARCOM`, `$ARFLAGS`, `$LIBPREFIX`, `$LIBSUFFIX`.

Uses: `$ARCOMSTR`.

mslink

Sets construction variables for the Microsoft linker.

Sets: `$LDMODULE`, `$LDMODULECOM`, `$LDMODULEFLAGS`, `$LDMODULEPREFIX`, `$LDMODULESUFFIX`, `$LIBDIRPREFIX`, `$LIBDIRSUFFIX`, `$LIBLINKPREFIX`, `$LIBLINKSUFFIX`, `$LINK`, `$LINKCOM`, `$LINKFLAGS`, `$REGSVR`, `$REGSVRCOM`, `$REGSVRFLAGS`, `$SHLINK`, `$SHLINKCOM`, `$SHLINKFLAGS`, `$WINDOWSDEFPREFIX`, `$WINDOWSDEFSUFFIX`, `$WINDOWSEXPPREFIX`, `$WINDOWSEXPSUFFIX`, `$WINDOWSPROGMANIFESTPREFIX`, `$WINDOWSPROGMANIFESTSUFFIX`, `$WINDOWSSHLIBMANIFESTPREFIX`, `$WINDOWSSHLIBMANIFESTSUFFIX`, `$WINDOWS_INSERT_DEF`.

Uses: `$LDMODULECOMSTR`, `$LINKCOMSTR`, `$REGSVRCOMSTR`, `$SHLINKCOMSTR`.

mssdk

Sets variables for Microsoft Platform SDK and/or Windows SDK. Note that unlike most other Tool modules, mssdk does not set construction variables, but sets the *environment variables* in the environment SCons uses to execute the Microsoft toolchain: `%INCLUDE%`, `%LIB%`, `%LIBPATH%` and `%PATH%`.

Uses: `$MSSDK_DIR`, `$MSSDK_VERSION`, `$MSVS_VERSION`.

msvc

Sets construction variables for the Microsoft Visual C/C++ compiler.

Sets: `$BUILDERS`, `$CC`, `$CCCOM`, `$CCFLAGS`, `$CCPCHFLAGS`, `$CCPDBFLAGS`, `$CFILESUFFIX`, `$CFLAGS`, `$CPPDEFPREFIX`, `$CPPDEFSUFFIX`, `$CXX`, `$CXXCOM`, `$CXXFILESUFFIX`, `$CXXFLAGS`, `$INCPREFIX`, `$INCSUFFIX`, `$OBJPREFIX`, `$OBJSUFFIX`, `$PCHCOM`, `$PCHPDBFLAGS`, `$RC`, `$RCCOM`, `$RCFLAGS`, `$SHCC`, `$SHCCCOM`, `$SHCCFLAGS`, `$SHCFLAGS`, `$SHCXX`, `$SHCXXCOM`, `$SHCXXFLAGS`, `$SHOBJPREFIX`, `$SHOBJSUFFIX`.

Uses: `$CCCOMSTR`, `$CXXCOMSTR`, `$PCH`, `$PCHSTOP`, `$PDB`, `$SHCCCOMSTR`, `$SHCXXCOMSTR`.

msvs

Sets construction variables for Microsoft Visual Studio.

Sets: `$MSVSBUILDCOM`, `$MSVSCLEANCOM`, `$MSVSENCODING`, `$MSVSPROJECTCOM`, `$MSVSREBUILDCOM`, `$MSVSSCONS`, `$MSVSSCONSCOM`, `$MSVSSCONSCRIPT`, `$MSVSSCONSFLAGS`, `$MSVSSOLUTIONCOM`.

mwcc

Sets construction variables for the Metrowerks CodeWarrior compiler.

Sets: `$CC`, `$CCCOM`, `$CFILESUFFIX`, `$CPPDEFPREFIX`, `$CPPDEFSUFFIX`, `$CXX`, `$CXXCOM`, `$CXXFILESUFFIX`, `$INCPREFIX`, `$INCSUFFIX`, `$MWCW_VERSION`, `$MWCW_VERSIONS`, `$SHCC`, `$SHCCCOM`, `$SHCCFLAGS`, `$SHCFLAGS`, `$SHCXX`, `$SHCXXCOM`, `$SHCXXFLAGS`.

Uses: `$CCCOMSTR`, `$CXXCOMSTR`, `$SHCCCOMSTR`, `$SHCXXCOMSTR`.

mwld

Sets construction variables for the Metrowerks CodeWarrior linker.

Sets: `$AR`, `$ARCOM`, `$LIBDIRPREFIX`, `$LIBDIRSUFFIX`, `$LIBLINKPREFIX`, `$LIBLINKSUFFIX`, `$LINK`, `$LINKCOM`, `$SHLINK`, `$SHLINKCOM`, `$SHLINKFLAGS`.

nasm

Sets construction variables for the nasm Netwide Assembler.

Sets: `$AS`, `$ASCOM`, `$ASFLAGS`, `$ASPPCOM`, `$ASPPFLAGS`.

Uses: `$ASCOMSTR`, `$ASPPCOMSTR`.

ninja

Sets up `Ninja` builder which generates a ninja build file, and then optionally runs ninja.

Note

This is an experimental feature.

This functionality is subject to change and/or removal without deprecation cycle.

Sets: `$IMPLICIT_COMMAND_DEPENDENCIES`, `$NINJA_ALIAS_NAME`, `$NINJA_COMPDB_EXPAND`, `$NINJA_DIR`, `$NINJA_DISABLE_AUTO_RUN`, `$NINJA_ENV_VAR_CACHE`, `$NINJA_FILE_NAME`, `$NINJA_FORCE_SCONS_BUILD`, `$NINJA_GENERATED_SOURCE_SUFFIXES`, `$NINJA_MSVC_DEPS_PREFIX`, `$NINJA_POOL`, `$NINJA_REGENERATE_DEPS`, `$NINJA_SYNTAX`, `$_NINJA_REGENERATE_DEPS_FUNC`, `$__NINJA_NO`.

Uses: `$AR`, `$ARCOM`, `$ARFLAGS`, `$CC`, `$CCCOM`, `$CCFLAGS`, `$CXX`, `$CXXCOM`, `$ESCAPE`, `$LINK`, `$LINKCOM`, `$PLATFORM`, `$PRINT_CMD_LINE_FUNC`, `$PROGSUFFIX`, `$RANLIB`, `$RANLIBCOM`, `$SHCCCOM`, `$SHCXXCOM`, `$SHLINK`, `$SHLINKCOM`.

packaging

Sets construction variables for the `Package` Builder. If this tool is enabled, the `--package-type` command-line option is also enabled.

pdf

Sets construction variables for the Portable Document Format builder.

Sets: `$PDFPREFIX`, `$PDFSUFFIX`.

pdflatex

Sets construction variables for the pdflatex utility.

Sets: `$LATEXRETRIES`, `$PDFLATEX`, `$PDFLATEXCOM`, `$PDFLATEXFLAGS`.

Uses: `$PDFLATEXCOMSTR`.

pdftex

Sets construction variables for the pdftex utility.

Sets: `$LATEXRETRIES`, `$PDFLATEX`, `$PDFLATEXCOM`, `$PDFLATEXFLAGS`, `$PDFTEX`, `$PDFTEXCOM`, `$PDFTEXFLAGS`.

Uses: `$PDFLATEXCOMSTR`, `$PDFTEXCOMSTR`.

python

Loads the Python source scanner into the invoking environment. When loaded, the scanner will attempt to find implicit dependencies for any Python source files in the list of sources provided to an Action that uses this environment.

*Available since **scons** 4.0..*

qt

Sets construction variables for building Qt3 applications.

Note

This tool is only suitable for building targeted to Qt3, which is obsolete (*the tool is deprecated since 4.3*). There are contributed tools for Qt4 and Qt5, see https://github.com/SCons/scons-contrib [https://github.com/SCons/scons-contrib]. Qt4 has also passed end of life for standard support (in Dec 2015).

Note paths for these construction variables are assembled using the `os.path.join` method so they will have the appropriate separator at runtime, but are listed here in the various entries only with the `'/'` separator for simplicity.

In addition, the construction variables `$CPPPATH`, `$LIBPATH` and `$LIBS` may be modified and the variables `$PROGEMITTER`, `$SHLIBEMITTER` and `$LIBEMITTER` are modified. Because the build-performance is affected when using this tool, you have to explicitly specify it at Environment creation:

```
Environment(tools=['default','qt'])
```

The `qt` tool supports the following operations:

Automatic moc file generation from header files. You do not have to specify moc files explicitly, the tool does it for you. However, there are a few preconditions to do so: Your header file must have the same filebase as your implementation file and must stay in the same directory. It must have one of the suffixes `.h`, `.hpp`, `.H`, `.hxx`, `.hh`. You can turn off automatic moc file generation by setting `$QT_AUTOSCAN` to `False`. See also the corresponding `Moc` Builder.

Automatic moc file generation from C++ files. As described in the Qt documentation, include the moc file at the end of the C++ file. Note that you have to include the file, which is generated by the transformation `${QT_MOCCXXPREFIX}<basename>${QT_MOCCXXSUFFIX}`, by default `<basename>.mo`. A warning is generated after building the moc file if you do not include the correct file. If you are using `VariantDir`, you may need to specify *`duplicate=True`*. You can turn off automatic moc file generation by setting `$QT_AUTOSCAN` to `False`. See also the corresponding `Moc` Builder.

Automatic handling of .ui files. The implementation files generated from `.ui` files are handled much the same as yacc or lex files. Each **.ui** file given as a source of `Program`, `Library` or `SharedLibrary` will generate three files: the declaration file, the implementation file and a moc file. Because there are also generated headers, you may need to specify *`duplicate=True`* in calls to `VariantDir`. See also the corresponding `Uic` Builder.

Sets: `$QTDIR`, `$QT_AUTOSCAN`, `$QT_BINPATH`, `$QT_CPPPATH`, `$QT_LIB`, `$QT_LIBPATH`, `$QT_MOC`, `$QT_MOCCXXPREFIX`, `$QT_MOCCXXSUFFIX`, `$QT_MOCFROMCXXCOM`, `$QT_MOCFROMCXXFLAGS`, `$QT_MOCFROMHCOM`, `$QT_MOCFROMHFLAGS`, `$QT_MOCHPREFIX`, `$QT_MOCHSUFFIX`, `$QT_UIC`, `$QT_UICCOM`, `$QT_UICDECLFLAGS`, `$QT_UICDECLPREFIX`, `$QT_UICDECLSUFFIX`, `$QT_UICIMPLFLAGS`, `$QT_UICIMPLPREFIX`, `$QT_UICIMPLSUFFIX`, `$QT_UISUFFIX`.

Uses: `$QTDIR`.

rmic

Sets construction variables for the rmic utility.

Sets: `$JAVACLASSSUFFIX`, `$RMIC`, `$RMICCOM`, `$RMICFLAGS`.

Uses: `$RMICCOMSTR`.

rpcgen
Sets construction variables for building with RPCGEN.

Sets: `$RPCGEN`, `$RPCGENCLIENTFLAGS`, `$RPCGENFLAGS`, `$RPCGENHEADERFLAGS`, `$RPCGENSERVICEFLAGS`, `$RPCGENXDRFLAGS`.

sgiar
Sets construction variables for the SGI library archiver.

Sets: `$AR`, `$ARCOMSTR`, `$ARFLAGS`, `$LIBPREFIX`, `$LIBSUFFIX`, `$SHLINK`, `$SHLINKFLAGS`.

Uses: `$ARCOMSTR`, `$SHLINKCOMSTR`.

sgic++
Sets construction variables for the SGI C++ compiler.

Sets: `$CXX`, `$CXXFLAGS`, `$SHCXX`, `$SHOBJSUFFIX`.

sgicc
Sets construction variables for the SGI C compiler.

Sets: `$CXX`, `$SHOBJSUFFIX`.

sgilink
Sets construction variables for the SGI linker.

Sets: `$LINK`, `$RPATHPREFIX`, `$RPATHSUFFIX`, `$SHLINKFLAGS`.

sunar
Sets construction variables for the Sun library archiver.

Sets: `$AR`, `$ARCOM`, `$ARFLAGS`, `$LIBPREFIX`, `$LIBSUFFIX`.

Uses: `$ARCOMSTR`.

sunc++
Sets construction variables for the Sun C++ compiler.

Sets: `$CXX`, `$CXXVERSION`, `$SHCXX`, `$SHCXXFLAGS`, `$SHOBJPREFIX`, `$SHOBJSUFFIX`.

suncc
Sets construction variables for the Sun C compiler.

Sets: `$CXX`, `$SHCCFLAGS`, `$SHOBJPREFIX`, `$SHOBJSUFFIX`.

sunf77
Set construction variables for the Sun f77 Fortran compiler.

Sets: `$F77`, `$FORTRAN`, `$SHF77`, `$SHF77FLAGS`, `$SHFORTRAN`, `$SHFORTRANFLAGS`.

sunf90
Set construction variables for the Sun f90 Fortran compiler.

Sets: `$F90`, `$FORTRAN`, `$SHF90`, `$SHF90FLAGS`, `$SHFORTRAN`, `$SHFORTRANFLAGS`.

sunf95

Set construction variables for the Sun f95 Fortran compiler.

Sets: `$F95`, `$FORTRAN`, `$SHF95`, `$SHF95FLAGS`, `$SHFORTRAN`, `$SHFORTRANFLAGS`.

sunlink

Sets construction variables for the Sun linker.

Sets: `$RPATHPREFIX`, `$RPATHSUFFIX`, `$SHLINKFLAGS`.

swig

Sets construction variables for the SWIG interface generator.

Sets: `$SWIG`, `$SWIGCFILESUFFIX`, `$SWIGCOM`, `$SWIGCXXFILESUFFIX`, `$SWIGDIRECTORSUFFIX`, `$SWIGFLAGS`, `$SWIGINCPREFIX`, `$SWIGINCSUFFIX`, `$SWIGPATH`, `$SWIGVERSION`, `$_SWIGINCFLAGS`.

Uses: `$SWIGCOMSTR`.

tar

Sets construction variables for the tar archiver.

Sets: `$TAR`, `$TARCOM`, `$TARFLAGS`, `$TARSUFFIX`.

Uses: `$TARCOMSTR`.

tex

Sets construction variables for the TeX formatter and typesetter.

Sets: `$BIBTEX`, `$BIBTEXCOM`, `$BIBTEXFLAGS`, `$LATEX`, `$LATEXCOM`, `$LATEXFLAGS`, `$MAKEINDEX`, `$MAKEINDEXCOM`, `$MAKEINDEXFLAGS`, `$TEX`, `$TEXCOM`, `$TEXFLAGS`.

Uses: `$BIBTEXCOMSTR`, `$LATEXCOMSTR`, `$MAKEINDEXCOMSTR`, `$TEXCOMSTR`.

textfile

Set construction variables for the `Textfile` and `Substfile` builders.

Sets: `$LINESEPARATOR`, `$SUBSTFILEPREFIX`, `$SUBSTFILESUFFIX`, `$TEXTFILEPREFIX`, `$TEXTFILESUFFIX`.

Uses: `$SUBST_DICT`.

tlib

Sets construction variables for the Borlan tib library archiver.

Sets: `$AR`, `$ARCOM`, `$ARFLAGS`, `$LIBPREFIX`, `$LIBSUFFIX`.

Uses: `$ARCOMSTR`.

xgettext

This scons tool is a part of scons `gettext` toolset. It provides scons interface to **xgettext(1)** program, which extracts internationalized messages from source code. The tool provides `POTUpdate` builder to make `PO` *Template* files.

Sets: `$POTSUFFIX`, `$POTUPDATE_ALIAS`, `$XGETTEXTCOM`, `$XGETTEXTCOMSTR`, `$XGETTEXTFLAGS`, `$XGETTEXTFROM`, `$XGETTEXTFROMPREFIX`, `$XGETTEXTFROMSUFFIX`,

$XGETTEXTPATH, $XGETTEXTPATHPREFIX, $XGETTEXTPATHSUFFIX, $_XGETTEXTDOMAIN, $_XGETTEXTFROMFLAGS, $_XGETTEXTPATHFLAGS.

Uses: $POTDOMAIN.

yacc
Sets construction variables for the yacc parse generator.

Sets: $YACC, $YACCCOM, $YACCFLAGS, $YACCHFILESUFFIX, $YACCHXXFILESUFFIX, $YACCVCGFILESUFFIX.

Uses: $YACCCOMSTR.

zip
Sets construction variables for the zip archiver.

Sets: $ZIP, $ZIPCOM, $ZIPCOMPRESSION, $ZIPFLAGS, $ZIPSUFFIX.

Uses: $ZIPCOMSTR.

Builder Methods

You tell **scons** what to build by calling *Builders*, functions which take particular action(s) to produce a particular result type (conventionally described by the builder name such as `Program`) when given source files of a particular type. Calling a builder defines one or more targets to the build system; whether the targets are actually built on a given invocation is determined by command-line options, target selection rules, and whether SCons determines the target(s) are out of date.

SCons defines a number of builders, and you can also write your own. Builders are attached to a construction environment as methods, and the available builder methods are listed as key-value pairs in the `BUILDERS` attribute of the construction environment. The available builders can be displayed like this for debugging purposes:

```
env = Environment()
print("Builders:", list(env['BUILDERS']))
```

Builder methods take two required arguments: *`target`* and *`source`*. Either can be passed as a scalar or as a list. The *`target`* and *`source`* arguments can be specified either as positional arguments, in which case *`target`* comes first, or as keyword arguments, using *`target`*= and *`source`*=. Although both arguments are nominally required, if there is a single source and the target can be inferred the *`target`* argument can be omitted (see below). Builder methods also take a variety of keyword arguments, described below.

The builder *may* add other targets beyond those requested if indicated by an *Emitter* (see the section called “Builder Objects” and, for example, $PROGEMITTER for more information).

Because long lists of file names can lead to a lot of quoting, **scons** supplies a `Split` global function and a same-named environment method that splits a single string into a list, using strings of white-space characters as the delimiter. (similar to the Python string `split` method, but succeeds even if the input isn't a string.)

The following are equivalent examples of calling the `Program` builder method:

```
env.Program('bar', ['bar.c', 'foo.c'])
env.Program('bar', Split('bar.c foo.c'))
env.Program('bar', env.Split('bar.c foo.c'))
env.Program(source=['bar.c', 'foo.c'], target='bar')
env.Program(target='bar', source=Split('bar.c foo.c'))
env.Program(target='bar', source=env.Split('bar.c foo.c'))
env.Program('bar', source='bar.c foo.c'.split())
```

Python follows the POSIX pathname convention for path strings: if a string begins with the operating system pathname separator (on Windows both the slash and backslash separator work, and any leading drive specifier is ignored for the determination) it is considered an absolute path, otherwise it is a relative path. If the path string contains no separator characters, it is searched for as a file in the current directory. If it contains separator characters, the search follows down from the starting point, which is the top of the directory tree for an absolute path and the current directory for a relative path.

scons recognizes a third way to specify path strings: if the string begins with the **#** character it is *top-relative* - it works like a relative path but the search follows down from the directory containing the top-level `SConstruct` rather than from the current directory. The **#** is allowed to be followed by a pathname separator, which is ignored if found in that position. Top-relative paths only work in places where **scons** will interpret the path (see some examples below). To be used in other contexts the string will need to be converted to a relative or absolute path first.

`target` and *`source`* can be absolute, relative, or top-relative. Relative pathnames are searched considering the directory of the SConscript file currently being processed as the "current directory".

Examples:

```

# The comments describing the targets that will be built
# assume these calls are in a SConscript file in the
# a subdirectory named "subdir".

# Builds the program "subdir/foo" from "subdir/foo.c":
env.Program('foo', 'foo.c')

# Builds the program "/tmp/bar" from "subdir/bar.c":
env.Program('/tmp/bar', 'bar.c')

# An initial '#' or '#/' are equivalent; the following
# calls build the programs "foo" and "bar" (in the
# top-level SConstruct directory) from "subdir/foo.c" and
# "subdir/bar.c", respectively:
env.Program('#foo', 'foo.c')
env.Program('#/bar', 'bar.c')

# Builds the program "other/foo" (relative to the top-level
# SConstruct directory) from "subdir/foo.c":
env.Program('#other/foo', 'foo.c')

# This will not work, only SCons interfaces understand '#',
# os.path.exists is pure Python:
if os.path.exists('#inc/foo.h'):
    env.Append(CPPPATH='#inc')
```

When the target shares the same base name as the source and only the suffix varies, and if the builder method has a suffix defined for the target file type, then the target argument may be omitted completely, and **scons** will deduce the target file name from the source file name. The following examples all build the executable program **bar** (on POSIX systems) or **bar.exe** (on Windows systems) from the `bar.c` source file:

```
env.Program(target='bar', source='bar.c')
env.Program('bar', source='bar.c')
env.Program(source='bar.c')
env.Program('bar.c')
```

As a convenience, a *`srcdir`* keyword argument may be specified when calling a Builder. When specified, all source file strings that are not absolute paths or top-relative paths will be interpreted relative to the specified *`srcdir`*. The following example will build the `build/prog` (or `build/prog.exe` on Windows) program from the files `src/f1.c` and `src/f2.c`:

```
env.Program('build/prog', ['f1.c', 'f2.c'], srcdir='src')
```

Keyword arguments that are not specifically recognized are treated as construction variable *overrides*, which replace or add those variables on a limited basis. These overrides will only be in effect when building the target of the builder call, and will not affect other parts of the build. For example, if you want to specify some libraries needed by just one program:

```
env.Program('hello', 'hello.c', LIBS=['gl', 'glut'])
```

or generate a shared library with a non-standard suffix:

```
env.SharedLibrary(
    target='word',
    source='word.cpp',
    SHLIBSUFFIX='.ocx',
    LIBSUFFIXES=['.ocx'],
)
```

Note that both the `$SHLIBSUFFIX` and `$LIBSUFFIXES` variables must be set if you want **scons** to search automatically for dependencies on the non-standard library names; see the descriptions below of these variables for more information.

The optional *`parse_flags`* keyword argument is recognized by builders. This works similarly to the `env.MergeFlags` method, where the argument value is broken into individual settings and merged into the appropriate construction variables.

```
env.Program('hello', 'hello.c', parse_flags='-Iinclude -DEBUG -lm')
```

This example adds 'include' to `CPPPATH`, 'EBUG' to `CPPDEFINES`, and 'm' to `LIBS`.

Although the builder methods defined by **scons** are, in fact, methods of a construction environment object, many may also be called without an explicit environment:

```
Program('hello', 'hello.c')
SharedLibrary('word', 'word.cpp')
```

If called this way, methods will internally use the default environment that consists of the tools and values that **scons** has determined are appropriate for the local system.

Builder methods that can be called without an explicit environment (indicated in the listing of builders without a leading `env.`) may be called from custom Python modules that you import into an SConscript file by adding the following to the Python module:

```
from SCons.Script import *
```

Builder methods return a `NodeList`, a list-like object whose elements are Nodes, SCons' internal representation of build targets or sources. See the section called “File and Directory Nodes” for more information. The returned `NodeList` object can be passed to other builder methods as source(s) or passed to any SCons function or method where a filename would normally be accepted.

For example, to add a specific preprocessor define when compiling one specific object file but not the others:

```
bar_obj_list = env.StaticObject('bar.c', CPPDEFINES='-DBAR')
env.Program("prog", ['foo.c', bar_obj_list, 'main.c'])
```

Using a Node as in this example makes for a more portable build by avoiding having to specify a platform-specific object suffix when calling the `Program` builder method.

The `NodeList` object is also convenient to pass to the `Default` function, for the same reason of avoiding a platform-specific name:

```
tgt = env.Program("prog", ["foo.c", "bar.c", "main.c"])
Default(tgt)
```

Builder calls will automatically "flatten" lists passed as source and target, so they are free to contain elements which are themselves lists, such as `bar_obj_list` returned by the `StaticObject` call above. If you need to manipulate a list of lists returned by builders directly in Python code, you can either build a new list by hand:

```
foo = Object('foo.c')
bar = Object('bar.c')
objects = ['begin.o'] + foo + ['middle.o'] + bar + ['end.o']
for obj in objects:
    print(str(obj))
```

Or you can use the `Flatten` function supplied by **scons** to create a list containing just the Nodes, which may be more convenient:

```
foo = Object('foo.c')
bar = Object('bar.c')
objects = Flatten(['begin.o', foo, 'middle.o', bar, 'end.o'])
for obj in objects:
    print(str(obj))
```

SCons builder calls return a list-like object, not an actual Python list, so it is not appropriate to use the Python add operator (`+` or `+=`) to append builder results to a Python list. Because the list and the object are different types, Python will not update the original list in place, but will instead create a new `NodeList` object containing the concatenation of the list elements and the builder results. This will cause problems for any other Python variables in your SCons configuration that still hold on to a reference to the original list. Instead, use the Python list `extend` method to make sure the list is updated in-place. Example:

```
object_files = []

# Do NOT use += here:
```

```
#    object_files += Object('bar.c')
#
# It will not update the object_files list in place.
#
# Instead, use the list extend method:
object_files.extend(Object('bar.c'))
```

The path name for a Node's file may be used by passing the Node to Python's builtin `str` function:

```
bar_obj_list = env.StaticObject('bar.c', CPPDEFINES='-DBAR')
print("The path to bar_obj is:", str(bar_obj_list[0]))
```

Note that because the Builder call returns a `NodeList`, you have to access the first element in the list, (`bar_obj_list[0]` in the example) to get at the Node that actually represents the object file.

Builder calls support a *`chdir`* keyword argument that specifies that the Builder's action(s) should be executed after changing directory. If the *`chdir`* argument is a string or a directory Node, scons will change to the specified directory. If the *`chdir`* is not a string or Node and is non-zero, then scons will change to the target file's directory.

```
# scons will change to the "sub" subdirectory
# before executing the "cp" command.
env.Command('sub/dir/foo.out', 'sub/dir/foo.in',
            "cp dir/foo.in dir/foo.out",
            chdir='sub')

# Because chdir is not a string, scons will change to the
# target's directory ("sub/dir") before executing the
# "cp" command.
env.Command('sub/dir/foo.out', 'sub/dir/foo.in',
            "cp foo.in foo.out",
            chdir=1)
```

Note that SCons will *not* automatically modify its expansion of construction variables like `$TARGET` and `$SOURCE` when using the *`chdir`* keyword argument--that is, the expanded file names will still be relative to the top-level directory where `SConstruct` was found, and consequently incorrect relative to the chdir directory. If you use the *`chdir`* keyword argument, you will typically need to supply a different command line using expansions like `${TARGET.file}` and `${SOURCE.file}` to use just the filename portion of the targets and source.

When trying to handle errors that may occur in a builder method, consider that the corresponding Action is executed at a different time than the SConscript file statement calling the builder. It is not useful to wrap a builder call in a `try` block, since success in the builder call is not the same as the builder itself succeeding. If necessary, a Builder's Action should be coded to exit with a useful exception message indicating the problem in the SConscript files - programmatically recovering from build errors is rarely useful.

scons predefines the following builder methods. Depending on the setup of a particular construction environment and on the type and software installation status of the underlying system, not all builders may be available to that construction environment.

`CFile()`
***`env`*.`CFile()`**

Builds a C source file given a lex (`.l`) or yacc (`.y`) input file. The suffix specified by the `$CFILESUFFIX` construction variable (`.c` by default) is automatically added to the target if it is not already present. Example:

```
# builds foo.c
env.CFile(target = 'foo.c', source = 'foo.l')
# builds bar.c
env.CFile(target = 'bar', source = 'bar.y')
```

Command()
***env*.Command()**

The `Command` "Builder" is actually a function that looks like a Builder, but takes a required third argument, which is the action to take to construct the target from the source, used for "one-off" builds where a full builder is not needed. Thus it does not follow the builder calling rules described at the start of this section. See instead the `Command` function description for the calling syntax and details.

CompilationDatabase()
***env*.CompilationDatabase()**

`CompilationDatabase` is a special builder which adds a target to create a JSON formatted compilation database compatible with `clang` tooling (see the LLVM specification [https://clang.llvm.org/docs/JSONCompilationDatabase.html]). This database is suitable for consumption by various tools and editors who can use it to obtain build and dependency information which otherwise would be internal to SCons. The builder does not require any source files to be specified, rather it arranges to emit information about all of the C, C++ and assembler source/output pairs identified in the build that are not excluded by the optional filter `$COMPILATIONDB_PATH_FILTER`. The target is subject to the usual SCons target selection rules.

If called with no arguments, the builder will default to a target name of `compile_commands.json`.

If called with a single positional argument, **scons** will "deduce" the target name from that source argument, giving it the same name, and then ignore the source. This is the usual way to call the builder if a non-default target name is wanted.

If called with either the *`target=`* or *`source=`* keyword arguments, the value of the argument is taken as the target name. If called with both, the *`target=`* value is used and *`source=`* is ignored. If called with multiple sources, the source list will be ignored, since there is no way to deduce what the intent was; in this case the default target name will be used.

Note

You must load the `compilation_db` tool prior to specifying any part of your build or some source/output files will not show up in the compilation database.

*Available since **scons** 4.0.*

CXXFile()
***env*.CXXFile()**

Builds a C++ source file given a lex (`.ll`) or yacc (`.yy`) input file. The suffix specified by the `$CXXFILESUFFIX` construction variable (`.cc` by default) is automatically added to the target if it is not already present. Example:

```

# builds foo.cc
env.CXXFile(target = 'foo.cc', source = 'foo.ll')
# builds bar.cc
env.CXXFile(target = 'bar', source = 'bar.yy')
```

DocbookEpub()
***env*.DocbookEpub()**

A pseudo-Builder, providing a Docbook toolchain for EPUB output.

```
env = Environment(tools=['docbook'])
env.DocbookEpub('manual.epub', 'manual.xml')
```

or simply

```
env = Environment(tools=['docbook'])
env.DocbookEpub('manual')
```

`DocbookHtml()`
***`env`*.`DocbookHtml()`**

A pseudo-Builder, providing a Docbook toolchain for HTML output.

```
env = Environment(tools=['docbook'])
env.DocbookHtml('manual.html', 'manual.xml')
```

or simply

```
env = Environment(tools=['docbook'])
env.DocbookHtml('manual')
```

`DocbookHtmlChunked()`
***`env`*.`DocbookHtmlChunked()`**

A pseudo-Builder providing a Docbook toolchain for chunked HTML output. It supports the *`base.dir`* parameter. The `chunkfast.xsl` file (requires "EXSLT") is used as the default stylesheet. Basic syntax:

```
env = Environment(tools=['docbook'])
env.DocbookHtmlChunked('manual')
```

where `manual.xml` is the input file.

If you use the *`root.filename`* parameter in your own stylesheets you have to specify the new target name. This ensures that the dependencies get correct, especially for the cleanup via “`scons -c`”:

```
env = Environment(tools=['docbook'])
env.DocbookHtmlChunked('mymanual.html', 'manual', xsl='htmlchunk.xsl')
```

Some basic support for the *`base.dir`* parameter is provided. You can add the *`base_dir`* keyword to your Builder call, and the given prefix gets prepended to all the created filenames:

```
env = Environment(tools=['docbook'])
env.DocbookHtmlChunked('manual', xsl='htmlchunk.xsl', base_dir='output/')
```

Make sure that you don't forget the trailing slash for the base folder, else your files get renamed only!

`DocbookHtmlhelp()`
***`env`*.`DocbookHtmlhelp()`**

A pseudo-Builder, providing a Docbook toolchain for HTMLHELP output. Its basic syntax is:

```
env = Environment(tools=['docbook'])
env.DocbookHtmlhelp('manual')
```

where `manual.xml` is the input file.

If you use the *`root.filename`* parameter in your own stylesheets you have to specify the new target name. This ensures that the dependencies get correct, especially for the cleanup via “**`scons -c`**”:

```
env = Environment(tools=['docbook'])
```

```
env.DocbookHtmlhelp('mymanual.html', 'manual', xsl='htmlhelp.xsl')
```

Some basic support for the *`base.dir`* parameter is provided. You can add the *`base_dir`* keyword to your Builder call, and the given prefix gets prepended to all the created filenames:

```
env = Environment(tools=['docbook'])
env.DocbookHtmlhelp('manual', xsl='htmlhelp.xsl', base_dir='output/')
```

Make sure that you don't forget the trailing slash for the base folder, else your files get renamed only!

DocbookMan()
***env*.DocbookMan()**

A pseudo-Builder, providing a Docbook toolchain for Man page output. Its basic syntax is:

```
env = Environment(tools=['docbook'])
env.DocbookMan('manual')
```

where `manual.xml` is the input file. Note, that you can specify a target name, but the actual output names are automatically set from the `refname` entries in your XML source.

DocbookPdf()
***env*.DocbookPdf()**

A pseudo-Builder, providing a Docbook toolchain for PDF output.

```
env = Environment(tools=['docbook'])
env.DocbookPdf('manual.pdf', 'manual.xml')
```

or simply

```
env = Environment(tools=['docbook'])
env.DocbookPdf('manual')
```

DocbookSlidesHtml()
***env*.DocbookSlidesHtml()**

A pseudo-Builder, providing a Docbook toolchain for HTML slides output.

```
env = Environment(tools=['docbook'])
env.DocbookSlidesHtml('manual')
```

If you use the *`titlefoil.html`* parameter in your own stylesheets you have to give the new target name. This ensures that the dependencies get correct, especially for the cleanup via “**`scons -c`**”:

```
env = Environment(tools=['docbook'])
env.DocbookSlidesHtml('mymanual.html','manual', xsl='slideshtml.xsl')
```

Some basic support for the *`base.dir`* parameter is provided. You can add the *`base_dir`* keyword to your Builder call, and the given prefix gets prepended to all the created filenames:

```
env = Environment(tools=['docbook'])
env.DocbookSlidesHtml('manual', xsl='slideshtml.xsl', base_dir='output/')
```

Make sure that you don't forget the trailing slash for the base folder, else your files get renamed only!

DocbookSlidesPdf()
***env*.DocbookSlidesPdf()**

A pseudo-Builder, providing a Docbook toolchain for PDF slides output.

```
env = Environment(tools=['docbook'])
```

```
env.DocbookSlidesPdf('manual.pdf', 'manual.xml')
```

or simply

```
env = Environment(tools=['docbook'])
env.DocbookSlidesPdf('manual')
```

`DocbookXInclude`()
***env*.`DocbookXInclude`()**

A pseudo-Builder, for resolving XIncludes in a separate processing step.

```
env = Environment(tools=['docbook'])
env.DocbookXInclude('manual_xincluded.xml', 'manual.xml')
```

`DocbookXslt`()
***env*.`DocbookXslt`()**

A pseudo-Builder, applying a given XSL transformation to the input file.

```
env = Environment(tools=['docbook'])
env.DocbookXslt('manual_transformed.xml', 'manual.xml', xsl='transform.xslt')
```

Note, that this builder requires the *`xsl`* parameter to be set.

`DVI`()
***env*.`DVI`()**

Builds a `.dvi` file from a `.tex`, `.ltx` or `.latex` input file. If the source file suffix is `.tex`, **scons** will examine the contents of the file; if the string `\documentclass` or `\documentstyle` is found, the file is assumed to be a LaTeX file and the target is built by invoking the `$LATEXCOM` command line; otherwise, the `$TEXCOM` command line is used. If the file is a LaTeX file, the `DVI` builder method will also examine the contents of the `.aux` file and invoke the `$BIBTEX` command line if the string `bibdata` is found, start `$MAKEINDEX` to generate an index if a `.ind` file is found and will examine the contents `.log` file and re-run the `$LATEXCOM` command if the log file says it is necessary.

The suffix `.dvi` (hard-coded within TeX itself) is automatically added to the target if it is not already present. Examples:

```

# builds from aaa.tex
env.DVI(target = 'aaa.dvi', source = 'aaa.tex')
# builds bbb.dvi
env.DVI(target = 'bbb', source = 'bbb.ltx')
# builds from ccc.latex
env.DVI(target = 'ccc.dvi', source = 'ccc.latex')
```

`Gs`()
***env*.`Gs`()**

A Builder for explicitly calling the gs executable. Depending on the underlying OS, the different names gs, gsos2 and gswin32c are tried.

```

env = Environment(tools=['gs'])
env.Gs(
    'cover.jpg',
    'scons-scons.pdf',
    GSFLAGS='-dNOPAUSE -dBATCH -sDEVICE=jpeg -dFirstPage=1 -dLastPage=1 -q',
)
```

`Install()`
***`env`*.`Install()`**

Installs one or more source files or directories in the specified target, which must be a directory. The names of the specified source files or directories remain the same within the destination directory. The sources may be given as a string or as a node returned by a builder.

```
env.Install(target='/usr/local/bin', source=['foo', 'bar'])
```

Note that if target paths chosen for the `Install` builder (and the related `InstallAs` and `InstallVersionedLib` builders) are outside the project tree, such as in the example above, they may not be selected for "building" by default, since in the absence of other instructions **scons** builds targets that are underneath the top directory (the directory that contains the `SConstruct` file, usually the current directory). Use command line targets or the `Default` function in this case.

If the `--install-sandbox` command line option is given, the target directory will be prefixed by the directory path specified. This is useful to test installs without installing to a "live" location in the system.

See also `FindInstalledFiles`. For more thoughts on installation, see the User Guide (particularly the section on Command-Line Targets and the chapters on Installing Files and on Alias Targets).

`InstallAs()`
***`env`*.`InstallAs()`**

Installs one or more source files or directories to specific names, allowing changing a file or directory name as part of the installation. It is an error if the target and source arguments list different numbers of files or directories.

```
env.InstallAs(target='/usr/local/bin/foo',
              source='foo_debug')
env.InstallAs(target=['../lib/libfoo.a', '../lib/libbar.a'],
              source=['libFOO.a', 'libBAR.a'])
```

See the note under `Install`.

`InstallVersionedLib()`
***`env`*.`InstallVersionedLib()`**

Installs a versioned shared library. The symlinks appropriate to the architecture will be generated based on symlinks of the source library.

```
env.InstallVersionedLib(target='/usr/local/bin/foo',
                        source='libxyz.1.5.2.so')
```

See the note under `Install`.

`Jar()`
***`env`*.`Jar()`**

Builds a Java archive (`.jar`) file from the specified list of sources. Any directories in the source list will be searched for `.class` files). Any `.java` files in the source list will be compiled to `.class` files by calling the `Java` Builder.

If the `$JARCHDIR` value is set, the jar command will change to the specified directory using the `-C` option. If `$JARCHDIR` is not set explicitly, SCons will use the top of any subdirectory tree in which Java `.class` were built by the `Java` Builder.

If the contents any of the source files begin with the string `Manifest-Version`, the file is assumed to be a manifest and is passed to the jar command with the `m` option set.

```
env.Jar(target = 'foo.jar', source = 'classes')

env.Jar(target = 'bar.jar',
        source = ['bar1.java', 'bar2.java'])
```

Java()
***env*.Java()**

Builds one or more Java class files. The sources may be any combination of explicit `.java` files, or directory trees which will be scanned for `.java` files.

SCons will parse each source `.java` file to find the classes (including inner classes) defined within that file, and from that figure out the target `.class` files that will be created. The class files will be placed underneath the specified target directory.

SCons will also search each Java file for the Java package name, which it assumes can be found on a line beginning with the string `package` in the first column; the resulting `.class` files will be placed in a directory reflecting the specified package name. For example, the file `Foo.java` defining a single public `Foo` class and containing a package name of `sub.dir` will generate a corresponding `sub/dir/Foo.class` class file.

Examples:

```
env.Java(target = 'classes', source = 'src')
env.Java(target = 'classes', source = ['src1', 'src2'])
env.Java(target = 'classes', source = ['File1.java', 'File2.java'])
```

Java source files can use the native encoding for the underlying OS. Since SCons compiles in simple ASCII mode by default, the compiler will generate warnings about unmappable characters, which may lead to errors as the file is processed further. In this case, the user must specify the `LANG` environment variable to tell the compiler what encoding is used. For portibility, it's best if the encoding is hard-coded so that the compile will work if it is done on a system with a different encoding.

```
env = Environment()
env['ENV']['LANG'] = 'en_GB.UTF-8'
```

JavaH()
***env*.JavaH()**

Builds C header and source files for implementing Java native methods. The target can be either a directory in which the header files will be written, or a header file name which will contain all of the definitions. The source can be the names of `.class` files, the names of `.java` files to be compiled into `.class` files by calling the `Java` builder method, or the objects returned from the `Java` builder method.

If the construction variable `$JAVACLASSDIR` is set, either in the environment or in the call to the `JavaH` builder method itself, then the value of the variable will be stripped from the beginning of any `.class` file names.

Examples:

```
# builds java_native.h
classes = env.Java(target="classdir", source="src")
env.JavaH(target="java_native.h", source=classes)
```

```
# builds include/package_foo.h and include/package_bar.h
env.JavaH(target="include", source=["package/foo.class", "package/bar.class"])

# builds export/foo.h and export/bar.h
env.JavaH(
    target="export",
    source=["classes/foo.class", "classes/bar.class"],
    JAVACLASSDIR="classes",
)
```

Note

Java versions starting with 10.0 no longer use the **javah** command for generating JNI headers/ sources, and indeed have removed the command entirely (see Java Enhancement Proposal JEP 313 [https:openjdk.java.net/jeps/313]), making this tool harder to use for that purpose. SCons may autodiscover a **javah** belonging to an older release if there are multiple Java versions on the system, which will lead to incorrect results. To use with a newer Java, override the default values of `$JAVAH` (to contain the path to the **javac**) and `$JAVAHFLAGS` (to contain at least a `-h` flag) and note that generating headers with **javac** requires supplying source `.java` files only, not `.class` files.

`Library()`
***`env`*.`Library()`**

A synonym for the `StaticLibrary` builder method.

`LoadableModule()`
***`env`*.`LoadableModule()`**

On most systems, this is the same as `SharedLibrary`. On Mac OS X (Darwin) platforms, this creates a loadable module bundle.

`M4()`
***`env`*.`M4()`**

Builds an output file from an M4 input file. This uses a default `$M4FLAGS` value of `-E`, which considers all warnings to be fatal and stops on the first warning when using the GNU version of m4. Example:

```
env.M4(target = 'foo.c', source = 'foo.c.m4')
```

`Moc()`
***`env`*.`Moc()`**

Builds an output file from a **moc** input file. **moc** input files are either header files or C++ files. This builder is only available after using the tool `qt`. See the `$QTDIR` variable for more information. Example:

```
env.Moc('foo.h')  # generates moc_foo.cc
env.Moc('foo.cpp')  # generates foo.moc
```

`MOFiles()`
***`env`*.`MOFiles()`**

This builder belongs to `msgfmt` tool. The builder compiles `PO` files to `MO` files.

Example 1. Create `pl.mo` and `en.mo` by compiling `pl.po` and `en.po`:

```
  # ...
```

```
env.MOFiles(['pl', 'en'])
```

Example 2. Compile files for languages defined in `LINGUAS` file:

```
# ...
env.MOFiles(LINGUAS_FILE = 1)
```

Example 3. Create `pl.mo` and `en.mo` by compiling `pl.po` and `en.po` plus files for languages defined in `LINGUAS` file:

```
# ...
env.MOFiles(['pl', 'en'], LINGUAS_FILE = 1)
```

Example 4. Compile files for languages defined in `LINGUAS` file (another version):

```
# ...
env['LINGUAS_FILE'] = 1
env.MOFiles()
```

`MSVSProject()`
***`env`*`.MSVSProject()`**

Builds a Microsoft Visual Studio project file, and by default builds a solution file as well.

This builds a Visual Studio project file, based on the version of Visual Studio that is configured (either the latest installed version, or the version specified by `$MSVS_VERSION` in the Environment constructor). For Visual Studio 6, it will generate a `.dsp` file. For Visual Studio 7, 8, and 9, it will generate a `.vcproj` file. For Visual Studio 10 and later, it will generate a `.vcxproj` file.

By default, this also generates a solution file for the specified project, a `.dsw` file for Visual Studio 6 or a `.sln` file for Visual Studio 7 and later. This behavior may be disabled by specifying `auto_build_solution=0` when you call `MSVSProject`, in which case you presumably want to build the solution file(s) by calling the `MSVSSolution` Builder (see below).

The `MSVSProject` builder takes several lists of filenames to be placed into the project file. These are currently limited to `srcs`, `incs`, `localincs`, `resources`, and `misc`. These are pretty self-explanatory, but it should be noted that these lists are added to the `$SOURCES` construction variable as strings, NOT as SCons File Nodes. This is because they represent file names to be added to the project file, not the source files used to build the project file.

The above filename lists are all optional, although at least one must be specified for the resulting project file to be non-empty.

In addition to the above lists of values, the following values may be specified:

target
The name of the target `.dsp` or `.vcproj` file. The correct suffix for the version of Visual Studio must be used, but the `$MSVSPROJECTSUFFIX` construction variable will be defined to the correct value (see example below).

variant
The name of this particular variant. For Visual Studio 7 projects, this can also be a list of variant names. These are typically things like "Debug" or "Release", but really can be anything you want. For Visual Studio 7 projects, they may also specify a target platform separated from the variant name by a `|` (vertical pipe) character: `Debug|Xbox`. The default target platform is Win32. Multiple calls to `MSVSProject` with

different variants are allowed; all variants will be added to the project file with their appropriate build targets and sources.

cmdargs
: Additional command line arguments for the different variants. The number of `cmdargs` entries must match the number of `variant` entries, or be empty (not specified). If you give only one, it will automatically be propagated to all variants.

cppdefines
: Preprocessor definitions for the different variants. The number of `cppdefines` entries must match the number of `variant` entries, or be empty (not specified). If you give only one, it will automatically be propagated to all variants. If you don't give this parameter, SCons will use the invoking environment's `CPPDEFINES` entry for all variants.

cppflags
: Compiler flags for the different variants. If a /std:c++ flag is found then /Zc:__cplusplus is appended to the flags if not already found, this ensures that intellisense uses the /std:c++ switch. The number of `cppflags` entries must match the number of `variant` entries, or be empty (not specified). If you give only one, it will automatically be propagated to all variants. If you don't give this parameter, SCons will combine the invoking environment's `CCFLAGS`, `CXXFLAGS`, `CPPFLAGS` entries for all variants.

cpppaths
: Compiler include paths for the different variants. The number of `cpppaths` entries must match the number of `variant` entries, or be empty (not specified). If you give only one, it will automatically be propagated to all variants. If you don't give this parameter, SCons will use the invoking environment's `CPPPATH` entry for all variants.

buildtarget
: An optional string, node, or list of strings or nodes (one per build variant), to tell the Visual Studio debugger what output target to use in what build variant. The number of `buildtarget` entries must match the number of `variant` entries.

runfile
: The name of the file that Visual Studio 7 and later will run and debug. This appears as the value of the `Output` field in the resulting Visual Studio project file. If this is not specified, the default is the same as the specified `buildtarget` value.

Note that because SCons always executes its build commands from the directory in which the `SConstruct` file is located, if you generate a project file in a different directory than the `SConstruct` directory, users will not be able to double-click on the file name in compilation error messages displayed in the Visual Studio console output window. This can be remedied by adding the Visual C/C++ `/FC` compiler option to the `$CCFLAGS` variable so that the compiler will print the full path name of any files that cause compilation errors.

Example usage:

```
barsrcs = ['bar.cpp']
barincs = ['bar.h']
barlocalincs = ['StdAfx.h']
barresources = ['bar.rc','resource.h']
barmisc = ['bar_readme.txt']

dll = env.SharedLibrary(target='bar.dll',
                        source=barsrcs)
buildtarget = [s for s in dll if str(s).endswith('dll')]
env.MSVSProject(target='Bar' + env['MSVSPROJECTSUFFIX'],
                srcs=barsrcs,
                incs=barincs,
                localincs=barlocalincs,
                resources=barresources,
                misc=barmisc,
                buildtarget=buildtarget,
                variant='Release')
```

Starting with version 2.4 of SCons it is also possible to specify the optional argument *`DebugSettings`*, which creates files for debugging under Visual Studio:

DebugSettings
: A dictionary of debug settings that get written to the `.vcproj.user` or the `.vcxproj.user` file, depending on the version installed. As it is done for cmdargs (see above), you can specify a *`DebugSettings`* dictionary per variant. If you give only one, it will be propagated to all variants.

Currently, only Visual Studio v9.0 and Visual Studio version v11 are implemented, for other versions no file is generated. To generate the user file, you just need to add a *`DebugSettings`* dictionary to the environment with the right parameters for your MSVS version. If the dictionary is empty, or does not contain any good value, no file will be generated.

Following is a more contrived example, involving the setup of a project for variants and DebugSettings:

```
# Assuming you store your defaults in a file
vars = Variables('variables.py')
msvcver = vars.args.get('vc', '9')

# Check command args to force one Microsoft Visual Studio version
if msvcver == '9' or msvcver == '11':
  env = Environment(MSVC_VERSION=msvcver+'.0', MSVC_BATCH=False)
else:
  env = Environment()

AddOption('--userfile', action='store_true', dest='userfile', default=False,
          help="Create Visual Studio Project user file")

#
# 1. Configure your Debug Setting dictionary with options you want in the list
# of allowed options, for instance if you want to create a user file to launch
# a specific application for testing your dll with Microsoft Visual Studio 2008 (v9):
#
V9DebugSettings = {
    'Command':'c:\\myapp\\using\\thisdll.exe',
    'WorkingDirectory': 'c:\\myapp\\using\\',
    'CommandArguments': '-p password',
#     'Attach':'false',
#     'DebuggerType':'3',
#     'Remote':'1',
#     'RemoteMachine': None,
#     'RemoteCommand': None,
#     'HttpUrl': None,
#     'PDBPath': None,
#     'SQLDebugging': None,
#     'Environment': '',
#     'EnvironmentMerge':'true',
#     'DebuggerFlavor': None,
#     'MPIRunCommand': None,
#     'MPIRunArguments': None,
#     'MPIRunWorkingDirectory': None,
#     'ApplicationCommand': None,
#     'ApplicationArguments': None,
#     'ShimCommand': None,
#     'MPIAcceptMode': None,
#     'MPIAcceptFilter': None,
}

#
# 2. Because there are a lot of different options depending on the Microsoft
# Visual Studio version, if you use more than one version you have to
# define a dictionary per version, for instance if you want to create a user
# file to launch a specific application for testing your dll with Microsoft
# Visual Studio 2012 (v11):
#
V10DebugSettings = {
    'LocalDebuggerCommand': 'c:\\myapp\\using\\thisdll.exe',
    'LocalDebuggerWorkingDirectory': 'c:\\myapp\\using\\',
    'LocalDebuggerCommandArguments': '-p password',
#     'LocalDebuggerEnvironment': None,
#     'DebuggerFlavor': 'WindowsLocalDebugger',
#     'LocalDebuggerAttach': None,
#     'LocalDebuggerDebuggerType': None,
#     'LocalDebuggerMergeEnvironment': None,
#     'LocalDebuggerSQLDebugging': None,
#     'RemoteDebuggerCommand': None,
#     'RemoteDebuggerCommandArguments': None,
#     'RemoteDebuggerWorkingDirectory': None,
#     'RemoteDebuggerServerName': None,
#     'RemoteDebuggerConnection': None,
#     'RemoteDebuggerDebuggerType': None,
#     'RemoteDebuggerAttach': None,
#     'RemoteDebuggerSQLDebugging': None,
#     'DeploymentDirectory': None,
#     'AdditionalFiles': None,
#     'RemoteDebuggerDeployDebugCppRuntime': None,
#     'WebBrowserDebuggerHttpUrl': None,
#     'WebBrowserDebuggerDebuggerType': None,
#     'WebServiceDebuggerHttpUrl': None,
#     'WebServiceDebuggerDebuggerType': None,
#     'WebServiceDebuggerSQLDebugging': None,
}

#
# 3. Select the dictionary you want depending on the version of visual Studio
# Files you want to generate.
#
if not env.GetOption('userfile'):
    dbgSettings = None
```

```
elif env.get('MSVC_VERSION', None) == '9.0':
    dbgSettings = V9DebugSettings
elif env.get('MSVC_VERSION', None) == '11.0':
    dbgSettings = V10DebugSettings
else:
    dbgSettings = None

#
# 4. Add the dictionary to the DebugSettings keyword.
#
barsrcs = ['bar.cpp', 'dllmain.cpp', 'stdafx.cpp']
barincs = ['targetver.h']
barlocalincs = ['StdAfx.h']
barresources = ['bar.rc','resource.h']
barmisc = ['ReadMe.txt']

dll = env.SharedLibrary(target='bar.dll',
                        source=barsrcs)

env.MSVSProject(target='Bar' + env['MSVSPROJECTSUFFIX'],
                srcs=barsrcs,
                incs=barincs,
                localincs=barlocalincs,
                resources=barresources,
                misc=barmisc,
                buildtarget=[dll[0]] * 2,
                variant=('Debug|Win32', 'Release|Win32'),
                cmdargs='vc=%s' %  msvcver,
                DebugSettings=(dbgSettings, {}))
```

`MSVSSolution`()
***`env`*.`MSVSSolution`()**

Builds a Microsoft Visual Studio solution file.

This builds a Visual Studio solution file, based on the version of Visual Studio that is configured (either the latest installed version, or the version specified by `$MSVS_VERSION` in the construction environment). For Visual Studio 6, it will generate a `.dsw` file. For Visual Studio 7 (.NET), it will generate a `.sln` file.

The following values must be specified:

target
The name of the target .dsw or .sln file. The correct suffix for the version of Visual Studio must be used, but the value `$MSVSSOLUTIONSUFFIX` will be defined to the correct value (see example below).

variant
The name of this particular variant, or a list of variant names (the latter is only supported for MSVS 7 solutions). These are typically things like "Debug" or "Release", but really can be anything you want. For MSVS 7 they may also specify target platform, like this "Debug|Xbox". Default platform is Win32.

projects
A list of project file names, or Project nodes returned by calls to the `MSVSProject` Builder, to be placed into the solution file. It should be noted that these file names are NOT added to the $SOURCES environment variable in form of files, but rather as strings. This is because they represent file names to be added to the solution file, not the source files used to build the solution file.

Example Usage:

```
env.MSVSSolution(
    target="Bar" + env["MSVSSOLUTIONSUFFIX"],
    projects=["bar" + env["MSVSPROJECTSUFFIX"]],
    variant="Release",
)
```

`Ninja()`
***`env`*`.Ninja()`**

`Ninja` is a special builder which adds a target to create a ninja build file. The builder does not require any source files to be specified.

Note

This is an experimental feature. To enable it you must use one of the following methods

```
# On the command line
--experimental=ninja

# Or in your SConstruct
SetOption('experimental', 'ninja')
```

This functionality is subject to change and/or removal without deprecation cycle.

To use this tool you must install pypi's ninja package [https://pypi.org/project/ninja/]. This can be done via **pip install ninja**

If called with no arguments, the builder will default to a target name of `ninja.build`.

If called with a single positional argument, **scons** will "deduce" the target name from that source argument, giving it the same name, and then ignore the source. This is the usual way to call the builder if a non-default target name is wanted.

If called with either the *`target=`* or *`source=`* keyword arguments, the value of the argument is taken as the target name. If called with both, the *`target=`* value is used and *`source=`* is ignored. If called with multiple sources, the source list will be ignored, since there is no way to deduce what the intent was; in this case the default target name will be used.

*Available since **scons** 4.2.*

`Object()`
***`env`*`.Object()`**

A synonym for the `StaticObject` builder method.

`Package()`
***`env`*`.Package()`**

Builds software distribution packages. A *package* is a container format which includes files to install along with metadata. Packaging is optional, and must be enabled by specifying the `packaging` tool. For example:

```
env = Environment(tools=['default', 'packaging'])
```

SCons can build packages in a number of well known packaging formats. The target package type may be selected with the the `$PACKAGETYPE` construction variable or the `--package-type` command line option. The package type may be a list, in which case SCons will attempt to build packages for each type in the list. Example:

```
env.Package(PACKAGETYPE=['src_zip', 'src_targz'], ...other args...)
```

The currently supported packagers are:

`msi`	Microsoft Installer package
`rpm`	RPM Package Manger package
`ipkg`	Itsy Package Management package
`tarbz2`	bzip2-compressed tar file
`targz`	gzip-compressed tar file
`tarxz`	xz-compressed tar file
`zip`	zip file
`src_tarbz2`	bzip2-compressed tar file suitable as source to another packager
`src_targz`	gzip-compressed tar file suitable as source to another packager
`src_tarxz`	xz-compressed tar file suitable as source to another packager
`src_zip`	zip file suitable as source to another packager

The file list to include in the package may be specified with the `source` keyword argument. If omitted, the `FindInstalledFiles` function is called behind the scenes to select all files that have an `Install`, `InstallAs` or `InstallVersionedLib` Builder attached. If the `target` keyword argument is omitted, the target name(s) will be deduced from the package type(s).

The metadata comes partly from attributes of the files to be packaged, and partly from packaging *tags*. Tags can be passed as keyword arguments to the `Package` builder call, and may also be attached to files (or more accurately, Nodes representing files) with the `Tag` function. Some package-level tags are mandatory, and will lead to errors if omitted. The mandatory tags vary depending on the package type.

While packaging, the builder uses a temporary location named by the value of the `$PACKAGEROOT` variable - the package sources are copied there before packaging.

Packaging example:

```
env = Environment(tools=["default", "packaging"])
env.Install("/bin/", "my_program")
env.Package(
    NAME="foo",
    VERSION="1.2.3",
    PACKAGEVERSION=0,
    PACKAGETYPE="rpm",
    LICENSE="gpl",
    SUMMARY="balalalalal",
    DESCRIPTION="this should be really really long",
    X_RPM_GROUP="Application/fu",
    SOURCE_URL="https://foo.org/foo-1.2.3.tar.gz",
)
```

In this example, the target `/bin/my_program` created by the `Install` call would not be built by default since it is not under the project top directory. However, since no *`source`* is specified to the `Package` builder, it is selected for packaging by the default sources rule. Since packaging is done using `$PACKAGEROOT`, no write is actually done to the system's `/bin` directory, and the target *will* be selected since after rebasing to underneath `$PACKAGEROOT` it is now under the top directory of the project.

PCH()
***env*.PCH()**

Builds a Microsoft Visual C++ precompiled header. Calling this builder returns a list of two targets: the PCH as the first element, and the object file as the second element. Normally the object file is ignored. This builder is only provided when Microsoft Visual C++ is being used as the compiler. The `PCH` builder is generally used in conjunction with the `$PCH` construction variable to force object files to use the precompiled header:

```
env['PCH'] = env.PCH('StdAfx.cpp')[0]
```

PDF()
***env*.PDF()**

Builds a `.pdf` file from a `.dvi` input file (or, by extension, a `.tex`, `.ltx`, or `.latex` input file). The suffix specified by the `$PDFSUFFIX` construction variable (`.pdf` by default) is added automatically to the target if it is not already present. Example:

```
# builds from aaa.tex
env.PDF(target = 'aaa.pdf', source = 'aaa.tex')
# builds bbb.pdf from bbb.dvi
env.PDF(target = 'bbb', source = 'bbb.dvi')
```

POInit()
***env*.POInit()**

This builder belongs to `msginit` tool. The builder initializes missing `PO` file(s) if `$POAUTOINIT` is set. If `$POAUTOINIT` is not set (default), `POInit` prints instruction for user (that is supposed to be a translator), telling how the `PO` file should be initialized. In normal projects *you should not use `POInit` and use `POUpdate` instead*. `POUpdate` chooses intelligently between **msgmerge(1)** and **msginit(1)**. `POInit` always uses **msginit(1)** and should be regarded as builder for special purposes or for temporary use (e.g. for quick, one time initialization of a bunch of `PO` files) or for tests.

Target nodes defined through `POInit` are not built by default (they're `Ignore`d from `'.'` node) but are added to special `Alias` (`'po-create'` by default). The alias name may be changed through the `$POCREATE_ALIAS` construction variable. All `PO` files defined through `POInit` may be easily initialized by **scons po-create**.

Example 1. Initialize `en.po` and `pl.po` from `messages.pot`:

```
  # ...
  env.POInit(['en', 'pl']) # messages.pot --> [en.po, pl.po]
```

Example 2. Initialize `en.po` and `pl.po` from `foo.pot`:

```
  # ...
  env.POInit(['en', 'pl'], ['foo']) # foo.pot --> [en.po, pl.po]
```

Example 3. Initialize `en.po` and `pl.po` from `foo.pot` but using `$POTDOMAIN` construction variable:

```
  # ...
  env.POInit(['en', 'pl'], POTDOMAIN='foo') # foo.pot --> [en.po, pl.po]
```

Example 4. Initialize `PO` files for languages defined in `LINGUAS` file. The files will be initialized from template `messages.pot`:

```
  # ...
  env.POInit(LINGUAS_FILE = 1) # needs 'LINGUAS' file
```

Example 5. Initialize `en.po` and `pl.pl` `PO` files plus files for languages defined in `LINGUAS` file. The files will be initialized from template `messages.pot`:

```
  # ...
  env.POInit(['en', 'pl'], LINGUAS_FILE = 1)
```

Example 6. You may preconfigure your environment first, and then initialize `PO` files:

```
  # ...
  env['POAUTOINIT'] = 1
  env['LINGUAS_FILE'] = 1
  env['POTDOMAIN'] = 'foo'
  env.POInit()
```

which has same efect as:

```
  # ...
  env.POInit(POAUTOINIT = 1, LINGUAS_FILE = 1, POTDOMAIN = 'foo')
```

`PostScript()`
***env*.`PostScript()`**

Builds a `.ps` file from a `.dvi` input file (or, by extension, a `.tex`, `.ltx`, or `.latex` input file). The suffix specified by the `$PSSUFFIX` construction variable (`.ps` by default) is added automatically to the target if it is not already present. Example:

```
# builds from aaa.tex
env.PostScript(target = 'aaa.ps', source = 'aaa.tex')
# builds bbb.ps from bbb.dvi
env.PostScript(target = 'bbb', source = 'bbb.dvi')
```

`POTUpdate()`
***env*.`POTUpdate()`**

The builder belongs to `xgettext` tool. The builder updates target `POT` file if exists or creates one if it doesn't. The node is not built by default (i.e. it is `Ignore`d from `'.'`), but only on demand (i.e. when given `POT` file is required or when special alias is invoked). This builder adds its targe node (`messages.pot`, say) to a special alias (`pot-update` by default, see `$POTUPDATE_ALIAS`) so you can update/create them easily with **scons pot-update**. The file is not written until there is no real change in internationalized messages (or in comments that enter `POT` file).

Note

You may see **xgettext(1)** being invoked by the `xgettext` tool even if there is no real change in internationalized messages (so the `POT` file is not being updated). This happens every time a source file has changed. In such case we invoke **xgettext(1)** and compare its output with the content of `POT` file to decide whether the file should be updated or not.

Example 1. Let's create `po/` directory and place following `SConstruct` script there:

```
# SConstruct in 'po/' subdir
env = Environment( tools = ['default', 'xgettext'] )
env.POTUpdate(['foo'], ['../a.cpp', '../b.cpp'])
env.POTUpdate(['bar'], ['../c.cpp', '../d.cpp'])
```

Then invoke scons few times:

```
user@host:$ scons             # Does not create foo.pot nor bar.pot
user@host:$ scons foo.pot     # Updates or creates foo.pot
user@host:$ scons pot-update  # Updates or creates foo.pot and bar.pot
user@host:$ scons -c          # Does not clean foo.pot nor bar.pot.
```

the results shall be as the comments above say.

Example 2. The `POTUpdate` builder may be used with no target specified, in which case default target `messages.pot` will be used. The default target may also be overridden by setting `$POTDOMAIN` construction variable or providing it as an override to `POTUpdate` builder:

```
# SConstruct script
env = Environment( tools = ['default', 'xgettext'] )
env['POTDOMAIN'] = "foo"
env.POTUpdate(source = ["a.cpp", "b.cpp"]) # Creates foo.pot ...
env.POTUpdate(POTDOMAIN = "bar", source = ["c.cpp", "d.cpp"]) # and bar.pot
```

Example 3. The sources may be specified within separate file, for example `POTFILES.in`:

```
# POTFILES.in in 'po/' subdirectory
../a.cpp
../b.cpp
# end of file
```

The name of the file (`POTFILES.in`) containing the list of sources is provided via `$XGETTEXTFROM`:

```
# SConstruct file in 'po/' subdirectory
env = Environment( tools = ['default', 'xgettext'] )
env.POTUpdate(XGETTEXTFROM = 'POTFILES.in')
```

Example 4. You may use `$XGETTEXTPATH` to define source search path. Assume, for example, that you have files `a.cpp`, `b.cpp`, `po/SConstruct`, `po/POTFILES.in`. Then your `POT`-related files could look as below:

```
# POTFILES.in in 'po/' subdirectory
a.cpp
```

```
b.cpp
# end of file
```

```
# SConstruct file in 'po/' subdirectory
env = Environment( tools = ['default', 'xgettext'] )
env.POTUpdate(XGETTEXTFROM = 'POTFILES.in', XGETTEXTPATH='../')
```

Example 5. Multiple search directories may be defined within a list, i.e. `XGETTEXTPATH = ['dir1', 'dir2', ...]`. The order in the list determines the search order of source files. The path to the first file found is used.

Let's create `0/1/po/SConstruct` script:

```
# SConstruct file in '0/1/po/' subdirectory
env = Environment( tools = ['default', 'xgettext'] )
env.POTUpdate(XGETTEXTFROM = 'POTFILES.in', XGETTEXTPATH=['../', '../../'])
```

and `0/1/po/POTFILES.in`:

```
# POTFILES.in in '0/1/po/' subdirectory
a.cpp
# end of file
```

Write two `*.cpp` files, the first one is `0/a.cpp`:

```
/* 0/a.cpp */
gettext("Hello from ../../a.cpp")
```

and the second is `0/1/a.cpp`:

```
/* 0/1/a.cpp */
gettext("Hello from ../a.cpp")
```

then run scons. You'll obtain `0/1/po/messages.pot` with the message `"Hello from ../a.cpp"`. When you reverse order in `$XGETTEXTFOM`, i.e. when you write SConscript as

```
# SConstruct file in '0/1/po/' subdirectory
env = Environment( tools = ['default', 'xgettext'] )
env.POTUpdate(XGETTEXTFROM = 'POTFILES.in', XGETTEXTPATH=['../../', '../'])
```

then the `messages.pot` will contain `msgid "Hello from ../../a.cpp"` line and not `msgid "Hello from ../a.cpp"`.

`POUpdate`()
***`env`*.`POUpdate`()**

The builder belongs to `msgmerge` tool. The builder updates `PO` files with **msgmerge(1)**, or initializes missing `PO` files as described in documentation of `msginit` tool and `POInit` builder (see also `$POAUTOINIT`). Note, that `POUpdate` *does not add its targets to* `po-create` *alias* as `POInit` does.

Target nodes defined through `POUpdate` are not built by default (they're `Ignore`d from `'.'` node). Instead, they are added automatically to special `Alias` (`'po-update'` by default). The alias name may be changed through the `$POUPDATE_ALIAS` construction variable. You can easily update `PO` files in your project by **scons po-update**.

Example 1. Update `en.po` and `pl.po` from `messages.pot` template (see also `$POTDOMAIN`), assuming that the later one exists or there is rule to build it (see `POTUpdate`):

```
# ...
env.POUpdate(['en','pl']) # messages.pot --> [en.po, pl.po]
```

Example 2. Update `en.po` and `pl.po` from `foo.pot` template:

```
# ...
env.POUpdate(['en', 'pl'], ['foo']) # foo.pot -->  [en.po, pl.pl]
```

Example 3. Update `en.po` and `pl.po` from `foo.pot` (another version):

```
# ...
env.POUpdate(['en', 'pl'], POTDOMAIN='foo') # foo.pot -- > [en.po, pl.pl]
```

Example 4. Update files for languages defined in `LINGUAS` file. The files are updated from `messages.pot` template:

```
# ...
env.POUpdate(LINGUAS_FILE = 1) # needs 'LINGUAS' file
```

Example 5. Same as above, but update from `foo.pot` template:

```
# ...
env.POUpdate(LINGUAS_FILE = 1, source = ['foo'])
```

Example 6. Update `en.po` and `pl.po` plus files for languages defined in `LINGUAS` file. The files are updated from `messages.pot` template:

```
# produce 'en.po', 'pl.po' + files defined in 'LINGUAS':
env.POUpdate(['en', 'pl' ], LINGUAS_FILE = 1)
```

Example 7. Use `$POAUTOINIT` to automatically initialize `PO` file if it doesn't exist:

```
# ...
env.POUpdate(LINGUAS_FILE = 1, POAUTOINIT = 1)
```

Example 8. Update `PO` files for languages defined in `LINGUAS` file. The files are updated from `foo.pot` template. All necessary settings are pre-configured via environment.

```
# ...
env['POAUTOINIT'] = 1
env['LINGUAS_FILE'] = 1
env['POTDOMAIN'] = 'foo'
env.POUpdate()
```

`Program()`
***env*.`Program()`**

Builds an executable given one or more object files or C, C++, D, or Fortran source files. If any C, C++, D or Fortran source files are specified, then they will be automatically compiled to object files using the `Object` builder method; see that builder method's description for a list of legal source file suffixes and how they are interpreted. The target executable file prefix, specified by the `$PROGPREFIX` construction variable (nothing by default), and suffix, specified by the `$PROGSUFFIX` construction variable (by default, `.exe` on Windows systems, nothing on POSIX systems), are automatically added to the target if not already present. Example:

```
env.Program(target='foo', source=['foo.o', 'bar.c', 'baz.f'])
```

`ProgramAllAtOnce()`
***env*.`ProgramAllAtOnce()`**

Builds an executable from D sources without first creating individual objects for each file.

D sources can be compiled file-by-file as C and C++ source are, and D is integrated into the **scons** Object and Program builders for this model of build. D codes can though do whole source meta-programming (some of the testing frameworks do this). For this it is imperative that all sources are compiled and linked in a single call to the D compiler. This builder serves that purpose.

```
    env.ProgramAllAtOnce('executable', ['mod_a.d, mod_b.d', 'mod_c.d'])
```

This command will compile the modules mod_a, mod_b, and mod_c in a single compilation process without first creating object files for the modules. Some of the D compilers will create executable.o others will not.

`RES()`
***env*.`RES()`**

Builds a Microsoft Visual C++ resource file. This builder method is only provided when Microsoft Visual C++ or MinGW is being used as the compiler. The `.res` (or `.o` for MinGW) suffix is added to the target name if no other suffix is given. The source file is scanned for implicit dependencies as though it were a C file. Example:

```
env.RES('resource.rc')
```

`RMIC()`
***env*.`RMIC()`**

Builds stub and skeleton class files for remote objects from Java `.class` files. The target is a directory relative to which the stub and skeleton class files will be written. The source can be the names of `.class` files, or the objects return from the `Java` builder method.

If the construction variable `$JAVACLASSDIR` is set, either in the environment or in the call to the `RMIC` builder method itself, then the value of the variable will be stripped from the beginning of any `.class` file names.

```
classes = env.Java(target = 'classdir', source = 'src')
env.RMIC(target = 'outdir1', source = classes)

env.RMIC(target = 'outdir2',
         source = ['package/foo.class', 'package/bar.class'])

env.RMIC(target = 'outdir3',
         source = ['classes/foo.class', 'classes/bar.class'],
         JAVACLASSDIR = 'classes')
```

`RPCGenClient()`
***`env`*`.RPCGenClient()`**

Generates an RPC client stub (`_clnt.c`) file from a specified RPC (`.x`) source file. Because rpcgen only builds output files in the local directory, the command will be executed in the source file's directory by default.

```
# Builds src/rpcif_clnt.c
env.RPCGenClient('src/rpcif.x')
```

`RPCGenHeader()`
***`env`*`.RPCGenHeader()`**

Generates an RPC header (`.h`) file from a specified RPC (`.x`) source file. Because rpcgen only builds output files in the local directory, the command will be executed in the source file's directory by default.

```
# Builds src/rpcif.h
env.RPCGenHeader('src/rpcif.x')
```

`RPCGenService()`
***`env`*`.RPCGenService()`**

Generates an RPC server-skeleton (`_svc.c`) file from a specified RPC (`.x`) source file. Because rpcgen only builds output files in the local directory, the command will be executed in the source file's directory by default.

```
# Builds src/rpcif_svc.c
env.RPCGenClient('src/rpcif.x')
```

`RPCGenXDR()`
***`env`*`.RPCGenXDR()`**

Generates an RPC XDR routine (`_xdr.c`) file from a specified RPC (`.x`) source file. Because rpcgen only builds output files in the local directory, the command will be executed in the source file's directory by default.

```
# Builds src/rpcif_xdr.c
env.RPCGenClient('src/rpcif.x')
```

`SharedLibrary()`
***`env`*`.SharedLibrary()`**

Builds a shared library (`.so` on a POSIX system, `.dll` on Windows) given one or more object files or C, C++, D or Fortran source files. If any source files are given, then they will be automatically compiled to object files. The target library file prefix, specified by the `$SHLIBPREFIX` construction variable (by default, `lib` on POSIX systems, nothing on Windows systems), and suffix, specified by the `$SHLIBSUFFIX` construction variable (by default, `.dll` on Windows systems, `.so` on POSIX systems), are automatically added to the target if not already present. Example:

```
env.SharedLibrary(target='bar', source=['bar.c', 'foo.o'])
```

On Windows systems, the `SharedLibrary` builder method will always build an import library (`.lib`) in addition to the shared library (`.dll`), adding a `.lib` library with the same basename if there is not already a `.lib` file explicitly listed in the targets.

On Cygwin systems, the `SharedLibrary` builder method will always build an import library (`.dll.a`) in addition to the shared library (`.dll`), adding a `.dll.a` library with the same basename if there is not already a `.dll.a` file explicitly listed in the targets.

Any object files listed in the *source* must have been built for a shared library (that is, using the `SharedObject` builder method). **scons** will raise an error if there is any mismatch.

On some platforms, there is a distinction between a shared library (loaded automatically by the system to resolve external references) and a loadable module (explicitly loaded by user action). For maximum portability, use the `LoadableModule` builder for the latter.

When the `$SHLIBVERSION` construction variable is defined, a versioned shared library is created. This modifies `$SHLINKFLAGS` as required, adds the version number to the library name, and creates any symbolic links that are needed.

```
env.SharedLibrary(target='bar', source=['bar.c', 'foo.o'], SHLIBVERSION='1.5.2')
```

On a POSIX system, versions with a single token create exactly one symlink: `libbar.so.6` would have symlink `libbar.so` only. On a POSIX system, versions with two or more tokens create exactly two symlinks: `libbar.so.2.3.1` would have symlinks `libbar.so` and `libbar.so.2`; on a Darwin (OSX) system the library would be `libbar.2.3.1.dylib` and the link would be `libbar.dylib`.

On Windows systems, specifying *register=1* will cause the `.dll` to be registered after it is built. The command that is run is determined by the `$REGSVR` construction variable (**regsvr32** by default), and the flags passed are determined by `$REGSVRFLAGS`. By default, `$REGSVRFLAGS` includes the `/s` option, to prevent dialogs from popping up and requiring user attention when it is run. If you change `$REGSVRFLAGS`, be sure to include the `/s` option. For example,

```
env.SharedLibrary(target='bar', source=['bar.cxx', 'foo.obj'], register=1)
```

will register `bar.dll` as a COM object when it is done linking it.

SharedObject()
***env*.SharedObject()**

Builds an object file intended for inclusion in a shared library. Source files must have one of the same set of extensions specified above for the `StaticObject` builder method. On some platforms building a shared object requires additional compiler option (e.g. `-fPIC` for **gcc**) in addition to those needed to build a normal (static) object, but on some platforms there is no difference between a shared object and a normal (static) one. When there is a difference, SCons will only allow shared objects to be linked into a shared library, and will use a different suffix for shared objects. On platforms where there is no difference, SCons will allow both normal (static) and shared objects to be linked into a shared library, and will use the same suffix for shared and normal (static) objects. The target object file prefix, specified by the `$SHOBJPREFIX` construction variable (by default, the same as `$OBJPREFIX`), and suffix, specified by the `$SHOBJSUFFIX` construction variable, are automatically added to the target if not already present. Examples:

```
env.SharedObject(target='ddd', source='ddd.c')
env.SharedObject(target='eee.o', source='eee.cpp')
env.SharedObject(target='fff.obj', source='fff.for')
```

Note that the source files will be scanned according to the suffix mappings in the `SourceFileScanner` object. See the manpage section "Scanner Objects" for more information.

StaticLibrary()
***env*.StaticLibrary()**

Builds a static library given one or more object files or C, C++, D or Fortran source files. If any source files are given, then they will be automatically compiled to object files. The static library file prefix, specified by the `$LIBPREFIX` construction variable (by default, `lib` on POSIX systems, nothing on Windows systems), and suffix, specified by the `$LIBSUFFIX` construction variable (by default, `.lib` on Windows systems, `.a` on POSIX systems), are automatically added to the target if not already present. Example:

```
env.StaticLibrary(target='bar', source=['bar.c', 'foo.o'])
```

Any object files listed in the *`source`* must have been built for a static library (that is, using the `StaticObject` builder method). **scons** will raise an error if there is any mismatch.

`StaticObject()`
***`env`*.`StaticObject()`**

Builds a static object file from one or more C, C++, D, or Fortran source files. Source files must have one of the following extensions:

```
  .asm    assembly language file
  .ASM    assembly language file
  .c      C file
  .C      Windows:  C file
          POSIX:  C++ file
  .cc     C++ file
  .cpp    C++ file
  .cxx    C++ file
  .cxx    C++ file
  .c++    C++ file
  .C++    C++ file
  .d      D file
  .f      Fortran file
  .F      Windows:  Fortran file
          POSIX:  Fortran file + C pre-processor
  .for    Fortran file
  .FOR    Fortran file
  .fpp    Fortran file + C pre-processor
  .FPP    Fortran file + C pre-processor
  .m      Object C file
  .mm     Object C++ file
  .s      assembly language file
  .S      Windows:  assembly language file
          ARM: CodeSourcery Sourcery Lite
  .sx     assembly language file + C pre-processor
          POSIX:  assembly language file + C pre-processor
  .spp    assembly language file + C pre-processor
  .SPP    assembly language file + C pre-processor
```

The target object file prefix, specified by the `$OBJPREFIX` construction variable (nothing by default), and suffix, specified by the `$OBJSUFFIX` construction variable (`.obj` on Windows systems, `.o` on POSIX systems), are automatically added to the target if not already present. Examples:

```
env.StaticObject(target='aaa', source='aaa.c')
env.StaticObject(target='bbb.o', source='bbb.c++')
env.StaticObject(target='ccc.obj', source='ccc.f')
```

Note that the source files will be scanned according to the suffix mappings in the `SourceFileScanner` object. See the manpage section "Scanner Objects" for more information.

`Substfile()`
***`env`*.`Substfile()`**

The `Substfile` builder creates a single text file from a template consisting of a file or set of files (or nodes), replacing text using the `$SUBST_DICT` construction variable (if set). If a set, they are concatenated into the target file using the value of the `$LINESEPARATOR` construction variable as a separator between contents; the separator is not emitted after the contents of the last file. Nested lists of source files are flattened. See also `Textfile`.

If a single source file name is specified and has a `.in` suffix, the suffix is stripped and the remainder of the name is used as the default target name.

The prefix and suffix specified by the `$SUBSTFILEPREFIX` and `$SUBSTFILESUFFIX` construction variables (an empty string by default in both cases) are automatically added to the target if they are not already present.

If a construction variable named `$SUBST_DICT` is present, it may be either a Python dictionary or a sequence of (*`key`*, *`value`*) tuples. If it is a dictionary it is converted into a list of tuples with unspecified order, so if one key is a prefix of another key or if one substitution could be further expanded by another subsitition, it is unpredictable whether the expansion will occur.

Any occurrences of a key in the source are replaced by the corresponding value, which may be a Python callable function or a string. If the value is a callable, it is called with no arguments to get a string. Strings are *subst*-expanded and the result replaces the key.

```
env = Environment(tools=['default'])

env['prefix'] = '/usr/bin'
script_dict = {'@prefix@': '/bin', '@exec_prefix@': '$prefix'}
env.Substfile('script.in', SUBST_DICT=script_dict)

conf_dict = {'%VERSION%': '1.2.3', '%BASE%': 'MyProg'}
env.Substfile('config.h.in', conf_dict, SUBST_DICT=conf_dict)

# UNPREDICTABLE - one key is a prefix of another
bad_foo = {'$foo': '$foo', '$foobar': '$foobar'}
env.Substfile('foo.in', SUBST_DICT=bad_foo)

# PREDICTABLE - keys are applied longest first
good_foo = [('$foobar', '$foobar'), ('$foo', '$foo')]
env.Substfile('foo.in', SUBST_DICT=good_foo)

# UNPREDICTABLE - one substitution could be futher expanded
bad_bar = {'@bar@': '@soap@', '@soap@': 'lye'}
env.Substfile('bar.in', SUBST_DICT=bad_bar)

# PREDICTABLE - substitutions are expanded in order
good_bar = (('@bar@', '@soap@'), ('@soap@', 'lye'))
env.Substfile('bar.in', SUBST_DICT=good_bar)

# the SUBST_DICT may be in common (and not an override)
substutions = {}
subst = Environment(tools=['textfile'], SUBST_DICT=substitutions)
substitutions['@foo@'] = 'foo'
subst['SUBST_DICT']['@bar@'] = 'bar'
subst.Substfile(
    'pgm1.c',
    [Value('#include "@foo@.h"'), Value('#include "@bar@.h"'), "common.in", "pgm1.in"],
)
subst.Substfile(
    'pgm2.c',
    [Value('#include "@foo@.h"'), Value('#include "@bar@.h"'), "common.in", "pgm2.in"],
)
```

Tar()
***env*.Tar()**

Builds a tar archive of the specified files and/or directories. Unlike most builder methods, the `Tar` builder method may be called multiple times for a given target; each additional call adds to the list of entries that will be built into the archive. Any source directories will be scanned for changes to any on-disk files, regardless of whether or not **scons** knows about them from other Builder or function calls.

```
env.Tar('src.tar', 'src')

# Create the stuff.tar file.
env.Tar('stuff', ['subdir1', 'subdir2'])
# Also add "another" to the stuff.tar file.
env.Tar('stuff', 'another')

# Set TARFLAGS to create a gzip-filtered archive.
env = Environment(TARFLAGS = '-c -z')
env.Tar('foo.tar.gz', 'foo')

# Also set the suffix to .tgz.
env = Environment(TARFLAGS = '-c -z',
                  TARSUFFIX = '.tgz')
env.Tar('foo')
```

Textfile()
***env*.Textfile()**

The `Textfile` builder generates a single text file from a template consisting of a list of strings, replacing text using the `$SUBST_DICT` construction variable (if set) - see `Substfile` for a description of replacement. The strings will be separated in the target file using the value of the `$LINESEPARATOR` construction variable; the line separator is not emitted after the last string. Nested lists of source strings are flattened. Source strings need not literally be Python strings: they can be Nodes or Python objects that convert cleanly to `Value` nodes

The prefix and suffix specified by the `$TEXTFILEPREFIX` and `$TEXTFILESUFFIX` construction variables (by default an empty string and `.txt`, respectively) are automatically added to the target if they are not already present. Examples:

```
# builds/writes foo.txt
env.Textfile(target='foo.txt', source=['Goethe', 42, 'Schiller'])

# builds/writes bar.txt
env.Textfile(target='bar', source=['lalala', 'tanteratei'], LINESEPARATOR='|*')

# nested lists are flattened automatically
env.Textfile(target='blob', source=['lalala', ['Goethe', 42, 'Schiller'], 'tanteratei']

# files may be used as input by wraping them in File()
```

```
env.Textfile(
    target='concat',  # concatenate files with a marker between
    source=[File('concat1'), File('concat2')],
    LINESEPARATOR='====================\n',
)
```

Results:

`foo.txt`

```

  Goethe
  42
  Schiller
```

`bar.txt`

```

  lalala|*tanteratei
```

`blob.txt`

```

  lalala
  Goethe
  42
  Schiller
  tanteratei
```

Translate()
***env*.Translate()**

This pseudo-builder belongs to `gettext` toolset. The builder extracts internationalized messages from source files, updates `POT` template (if necessary) and then updates `PO` translations (if necessary). If `$POAUTOINIT` is set, missing `PO` files will be automatically created (i.e. without translator person intervention). The variables `$LINGUAS_FILE` and `$POTDOMAIN` are taken into acount too. All other construction variables used by `POTUpdate`, and `POUpdate` work here too.

Example 1. The simplest way is to specify input files and output languages inline in a SCons script when invoking `Translate`

```

# SConscript in 'po/' directory
env = Environment( tools = ["default", "gettext"] )
env['POAUTOINIT'] = 1
env.Translate(['en','pl'], ['../a.cpp','../b.cpp'])
```

Example 2. If you wish, you may also stick to conventional style known from autotools, i.e. using `POTFILES.in` and `LINGUAS` files

```

# LINGUAS
en pl
#end
```

```

# POTFILES.in
```

```
a.cpp
b.cpp
# end
```

```
# SConscript
env = Environment( tools = ["default", "gettext"] )
env['POAUTOINIT'] = 1
env['XGETTEXTPATH'] = ['../']
env.Translate(LINGUAS_FILE = 1, XGETTEXTFROM = 'POTFILES.in')
```

The last approach is perhaps the recommended one. It allows easily split internationalization/localization onto separate SCons scripts, where a script in source tree is responsible for translations (from sources to `PO` files) and script(s) under variant directories are responsible for compilation of `PO` to `MO` files to and for installation of `MO` files. The "gluing factor" synchronizing these two scripts is then the content of `LINGUAS` file. Note, that the updated `POT` and `PO` files are usually going to be committed back to the repository, so they must be updated within the source directory (and not in variant directories). Additionaly, the file listing of `po/` directory contains `LINGUAS` file, so the source tree looks familiar to translators, and they may work with the project in their usual way.

Example 3. Let's prepare a development tree as below

```
project/
 + SConstruct
 + build/
 + src/
     + po/
         + SConscript
         + SConscript.i18n
         + POTFILES.in
         + LINGUAS
```

with `build` being variant directory. Write the top-level `SConstruct` script as follows

```
# SConstruct
env = Environment( tools = ["default", "gettext"] )
VariantDir('build', 'src', duplicate = 0)
env['POAUTOINIT'] = 1
SConscript('src/po/SConscript.i18n', exports = 'env')
SConscript('build/po/SConscript', exports = 'env')
```

the `src/po/SConscript.i18n` as

```
# src/po/SConscript.i18n
Import('env')
env.Translate(LINGUAS_FILE=1, XGETTEXTFROM='POTFILES.in', XGETTEXTPATH=['../'])
```

and the `src/po/SConscript`

```
# src/po/SConscript
Import('env')
env.MOFiles(LINGUAS_FILE = 1)
```

Such setup produces `POT` and `PO` files under source tree in `src/po/` and binary `MO` files under variant tree in `build/po/`. This way the `POT` and `PO` files are separated from other output files, which must not be committed back to source repositories (e.g. `MO` files).

Note

In above example, the `PO` files are not updated, nor created automatically when you issue **scons** **'.'** command. The files must be updated (created) by hand via **scons po-update** and then `MO` files can be compiled by running **scons** **'.'**.

`TypeLibrary()`
***`env`*.`TypeLibrary()`**

Builds a Windows type library (`.tlb`) file from an input IDL file (`.idl`). In addition, it will build the associated interface stub and proxy source files, naming them according to the base name of the `.idl` file. For example,

```
env.TypeLibrary(source="foo.idl")
```

Will create `foo.tlb`, `foo.h`, `foo_i.c`, `foo_p.c` and `foo_data.c` files.

`Uic()`
***`env`*.`Uic()`**

Builds a header file, an implementation file and a moc file from an ui file. and returns the corresponding nodes in the that order. This builder is only available after using the tool `qt`. Note: you can specify `.ui` files directly as source files to the `Program`, `Library` and `SharedLibrary` builders without using this builder. Using this builder lets you override the standard naming conventions (be careful: prefixes are always prepended to names of built files; if you don't want prefixes, you may set them to ``). See the `$QTDIR` variable for more information. Example:

```
env.Uic('foo.ui')  # -> ['foo.h', 'uic_foo.cc', 'moc_foo.cc']
env.Uic(
    target=Split('include/foo.h gen/uicfoo.cc gen/mocfoo.cc'),
    source='foo.ui'
)  # -> ['include/foo.h', 'gen/uicfoo.cc', 'gen/mocfoo.cc']
```

`Zip()`
***`env`*.`Zip()`**

Builds a zip archive of the specified files and/or directories. Unlike most builder methods, the `Zip` builder method may be called multiple times for a given target; each additional call adds to the list of entries that will be built into the archive. Any source directories will be scanned for changes to any on-disk files, regardless of whether or not **scons** knows about them from other Builder or function calls.

```
env.Zip('src.zip', 'src')

# Create the stuff.zip file.
env.Zip('stuff', ['subdir1', 'subdir2'])
# Also add "another" to the stuff.tar file.
env.Zip('stuff', 'another')
```

All targets of builder methods automatically depend on their sources. An explicit dependency can be specified using the `env.Depends` method of a construction environment (see below).

In addition, **scons** automatically scans source files for various programming languages, so the dependencies do not need to be specified explicitly. By default, SCons can C source files, C++ source files, Fortran source files with `.F`

(POSIX systems only), `.fpp`, or `.FPP` file extensions, and assembly language files with `.S` (POSIX systems only), `.spp`, or `.SPP` files extensions for C preprocessor dependencies. SCons also has default support for scanning D source files, You can also write your own Scanners to add support for additional source file types. These can be added to the default Scanner object used by the `Object`, `StaticObject` and `SharedObject` Builders by adding them to the `SourceFileScanner` object. See the section called “Scanner Objects” for more information about defining your own Scanner objects and using the `SourceFileScanner` object.

Methods and Functions To Do Things

In addition to Builder methods, **scons** provides a number of other construction environment methods and global functions to manipulate the build configuration.

Usually, a construction environment method and global function with the same name both exist for convenience. In the following list, the global function is documented in this style:

```
Function(arguments, [optional arguments])
```

and the construction environment method looks like:

```
env.Function(arguments, [optional arguments])
```

If the function can be called both ways, then both forms are listed.

The global function and same-named construction environment method provide almost identical functionality, with a couple of exceptions. First, many of the construction environment methods affect only that construction environment, while the global function has a global effect. Second, where appropriate, calling the functionality through a construction environment will substitute construction variables into any supplied string arguments, while the global function doesn't have the context of a construction environment to pick variables from, so it cannot perform the substitution. For example:

```
Default('$FOO')

env = Environment(FOO='foo')
env.Default('$FOO')
```

In the above example, the call to the global `Default` function will add a target named **$FOO** to the list of default targets, while the call to the `env.Default` construction environment method will expand the value and add a target named **foo** to the list of default targets. For more on construction variable expansion, see the next section on construction variables.

Global functions may be called from custom Python modules that you import into an SConscript file by adding the following import to the Python module:

```
from SCons.Script import *
```

Construction environment methods and global functions provided by **scons** include:

Action(*action, [output, [var, ...]] [key=value, ...]*)
***env*.Action(*action, [output, [var, ...]] [key=value, ...]*)**

A factory function to create an Action object for the specified *`action`*. See the manpage section "Action Objects" for a complete explanation of the arguments and behavior.

Note that the `env.Action` form of the invocation will expand construction variables in any argument strings, including the *`action`* argument, at the time it is called using the construction variables in the *`env`* construction environment through which `env.Action` was called. The `Action` global function form delays all variable expansion until the Action object is actually used.

`AddMethod(`*`object, function, [name]`*`)`
***`env`*`.AddMethod(`*`function, [name]`*`)`**

Adds *`function`* to an object as a method. *`function`* will be called with an instance object as the first argument as for other methods. If *`name`* is given, it is used as the name of the new method, else the name of *`function`* is used.

When the global function `AddMethod` is called, the object to add the method to must be passed as the first argument; typically this will be `Environment`, in order to create a method which applies to all construction environments subsequently constructed. When called using the `env.AddMethod` form, the method is added to the specified construction environment only. Added methods propagate through `env.Clone` calls.

Examples:

```
# Function to add must accept an instance argument.
# The Python convention is to call this 'self'.
def my_method(self, arg):
    print("my_method() got", arg)

# Use the global function to add a method to the Environment class:
AddMethod(Environment, my_method)
env = Environment()
env.my_method('arg')

# Use the optional name argument to set the name of the method:
env.AddMethod(my_method, 'other_method_name')
env.other_method_name('another arg')
```

`AddOption(`*`arguments`*`)`

Adds a local (project-specific) command-line option. *`arguments`* are the same as those supported by the `add_option` method in the standard Python library module `optparse`, with a few additional capabilities noted below. See the documentation for `optparse` for a thorough discussion of its option-processing capabities.

In addition to the arguments and values supported by the `optparse` `add_option` method, `AddOption` allows setting the *`nargs`* keyword value to a string consisting of a question mark (`'?'`) to indicate that the option argument for that option string is optional. If the option string is present on the command line but has no matching option argument, the value of the *`const`* keyword argument is produced as the value of the option. If the option string is omitted from the command line, the value of the *`default`* keyword argument is produced, as usual; if there is no *`default`* keyword argument in the `AddOption` call, `None` is produced.

`optparse` recognizes abbreviations of long option names, as long as they can be unambiguously resolved. For example, if `add_option` is called to define a `--devicename` option, it will recognize `--device`, `--dev` and so forth as long as there is no other option which could also match to the same abbreviation. Options added via `AddOption` do not support the automatic recognition of abbreviations. Instead, to allow specific abbreviations, include them as synonyms in the `AddOption` call itself.

Once a new command-line option has been added with `AddOption`, the option value may be accessed using `GetOption` or `env.GetOption`. `SetOption` is not currently supported for options added with `AddOption`.

Help text for an option is a combination of the string supplied in the *`help`* keyword argument to `AddOption` and information collected from the other keyword arguments. Such help is displayed if the `-h` command line option is used (but not with `-H`). Help for all local options is displayed under the separate heading **Local Options**. The options are unsorted - they will appear in the help text in the order in which the `AddOption` calls occur.

Example:

```
AddOption(
    '--prefix',
    dest='prefix',
    nargs=1,
    type='string',
    action='store',
    metavar='DIR',
    help='installation prefix',
)
env = Environment(PREFIX=GetOption('prefix'))
```

For that example, the following help text would be produced:

```
Local Options:
  --prefix=DIR                installation prefix
```

Help text for local options may be unavailable if the `Help` function has been called, see the `Help` documentation for details.

Note

As an artifact of the internal implementation, the behavior of options added by `AddOption` which take option arguments is undefined *if* whitespace (rather than an `=` sign) is used as the separator on the command line. Users should avoid such usage; it is recommended to add a note to this effect to project documentation if the situation is likely to arise. In addition, if the *`nargs`* keyword is used to specify more than one following option argument (that is, with a value of `2` or greater), such arguments would necessarily be whitespace separated, triggering the issue. Developers should not use `AddOption` this way. Future versions of SCons will likely forbid such usage.

`AddPostAction(`*`target, action`*`)`
***`env`*`.AddPostAction(`*`target, action`*`)`**

Arranges for the specified *`action`* to be performed after the specified *`target`* has been built. The specified action(s) may be an Action object, or anything that can be converted into an Action object See the manpage section "Action Objects" for a complete explanation.

When multiple targets are supplied, the action may be called multiple times, once after each action that generates one or more targets in the list.

`AddPreAction(`*`target, action`*`)`
***`env`*`.AddPreAction(`*`target, action`*`)`**

Arranges for the specified *`action`* to be performed before the specified *`target`* is built. The specified action(s) may be an Action object, or anything that can be converted into an Action object See the manpage section "Action Objects" for a complete explanation.

When multiple targets are specified, the action(s) may be called multiple times, once before each action that generates one or more targets in the list.

Note that if any of the targets are built in multiple steps, the action will be invoked just before the "final" action that specifically generates the specified target(s). For example, when building an executable program from a specified source `.c` file via an intermediate object file:

```
foo = Program('foo.c')
AddPreAction(foo, 'pre_action')
```

The specified `pre_action` would be executed before **scons** calls the link command that actually generates the executable program binary `foo`, not before compiling the `foo.c` file into an object file.

`Alias(`*`alias, [targets, [action]]`*`)`
***`env`*`.Alias(`*`alias, [targets, [action]]`*`)`**

Creates one or more phony targets that expand to one or more other targets. An optional *`action`* (command) or list of actions can be specified that will be executed whenever the any of the alias targets are out-of-date. Returns the Node object representing the alias, which exists outside of any file system. This Node object, or the alias name, may be used as a dependency of any other target, including another alias. `Alias` can be called multiple times for the same alias to add additional targets to the alias, or additional actions to the list for this alias. Aliases are global even if set through the construction environment method.

Examples:

```
Alias('install')
Alias('install', '/usr/bin')
Alias(['install', 'install-lib'], '/usr/local/lib')

env.Alias('install', ['/usr/local/bin', '/usr/local/lib'])
env.Alias('install', ['/usr/local/man'])

env.Alias('update', ['file1', 'file2'], "update_database $SOURCES")
```

`AllowSubstExceptions(`*`[exception, ...]`*`)`

Specifies the exceptions that will be allowed when expanding construction variables. By default, any construction variable expansions that generate a `NameError` or `IndexError` exception will expand to a `''` (an empty string) and not cause scons to fail. All exceptions not in the specified list will generate an error message and terminate processing.

If `AllowSubstExceptions` is called multiple times, each call completely overwrites the previous list of allowed exceptions.

Example:

```
# Requires that all construction variable names exist.
# (You may wish to do this if you want to enforce strictly
# that all construction variables must be defined before use.)
AllowSubstExceptions()

# Also allow a string containing a zero-division expansion
# like '${1 / 0}' to evalute to ''.
AllowSubstExceptions(IndexError, NameError, ZeroDivisionError)
```

`AlwaysBuild(`*`target, ...`*`)`
***`env`*`.AlwaysBuild(`*`target, ...`*`)`**

Marks each given *`target`* so that it is always assumed to be out of date, and will always be rebuilt if needed. Note, however, that `AlwaysBuild` does not add its target(s) to the default target list, so the targets will only be built if they are specified on the command line, or are a dependent of a target specified on the command line--but they will *always* be built if so specified. Multiple targets can be passed in to a single call to `AlwaysBuild`.

***env*.Append(*key=val, [...]*)**

Intelligently append values to construction variables in the construction environment named by `env`. The construction variables and values to add to them are passed as *`key=val`* pairs (Python keyword arguments). `env.Append` is designed to allow adding values without normally having to know the data type of an existing construction variable. Regular Python syntax can also be used to manipulate the construction variable, but for that you must know the type of the construction variable: for example, different Python syntax is needed to combine a list of values with a single string value, or vice versa. Some pre-defined construction variables do have type expectations based on how SCons will use them, for example `$CPPDEFINES` is normally a string or a list of strings, but can be a string, a list of strings, a list of tuples, or a dictionary, while `$LIBEMITTER` would expect a callable or list of callables, and `$BUILDERS` would expect a mapping type. Consult the documentation for the various construction variables for more details.

The following descriptions apply to both the append and prepend functions, the only difference being the insertion point of the added values.

If `env`. does not have a construction variable indicated by *`key`*, *`val`* is added to the environment under that key as-is.

`val` can be almost any type, and SCons will combine it with an existing value into an appropriate type, but there are a few special cases to be aware of. When two strings are combined, the result is normally a new string, with the caller responsible for supplying any needed separation. The exception to this is the construction variable `$CPPDEFINES`, in which each item will be postprocessed by adding a prefix and/or suffix, so the contents are treated as a list of strings, that is, adding a string will result in a separate string entry, not a combined string. For `$CPPDEFINES` as well as for `$LIBS`, and the various `*PATH` variables, SCons will supply the compiler-specific syntax (e.g. adding a `-D` or `/D` prefix for `$CPPDEFINES`), so this syntax should be omitted when adding values to these variables. Example (gcc syntax shown in the expansion of `CPPDEFINES`):

```
env = Environment(CXXFLAGS="-std=c11", CPPDEFINES="RELEASE")
print("CXXFLAGS={}, CPPDEFINES={}".format(env['CXXFLAGS'], env['CPPDEFINES']))
# notice including a leading space in CXXFLAGS value
env.Append(CXXFLAGS=" -O", CPPDEFINES="EXTRA")
print("CXXFLAGS={}, CPPDEFINES={}".format(env['CXXFLAGS'], env['CPPDEFINES']))
print("CPPDEFINES will expand to {}".format(env.subst("$_CPPDEFFLAGS")))
```

```
$ scons -Q
CXXFLAGS=-std=c11, CPPDEFINES=RELEASE
CXXFLAGS=-std=c11 -O, CPPDEFINES=['RELEASE', 'EXTRA']
CPPDEFINES will expand to -DRELEASE -DEXTRA
scons: `.' is up to date.
```

Because `$CPPDEFINES` is intended to describe C/C++ pre-processor macro definitions, it accepts additional syntax. Preprocessor macros can be valued, or un-valued, as in `-DBAR=1` or `-DFOO`. The macro can be be supplied as a complete string including the value, or as a tuple (or list) of macro, value, or as a dictionary. Example (again gcc syntax in the expanded defines):

```
env = Environment(CPPDEFINES="FOO")
print("CPPDEFINES={}".format(env['CPPDEFINES']))
env.Append(CPPDEFINES="BAR=1")
print("CPPDEFINES={}".format(env['CPPDEFINES']))
env.Append(CPPDEFINES=("OTHER", 2))
print("CPPDEFINES={}".format(env['CPPDEFINES']))
env.Append(CPPDEFINES={"EXTRA": "arg"})
```

```
print("CPPDEFINES={}".format(env['CPPDEFINES']))
print("CPPDEFINES will expand to {}".format(env.subst("$_CPPDEFFLAGS")))
```

```
$ scons -Q
CPPDEFINES=FOO
CPPDEFINES=['FOO', 'BAR=1']
CPPDEFINES=['FOO', 'BAR=1', ('OTHER', 2)]
CPPDEFINES=['FOO', 'BAR=1', ('OTHER', 2), {'EXTRA': 'arg'}]
CPPDEFINES will expand to -DFOO -DBAR=1 -DOTHER=2 -DEXTRA=arg
scons: `.' is up to date.
```

Adding a string *val* to a dictonary construction variable will enter *val* as the key in the dict, and `None` as its value. Using a tuple type to supply a key + value only works for the special case of `$CPPDEFINES` described above.

Although most combinations of types work without needing to know the details, some combinations do not make sense and a Python exception will be raised.

When using `env.Append` to modify construction variables which are path specifications (conventionally, the names of such end in `PATH`), it is recommended to add the values as a list of strings, even if there is only a single string to add. The same goes for adding library names to `$LIBS`.

```
env.Append(CPPPATH=["#/include"])
```

See also `env.AppendUnique`, `env.Prepend` and `env.PrependUnique`.

***env*.AppendENVPath(*name, newpath, [envname, sep, delete_existing=False]*)**

Append new path elements to the given path in the specified external environment (`$ENV` by default). This will only add any particular path once (leaving the last one it encounters and ignoring the rest, to preserve path order), and to help assure this, will normalize all paths (using `os.path.normpath` and `os.path.normcase`). 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.

If *delete_existing* is `False`, then adding a path that already exists will not move it to the end; it will stay where it is in the list.

Example:

```
print('before:', env['ENV']['INCLUDE'])
include_path = '/foo/bar:/foo'
env.AppendENVPath('INCLUDE', include_path)
print('after:', env['ENV']['INCLUDE'])
```

Yields:

```
before: /foo:/biz
after: /biz:/foo/bar:/foo
```

***env*.AppendUnique(*key=val, [...], delete_existing=False*)**

Append values to construction variables in the current construction environment, maintaining uniqueness. Works like `env.Append` (see for details), except that values already present in the construction variable will not be added again. If *delete_existing* is `True`, the existing matching value is first removed, and the requested value is added, having the effect of moving such values to the end.

Example:

```
env.AppendUnique(CCFLAGS='-g', FOO=['foo.yyy'])
```

See also `env.Append`, `env.Prepend` and `env.PrependUnique`.

`Builder(`*`action, [arguments]`*`)`
***`env`*`.Builder(`*`action, [arguments]`*`)`**

Creates a Builder object for the specified *`action`*. See the manpage section "Builder Objects" for a complete explanation of the arguments and behavior.

Note that the `env.Builder()` form of the invocation will expand construction variables in any arguments strings, including the *`action`* argument, at the time it is called using the construction variables in the `env` construction environment through which `env.Builder` was called. The `Builder` form delays all variable expansion until after the Builder object is actually called.

`CacheDir(`*`cache_dir, custom_class=None`*`)`
***`env`*`.CacheDir(`*`cache_dir, custom_class=None`*`)`**

Direct **scons** to maintain a derived-file cache in *`cache_dir`*. The derived files in the cache will be shared among all the builds specifying the same *`cache_dir`*. Specifying a *`cache_dir`* of `None` disables derived file caching.

When specifying a *`custom_class`* which should be a class type which is a subclass of `SCons.CacheDir.CacheDir`, SCons will internally invoke this class to use for performing caching operations. This argument is optional and if left to default `None`, will use the default `SCons.CacheDir.CacheDir` class.

Calling the environment method `env.CacheDir` limits the effect to targets built through the specified construction environment. Calling the global function `CacheDir` sets a global default that will be used by all targets built through construction environments that do not set up environment-specific caching by calling `env.CacheDir`.

When derived-file caching is being used and **scons** finds a derived file that needs to be rebuilt, it will first look in the cache to see if a file with matching build signature exists (indicating the input file(s) and build action(s) were identical to those for the current target), and if so, will retrieve the file from the cache. **scons** will report `` Retrieved `file' from cache `` instead of the normal build message. If the derived file is not present in the cache, **scons** will build it and then place a copy of the built file in the cache, identified by its build signature, for future use.

The `` Retrieved `file' from cache `` messages are useful for human consumption, but less so when comparing log files between **scons** runs which will show differences that are noisy and not actually significant. To disable, use the `--cache-show` option. With this option, **scons** will print the action that would have been used to build the file without considering cache retrieval.

Derived-file caching may be disabled for any invocation of **scons** by giving the `--cache-disable` command line option. Cache updating may be disabled, leaving cache fetching enabled, by giving the `--cache-readonly`.

If the `--cache-force` option is used, **scons** will place a copy of *all* derived files in the cache, even if they already existed and were not built by this invocation. This is useful to populate a cache the first time a *`cache_dir`* is used for a build, or to bring a cache up to date after a build with cache updating disabled (`--cache-disable` or `--cache-readonly`) has been done.

The `NoCache` method can be used to disable caching of specific files. This can be useful if inputs and/or outputs of some tool are impossible to predict or prohibitively large.

Clean(*targets, files_or_dirs*)
***env*.Clean(*targets, files_or_dirs*)**

This specifies a list of files or directories which should be removed whenever the targets are specified with the `-c` command line option. The specified targets may be a list or an individual target. Multiple calls to `Clean` are legal, and create new targets or add files and directories to the clean list for the specified targets.

Multiple files or directories should be specified either as separate arguments to the `Clean` method, or as a list. `Clean` will also accept the return value of any of the construction environment Builder methods. Examples:

The related `NoClean` function overrides calling `Clean` for the same target, and any targets passed to both functions will *not* be removed by the `-c` option.

Examples:

```
Clean('foo', ['bar', 'baz'])
Clean('dist', env.Program('hello', 'hello.c'))
Clean(['foo', 'bar'], 'something_else_to_clean')
```

In this example, installing the project creates a subdirectory for the documentation. This statement causes the subdirectory to be removed if the project is deinstalled.

```
Clean(docdir, os.path.join(docdir, projectname))
```

***env*.Clone(*[key=val, ...]*)**

Returns a separate copy of a construction environment. If there are any keyword arguments specified, they are added to the returned copy, overwriting any existing values for the keywords.

Example:

```
env2 = env.Clone()
env3 = env.Clone(CCFLAGS='-g')
```

Additionally, a list of tools and a toolpath may be specified, as in the `Environment` constructor:

```
def MyTool(env):
    env['FOO'] = 'bar'

env4 = env.Clone(tools=['msvc', MyTool])
```

The *`parse_flags`* keyword argument is also recognized to allow merging command-line style arguments into the appropriate construction variables (see `env.MergeFlags`).

```
# create an environment for compiling programs that use wxWidgets
wx_env = env.Clone(parse_flags='!wx-config --cflags --cxxflags')
```

Command(*target, source, action, [key=val, ...]*)
***env*.Command(*target, source, action, [key=val, ...]*)**

Executes a specific *`action`* (or list of actions) to build a *`target`* file or files from a *`source`* file or files. This is more convenient than defining a separate Builder object for a single special-case build.

The `Command` function accepts *`source_scanner`*, *`target_scanner`*, *`source_factory`*, and *`target_factory`* keyword arguments. These arguments can be used to specify a Scanner object that will be used to apply a custom scanner for a source or target. For example, the global `DirScanner` object can be used if any of the sources will be directories that must be scanned on-disk for changes to files that aren't already specified in other Builder of function calls. The *`*_factory`* arguments take a factory function that `Command` will use to turn any sources or targets specified as strings into SCons Nodes. See the manpage section "Builder Objects" for more information about how these arguments work in a Builder.

Any other keyword arguments specified override any same-named existing construction variables.

An action can be an external command, specified as a string, or a callable Python object; see the manpage section "Action Objects" for more complete information. Also note that a string specifying an external command may be preceded by an at-sign (`@`) to suppress printing the command in question, or by a hyphen (`-`) to ignore the exit status of the external command.

Examples:

```
env.Command(
    target='foo.out',
    source='foo.in',
    action="$FOO_BUILD < $SOURCES > $TARGET"
)

env.Command(
    target='bar.out',
    source='bar.in',
    action=["rm -f $TARGET", "$BAR_BUILD < $SOURCES > $TARGET"],
    ENV={'PATH': '/usr/local/bin/'},
)


import os
def rename(env, target, source):
    os.rename('.tmp', str(target[0]))


env.Command(
    target='baz.out',
    source='baz.in',
    action=["$BAZ_BUILD < $SOURCES > .tmp", rename],
)
```

Note that the `Command` function will usually assume, by default, that the specified targets and/or sources are Files, if no other part of the configuration identifies what type of entries they are. If necessary, you can explicitly specify that targets or source nodes should be treated as directories by using the `Dir` or `env.Dir` functions.

Examples:

```
env.Command('ddd.list', Dir('ddd'), 'ls -l $SOURCE > $TARGET')

env['DISTDIR'] = 'destination/directory'
env.Command(env.Dir('$DISTDIR')), None, make_distdir)
```

Also note that SCons will usually automatically create any directory necessary to hold a target file, so you normally don't need to create directories by hand.

Configure(*env, [custom_tests, conf_dir, log_file, config_h]*)
***env*.Configure(*[custom_tests, conf_dir, log_file, config_h]*)**

Creates a Configure object for integrated functionality similar to GNU autoconf. See the manpage section "Configure Contexts" for a complete explanation of the arguments and behavior.

Decider(*function*)
***env*.Decider(*function*)**

Specifies that all up-to-date decisions for targets built through this construction environment will be handled by the specified *`function`*. *`function`* can be the name of a function or one of the following strings that specify the predefined decision function that will be applied:

"timestamp-newer"
: Specifies that a target shall be considered out of date and rebuilt if the dependency's timestamp is newer than the target file's timestamp. This is the behavior of the classic Make utility, and `make` can be used a synonym for `timestamp-newer`.

"timestamp-match"
: Specifies that a target shall be considered out of date and rebuilt if the dependency's timestamp is different than the timestamp recorded the last time the target was built. This provides behavior very similar to the classic Make utility (in particular, files are not opened up so that their contents can be checksummed) except that the target will also be rebuilt if a dependency file has been restored to a version with an *earlier* timestamp, such as can happen when restoring files from backup archives.

"content"
: Specifies that a target shall be considered out of date and rebuilt if the dependency's content has changed since the last time the target was built, as determined be performing an checksum on the dependency's contents and comparing it to the checksum recorded the last time the target was built. `MD5` can be used as a synonym for `content`, but it is deprecated.

"content-timestamp"
: Specifies that a target shall be considered out of date and rebuilt if the dependency's content has changed since the last time the target was built, except that dependencies with a timestamp that matches the last time the target was rebuilt will be assumed to be up-to-date and *not* rebuilt. This provides behavior very similar to the `content` behavior of always checksumming file contents, with an optimization of not checking the contents of files whose timestamps haven't changed. The drawback is that SCons will *not* detect if a file's content has changed but its timestamp is the same, as might happen in an automated script that runs a build, updates a file, and runs the build again, all within a single second. `MD5-timestamp` can be used as a synonym for `content-timestamp`, but it is deprecated.

Examples:

```
# Use exact timestamp matches by default.
Decider('timestamp-match')

# Use hash content signatures for any targets built
# with the attached construction environment.
env.Decider('content')
```

In addition to the above already-available functions, the *`function`* argument may be a Python function you supply. Such a function must accept the following four arguments:

dependency
: The Node (file) which should cause the *`target`* to be rebuilt if it has "changed" since the last tme *`target`* was built.

target
The Node (file) being built. In the normal case, this is what should get rebuilt if the *dependency* has "changed."

prev_ni
Stored information about the state of the *dependency* the last time the *target* was built. This can be consulted to match various file characteristics such as the timestamp, size, or content signature.

repo_node
If set, use this Node instead of the one specified by *dependency* to determine if the dependency has changed. This argument is optional so should be written as a default argument (typically it would be written as *repo_node=None*). A caller will normally only set this if the target only exists in a Repository.

The *function* should return a value which evaluates `True` if the *dependency* has "changed" since the last time the *target* was built (indicating that the target *should* be rebuilt), and a value which evaluates `False` otherwise (indicating that the target should *not* be rebuilt). Note that the decision can be made using whatever criteria are appopriate. Ignoring some or all of the function arguments is perfectly normal.

Example:

```
def my_decider(dependency, target, prev_ni, repo_node=None):
    return not os.path.exists(str(target))

env.Decider(my_decider)
```

Default(*target[, ...]*)
***env*.Default(*target[, ...]*)**
Specify default targets to the SCons target selection mechanism. Any call to `Default` will cause SCons to use the defined default target list instead of its built-in algorithm for determining default targets (see the manpage section "Target Selection").

target may be one or more strings, a list of strings, a `NodeList` as returned by a Builder, or `None`. A string *target* may be the name of a file or directory, or a target previously defined by a call to `Alias` (defining the alias later will still create the alias, but it will not be recognized as a default). Calls to `Default` are additive. A *target* of `None` will clear any existing default target list; subsequent calls to `Default` will add to the (now empty) default target list like normal.

Both forms of this call affect the same global list of default targets; the construction environment method applies construction variable expansion to the targets.

The current list of targets added using `Default` is available in the `DEFAULT_TARGETS` list (see below).

Examples:

```
Default('foo', 'bar', 'baz')
env.Default(['a', 'b', 'c'])
hello = env.Program('hello', 'hello.c')
env.Default(hello)
```

DefaultEnvironment(*[kwargs]*)**
Instantiates and returns the default construction environment object. The default environment is used internally by SCons in order to execute many of the global functions in this list (that is, those not called as methods of a specific construction environment). It is not mandatory to call `DefaultEnvironment`: the default environment will be instantiated automatically when the build phase begins if the function has not been called, however calling it explicitly gives the opportunity to affect and examine the contents of the default environment.

The default environment is a singleton, so the keyword arguments affect it only on the first call, on subsequent calls the already-constructed object is returned and any keyword arguments are silently ignored. The default environment can be modified after instantiation in the same way as any construction environment. Modifying the default environment has no effect on the construction environment constructed by an `Environment` or `Clone` call.

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

Specifies an explicit dependency; the *`target`* will be rebuilt whenever the *`dependency`* has changed. Both the specified *`target`* and *`dependency`* can be a string (usually the path name of a file or directory) or Node objects, or a list of strings or Node objects (such as returned by a Builder call). This should only be necessary for cases where the dependency is not caught by a Scanner for the file.

Example:

```
env.Depends('foo', 'other-input-file-for-foo')

mylib = env.Library('mylib.c')
installed_lib = env.Install('lib', mylib)
bar = env.Program('bar.c')

# Arrange for the library to be copied into the installation
# directory before trying to build the "bar" program.
# (Note that this is for example only.  A "real" library
# dependency would normally be configured through the $LIBS
# and $LIBPATH variables, not using an env.Depends() call.)

env.Depends(bar, installed_lib)
```

***`env`*`.Detect(`*`progs`*`)`**

Find an executable from one or more choices: *`progs`* may be a string or a list of strings. Returns the first value from *`progs`* that was found, or `None`. Executable is searched by checking the paths specified by `env['ENV']['PATH']`. On Windows systems, additionally applies the filename suffixes found in `env['ENV']['PATHEXT']` but will not include any such extension in the return value. `env.Detect` is a wrapper around `env.WhereIs`.

***`env`*`.Dictionary(`*`[vars]`*`)`**

Returns a dictionary object containing the construction variables in the construction environment. If there are any arguments specified, the values of the specified construction variables are returned as a string (if one argument) or as a list of strings.

Example:

```
cvars = env.Dictionary()
cc_values = env.Dictionary('CC', 'CCFLAGS', 'CCCOM')
```

`Dir(`*`name, [directory]`*`)`
***`env`*`.Dir(`*`name, [directory]`*`)`**

Returns Directory Node(s). A Directory Node is an object that represents a directory. *`name`* can be a relative or absolute path or a list of such paths. *`directory`* is an optional directory that will be used as the parent directory. If no *`directory`* is specified, the current script's directory is used as the parent.

If *`name`* is a single pathname, the corresponding node is returned. If *`name`* is a list, SCons returns a list of nodes. Construction variables are expanded in *`name`*.

Directory Nodes can be used anywhere you would supply a string as a directory name to a Builder method or function. Directory Nodes have attributes and methods that are useful in many situations; see manpage section "File and Directory Nodes" for more information.

***env*.Dump(*[key], [format]*)**

Serializes construction variables to a string. The method supports the following formats specified by *`format`*:

`pretty`

Returns a pretty printed representation of the environment (if *`format`* is not specified, this is the default).

`json`

Returns a JSON-formatted string representation of the environment.

If `key` is `None` (the default) the entire dictionary of construction variables is serialized. If supplied, it is taken as the name of a construction variable whose value is serialized.

This SConstruct:

```
env=Environment()
print(env.Dump('CCCOM'))
```

will print:

```
'$CC -c -o $TARGET $CCFLAGS $CPPFLAGS $_CPPDEFFLAGS $_CPPINCFLAGS $SOURCES'
```

While this SConstruct:

```
env = Environment()
print(env.Dump())
```

will print:

```
{ 'AR': 'ar',
  'ARCOM': '$AR $ARFLAGS $TARGET $SOURCES\n$RANLIB $RANLIBFLAGS $TARGET',
  'ARFLAGS': ['r'],
  'AS': 'as',
  'ASCOM': '$AS $ASFLAGS -o $TARGET $SOURCES',
  'ASFLAGS': [],
  ...
```

EnsurePythonVersion(*major, minor*)
***env*.EnsurePythonVersion(*major, minor*)**

Ensure that the Python version is at least `major`.`minor`. This function will print out an error message and exit SCons with a non-zero exit code if the actual Python version is not late enough.

Example:

```
EnsurePythonVersion(2,2)
```

EnsureSConsVersion(*major, minor, [revision]*)
***env*.EnsureSConsVersion(*major, minor, [revision]*)**

Ensure that the SCons version is at least `major.minor`, or `major.minor.revision`. if `revision` is specified. This function will print out an error message and exit SCons with a non-zero exit code if the actual SCons version is not late enough.

Examples:

```
EnsureSConsVersion(0,14)

EnsureSConsVersion(0,96,90)
```

Environment(*[key=value, ...]*)
***env*.Environment(*[key=value, ...]*)**

Return a new construction environment initialized with the specified `key`=`value` pairs. The keyword arguments *`parse_flags`*, *`platform`*, *`toolpath`*, *`tools`* and *`variables`* are also specially recognized. See the manpage section "Construction Environments" for more details.

Execute(*action, [strfunction, varlist]*)
***env*.Execute(*action, [strfunction, varlist]*)**

Executes an Action object. The specified *`action`* may be an Action object (see manpage section "Action Objects" for an explanation of behavior), or it may be a command-line string, list of commands, or executable Python function, each of which will be converted into an Action object and then executed. Any additional arguments to `Execute` (*`strfunction`*, *`varlist`*) are passed on to the `Action` factory function which actually creates the Action object. The exit value of the command or return value of the Python function will be returned.

Note that **scons** will print an error message if the executed *`action`* fails--that is, exits with or returns a non-zero value. **scons** will *not*, however, automatically terminate the build if the specified *`action`* fails. If you want the build to stop in response to a failed `Execute` call, you must explicitly check for a non-zero return value:

```
Execute(Copy('file.out', 'file.in'))

if Execute("mkdir sub/dir/ectory"):
    # The mkdir failed, don't try to build.
    Exit(1)
```

Exit(*[value]*)
***env*.Exit(*[value]*)**

This tells **scons** to exit immediately with the specified `value`. A default exit value of `0` (zero) is used if no value is specified.

Export(*[vars...], [key=value...]*)
***env*.Export(*[vars...], [key=value...]*)**

Exports variables from the current SConscript file to a global collection where they can be imported by other SConscript files. *`vars`* may be one or more strings representing variable names to be exported. If a string contains whitespace, it is split into separate strings, as if multiple string arguments had been given. A *`vars`* argument may also be a dictionary, which can be used to map variables to different names when exported. Keyword arguments can be used to provide names and their values.

`Export` calls are cumulative. Specifying a previously exported variable will overwrite the earlier value. Both local variables and global variables can be exported.

Examples:

```
env = Environment()
# Make env available for all SConscript files to Import().
Export("env")
```

```
package = 'my_name'
# Make env and package available for all SConscript files:.
Export("env", "package")

# Make env and package available for all SConscript files:
Export(["env", "package"])

# Make env available using the name debug:
Export(debug=env)

# Make env available using the name debug:
Export({"debug": env})
```

Note that the `SConscript` function supports an `exports` argument that allows exporting a variable or set of variables to a specific SConscript file or files. See the description below.

`File(`*`name, [directory]`*`)`
***`env`*`.File(`*`name, [directory]`*`)`**

Returns File Node(s). A File Node is an object that represents a file. *`name`* can be a relative or absolute path or a list of such paths. *`directory`* is an optional directory that will be used as the parent directory. If no *`directory`* is specified, the current script's directory is used as the parent.

If *`name`* is a single pathname, the corresponding node is returned. If *`name`* is a list, SCons returns a list of nodes. Construction variables are expanded in *`name`*.

File Nodes can be used anywhere you would supply a string as a file name to a Builder method or function. File Nodes have attributes and methods that are useful in many situations; see manpage section "File and Directory Nodes" for more information.

`FindFile(`*`file, dirs`*`)`
***`env`*`.FindFile(`*`file, dirs`*`)`**

Search for *`file`* in the path specified by *`dirs`*. *`dirs`* may be a list of directory names or a single directory name. In addition to searching for files that exist in the filesystem, this function also searches for derived files that have not yet been built.

Example:

```

foo = env.FindFile('foo', ['dir1', 'dir2'])
```

`FindInstalledFiles()`
***`env`*`.FindInstalledFiles()`**

Returns the list of targets set up by the `Install` or `InstallAs` builders.

This function serves as a convenient method to select the contents of a binary package.

Example:

```

Install('/bin', ['executable_a', 'executable_b'])

# will return the file node list
# ['/bin/executable_a', '/bin/executable_b']
FindInstalledFiles()

Install('/lib', ['some_library'])
```

```
# will return the file node list
# ['/bin/executable_a', '/bin/executable_b', '/lib/some_library']
FindInstalledFiles()
```

`FindPathDirs(`*`variable`*`)`

Returns a function (actually a callable Python object) intended to be used as the `path_function` of a Scanner object. The returned object will look up the specified `variable` in a construction environment and treat the construction variable's value as a list of directory paths that should be searched (like `$CPPPATH`, `$LIBPATH`, etc.).

Note that use of `FindPathDirs` is generally preferable to writing your own `path_function` for the following reasons: 1) The returned list will contain all appropriate directories found in source trees (when `VariantDir` is used) or in code repositories (when `Repository` or the `-Y` option are used). 2) scons will identify expansions of `variable` that evaluate to the same list of directories as, in fact, the same list, and avoid re-scanning the directories for files, when possible.

Example:

```
def my_scan(node, env, path, arg):
    # Code to scan file contents goes here...
    return include_files

scanner = Scanner(name = 'myscanner',
                  function = my_scan,
                  path_function = FindPathDirs('MYPATH'))
```

`FindSourceFiles(`*`node`*`=`*`'"."'`*`)`
***env*.`FindSourceFiles(`*`node`*`=`*`'"."'`*`)`**

Returns the list of nodes which serve as the source of the built files. It does so by inspecting the dependency tree starting at the optional argument *`node`* which defaults to the '"."'-node. It will then return all leaves of *`node`*. These are all children which have no further children.

This function is a convenient method to select the contents of a Source Package.

Example:

```
Program('src/main_a.c')
Program('src/main_b.c')
Program('main_c.c')

# returns ['main_c.c', 'src/main_a.c', 'SConstruct', 'src/main_b.c']
FindSourceFiles()

# returns ['src/main_b.c', 'src/main_a.c' ]
FindSourceFiles('src')
```

As you can see build support files (SConstruct in the above example) will also be returned by this function.

`Flatten(`*`sequence`*`)`
***env*.`Flatten(`*`sequence`*`)`**

Takes a sequence (that is, a Python list or tuple) that may contain nested sequences and returns a flattened list containing all of the individual elements in any sequence. This can be helpful for collecting the lists returned by calls to Builders; other Builders will automatically flatten lists specified as input, but direct Python manipulation of these lists does not.

Examples:

```
foo = Object('foo.c')
bar = Object('bar.c')

# Because `foo' and `bar' are lists returned by the Object() Builder,
# `objects' will be a list containing nested lists:
objects = ['f1.o', foo, 'f2.o', bar, 'f3.o']

# Passing such a list to another Builder is all right because
# the Builder will flatten the list automatically:
Program(source = objects)

# If you need to manipulate the list directly using Python, you need to
# call Flatten() yourself, or otherwise handle nested lists:
for object in Flatten(objects):
    print(str(object))
```

`GetBuildFailures`()

Returns a list of exceptions for the actions that failed while attempting to build targets. Each element in the returned list is a `BuildError` object with the following attributes that record various aspects of the build failure:

`.node` The node that was being built when the build failure occurred.

`.status` The numeric exit status returned by the command or Python function that failed when trying to build the specified Node.

`.errstr` The SCons error string describing the build failure. (This is often a generic message like "Error 2" to indicate that an executed command exited with a status of 2.)

`.filename` The name of the file or directory that actually caused the failure. This may be different from the `.node` attribute. For example, if an attempt to build a target named `sub/dir/target` fails because the `sub/dir` directory could not be created, then the `.node` attribute will be `sub/dir/target` but the `.filename` attribute will be `sub/dir`.

`.executor` The SCons Executor object for the target Node being built. This can be used to retrieve the construction environment used for the failed action.

`.action` The actual SCons Action object that failed. This will be one specific action out of the possible list of actions that would have been executed to build the target.

`.command` The actual expanded command that was executed and failed, after expansion of `$TARGET`, `$SOURCE`, and other construction variables.

Note that the `GetBuildFailures` function will always return an empty list until any build failure has occurred, which means that `GetBuildFailures` will always return an empty list while the `SConscript` files are being read. Its primary intended use is for functions that will be executed before SCons exits by passing them to the standard Python `atexit.register`() function. Example:

```
import atexit
```

```
def print_build_failures():
    from SCons.Script import GetBuildFailures
    for bf in GetBuildFailures():
        print("%s failed: %s" % (bf.node, bf.errstr))

atexit.register(print_build_failures)
```

GetBuildPath(*file, [...]*)
***env*.GetBuildPath(*file, [...]*)**

Returns the **scons** path name (or names) for the specified *file* (or files). The specified *file* or files may be **scons** Nodes or strings representing path names.

GetLaunchDir()
***env*.GetLaunchDir()**

Returns the absolute path name of the directory from which **scons** was initially invoked. This can be useful when using the `-u`, `-U` or `-D` options, which internally change to the directory in which the `SConstruct` file is found.

GetOption(*name*)
***env*.GetOption(*name*)**

This function provides a way to query the value of options which can be set via the command line or using the `SetOption` function.

name can be an entry from the following table, which shows the corresponding command line arguments that could affect the value. *name* can be also be the destination variable name from a project-specific option added using the `AddOption` function, as long as the addition happens prior to the `GetOption` call in the SConscript files.

Query name	Command-line options	Notes
`cache_debug`	`--cache-debug`	
`cache_disable`	`--cache-disable`, `--no-cache`	
`cache_force`	`--cache-force`, `--cache-populate`	
`cache_readonly`	`--cache-readonly`	
`cache_show`	`--cache-show`	
`clean`	`-c`, `--clean`, `--remove`	
`climb_up`	`-D` `-U` `-u` `--up` `--search_up`	
`config`	`--config`	
`debug`	`--debug`	
`directory`	`-C`, `--directory`	
`diskcheck`	`--diskcheck`	
`duplicate`	`--duplicate`	
`enable_virtualenv`	`--enable-virtualenv`	
`experimental`	`--experimental`	*since 4.2*
`file`	`-f`, `--file`, `--makefile`, `--sconstruct`	
`hash_format`	`--hash-format`	*since 4.2*
`help`	`-h`, `--help`	

Query name	Command-line options	Notes
`ignore_errors`	`-i`, `--ignore-errors`	
`ignore_virtualenv`	`--ignore-virtualenv`	
`implicit_cache`	`--implicit-cache`	
`implicit_deps_changed`	`--implicit-deps-changed`	
`implicit_deps_unchanged`	`--implicit-deps-unchanged`	
`include_dir`	`-I`, `--include-dir`	
`install_sandbox`	`--install-sandbox`	Available only if the `install` tool has been called
`keep_going`	`-k`, `--keep-going`	
`max_drift`	`--max-drift`	
`md5_chunksize`	`--hash-chunksize`, `--md5-chunksize`	*`--hash-chunksize` since 4.2*
`no_exec`	`-n`, `--no-exec`, `--just-print`, `--dry-run`, `--recon`	
`no_progress`	`-Q`	
`num_jobs`	`-j`, `--jobs`	
`package_type`	`--package-type`	Available only if the `packaging` tool has been called
`profile_file`	`--profile`	
`question`	`-q`, `--question`	
`random`	`--random`	
`repository`	`-Y`, `--repository`, `--srcdir`	
`silent`	`-s`, `--silent`, `--quiet`	
`site_dir`	`--site-dir`, `--no-site-dir`	
`stack_size`	`--stack-size`	
`taskmastertrace_file`	`--taskmastertrace`	
`tree_printers`	`--tree`	
`warn`	`--warn`, `--warning`	

See the documentation for the corresponding command line option for information about each specific option.

Glob(*pattern, [ondisk, source, strings, exclude]*)
***env*.Glob(*pattern, [ondisk, source, strings, exclude]*)**

Returns Nodes (or strings) that match the specified *`pattern`*, relative to the directory of the current `SConscript` file. The evironment method form (`env.Glob`) performs string substition on *`pattern`* and returns whatever matches the resulting expanded pattern.

The specified *`pattern`* uses Unix shell style metacharacters for matching:

```
*       matches everything
?       matches any single character
[seq]   matches any character in seq
```

```
  [!seq]  matches any char not in seq
```

If the first character of a filename is a dot, it must be matched explicitly. Character matches do *not* span directory separators.

The `Glob` knows about repositories (see the `Repository` function) and source directories (see the `VariantDir` function) and returns a Node (or string, if so configured) in the local (SConscript) directory if a matching Node is found anywhere in a corresponding repository or source directory.

The *`ondisk`* argument may be set to a value which evaluates `False` to disable the search for matches on disk, thereby only returning matches among already-configured File or Dir Nodes. The default behavior is to return corresponding Nodes for any on-disk matches found.

The *`source`* argument may be set to a value which evaluates `True` to specify that, when the local directory is a `VariantDir`, the returned Nodes should be from the corresponding source directory, not the local directory.

The *`strings`* argument may be set to a value which evaluates `True` to have the `Glob` function return strings, not Nodes, that represent the matched files or directories. The returned strings will be relative to the local (SConscript) directory. (Note that This may make it easier to perform arbitrary manipulation of file names, but if the returned strings are passed to a different `SConscript` file, any Node translation will be relative to the other `SConscript` directory, not the original `SConscript` directory.)

The *`exclude`* argument may be set to a pattern or a list of patterns (following the same Unix shell semantics) which must be filtered out of returned elements. Elements matching a least one pattern of this list will be excluded.

Examples:

```
Program("foo", Glob("*.c"))
Zip("/tmp/everything", Glob(".??*") + Glob("*"))
sources = Glob("*.cpp", exclude=["os_*_specific_*.cpp"]) + \
          Glob( "os_%s_specific_*.cpp" % currentOS)
```

`Help`(*`text, append=False`*)
***`env`*.`Help`(*`text, append=False`*)**

Specifies a local help message to be printed if the `-h` argument is given to **scons**. Subsequent calls to `Help` append *`text`* to the previously defined local help text.

For the first call to `Help` only, if *`append`* is `False` (the default) any local help message generated through `AddOption` calls is replaced. If *`append`* is `True`, *`text`* is appended to the existing help text.

`Ignore`(*`target, dependency`*)
***`env`*.`Ignore`(*`target, dependency`*)**

The specified dependency file(s) will be ignored when deciding if the target file(s) need to be rebuilt.

You can also use `Ignore` to remove a target from the default build. In order to do this you must specify the directory the target will be built in as the target, and the file you want to skip building as the dependency.

Note that this will only remove the dependencies listed from the files built by default. It will still be built if that dependency is needed by another object being built. See the third and forth examples below.

Examples:

```
env.Ignore('foo', 'foo.c')
env.Ignore('bar', ['bar1.h', 'bar2.h'])
env.Ignore('.', 'foobar.obj')
env.Ignore('bar', 'bar/foobar.obj')
```

Import(*vars...*)
***env*.Import(*vars...*)**

Imports variables into the current SConscript file. *vars* must be strings representing names of variables which have been previously exported either by the `Export` function or by the `exports` argument to `SConscript`. Variables exported by `SConscript` take precedence. Multiple variable names can be passed to `Import` as separate arguments or as words in a space-separated string. The wildcard `"*"` can be used to import all available variables.

Examples:

```
Import("env")
Import("env", "variable")
Import(["env", "variable"])
Import("*")
```

Literal(*string*)
***env*.Literal(*string*)**

The specified *string* will be preserved as-is and not have construction variables expanded.

Local(*targets*)
***env*.Local(*targets*)**

The specified *targets* will have copies made in the local tree, even if an already up-to-date copy exists in a repository. Returns a list of the target Node or Nodes.

***env*.MergeFlags(*arg, [unique]*)**

Merges values from *arg* into construction variables in the current construction environment. If *arg* is not a dictionary, it is converted to one by calling `env.ParseFlags` on the argument before the values are merged. Note that *arg* must be a single value, so multiple strings must be passed in as a list, not as separate arguments to `env.MergeFlags`.

By default, duplicate values are eliminated; you can, however, specify `unique=False` to allow duplicate values to be added. When eliminating duplicate values, any construction variables that end with the string `PATH` keep the left-most unique value. All other construction variables keep the right-most unique value.

Examples:

```
# Add an optimization flag to $CCFLAGS.
env.MergeFlags('-O3')

# Combine the flags returned from running pkg-config with an optimization
# flag and merge the result into the construction variables.
env.MergeFlags(['!pkg-config gtk+-2.0 --cflags', '-O3'])

# Combine an optimization flag with the flags returned from running pkg-config
# twice and merge the result into the construction variables.
env.MergeFlags(['-O3',
               '!pkg-config gtk+-2.0 --cflags --libs',
               '!pkg-config libpng12 --cflags --libs'])
```

NoCache(*target, ...*)
***env*.NoCache(*target, ...*)**

Specifies a list of files which should *not* be cached whenever the `CacheDir` method has been activated. The specified targets may be a list or an individual target.

Multiple files should be specified either as separate arguments to the `NoCache` method, or as a list. `NoCache` will also accept the return value of any of the construction environment Builder methods.

Calling `NoCache` on directories and other non-File Node types has no effect because only File Nodes are cached.

Examples:

```
NoCache('foo.elf')
NoCache(env.Program('hello', 'hello.c'))
```

`NoClean(`*`target, ...`*`)`
***`env`*`.NoClean(`*`target, ...`*`)`**

Specifies a list of files or directories which should *not* be removed whenever the targets (or their dependencies) are specified with the `-c` command line option. The specified targets may be a list or an individual target. Multiple calls to `NoClean` are legal, and prevent each specified target from being removed by calls to the `-c` option.

Multiple files or directories should be specified either as separate arguments to the `NoClean` method, or as a list. `NoClean` will also accept the return value of any of the construction environment Builder methods.

Calling `NoClean` for a target overrides calling `Clean` for the same target, and any targets passed to both functions will *not* be removed by the `-c` option.

Examples:

```
NoClean('foo.elf')
NoClean(env.Program('hello', 'hello.c'))
```

***`env`*`.ParseConfig(`*`command, [function, unique]`*`)`**

Updates the current construction environment with the values extracted from the output from running external *`command`*, by calling a helper function *`function`* which understands the output of *`command`*. *`command`* may be a string or a list of strings representing the command and its arguments. If *`function`* is not given, `env.MergeFlags` is used. By default, duplicate values are not added to any construction variables; you can specify *`unique=False`* to allow duplicate values to be added.

If `env.MergeFlags` is used, it expects a response in the style of a ***-config** command typical of the POSIX programming environment (for example, gtk-config) and adds the options to the appropriate construction variables. Interpreted options and the construction variables they affect are as specified for the `env.ParseFlags` method (which `env.MergeFlags` calls). See that method's description for a table of options and corresponding construction variables.

If `env.MergeFlags` cannot interpret the results of *`command`*, you can suppply a custom *`function`* to do so. *`function`* must accept three arguments: the construction environment to modify, the string returned by running *`command`*, and the optional *`unique`* flag.

`ParseDepends(`*`filename, [must_exist, only_one]`*`)`
***`env`*`.ParseDepends(`*`filename, [must_exist, only_one]`*`)`**

Parses the contents of the specified *`filename`* as a list of dependencies in the style of Make or mkdep, and explicitly establishes all of the listed dependencies.

By default, it is not an error if the specified *`filename`* does not exist. The optional *`must_exist`* argument may be set to a non-zero value to have scons throw an exception and generate an error if the file does not exist, or is otherwise inaccessible.

The optional *`only_one`* argument may be set to a non-zero value to have scons thrown an exception and generate an error if the file contains dependency information for more than one target. This can provide a small sanity check for files intended to be generated by, for example, the `gcc -M` flag, which should typically only write dependency information for one output file into a corresponding `.d` file.

The *`filename`* and all of the files listed therein will be interpreted relative to the directory of the `SConscript` file which calls the `ParseDepends` function.

***env*.ParseFlags(*flags, ...*)**

Parses one or more strings containing typical command-line flags for GCC tool chains and returns a dictionary with the flag values separated into the appropriate SCons construction variables. This is intended as a companion to the `env.MergeFlags` method, but allows for the values in the returned dictionary to be modified, if necessary, before merging them into the construction environment. (Note that `env.MergeFlags` will call this method if its argument is not a dictionary, so it is usually not necessary to call `env.ParseFlags` directly unless you want to manipulate the values.)

If the first character in any string is an exclamation mark (!), the rest of the string is executed as a command, and the output from the command is parsed as GCC tool chain command-line flags and added to the resulting dictionary.

Flag values are translated accordig to the prefix found, and added to the following construction variables:

```
-arch                   CCFLAGS, LINKFLAGS
-D                      CPPDEFINES
-framework              FRAMEWORKS
-frameworkdir=          FRAMEWORKPATH
-fmerge-all-constants   CCFLAGS, LINKFLAGS
-fopenmp                CCFLAGS, LINKFLAGS
-include                CCFLAGS
-imacros                CCFLAGS
-isysroot               CCFLAGS, LINKFLAGS
-isystem                CCFLAGS
-iquote                 CCFLAGS
-idirafter              CCFLAGS
-I                      CPPPATH
-l                      LIBS
-L                      LIBPATH
-mno-cygwin             CCFLAGS, LINKFLAGS
-mwindows               LINKFLAGS
-openmp                 CCFLAGS, LINKFLAGS
-pthread                CCFLAGS, LINKFLAGS
-std=                   CFLAGS
-Wa,                    ASFLAGS, CCFLAGS
-Wl,-rpath=             RPATH
-Wl,-R,                 RPATH
-Wl,-R                  RPATH
-Wl,                    LINKFLAGS
-Wp,                    CPPFLAGS
-                       CCFLAGS
+                       CCFLAGS, LINKFLAGS
```

Any other strings not associated with options are assumed to be the names of libraries and added to the `$LIBS` construction variable.

Examples (all of which produce the same result):

```
dict = env.ParseFlags('-O2 -Dfoo -Dbar=1')
dict = env.ParseFlags('-O2', '-Dfoo', '-Dbar=1')
dict = env.ParseFlags(['-O2', '-Dfoo -Dbar=1'])
dict = env.ParseFlags('-O2', '!echo -Dfoo -Dbar=1')
```

`Platform(`*`plat`*`)`
***`env`*`.Platform(`*`plat`*`)`**

When called as a global function, returns a callable platform object selected by *`plat`* (defaults to the detected platform for the current system) that can be used to initialize a construction environment by passing it as the *`platform`* keyword argument to the `Environment` function.

Example:

```
env = Environment(platform=Platform('win32'))
```

When called as a method of an environment, calls the platform object indicated by *`plat`* to update that environment.

```
env.Platform('posix')
```

See the manpage section "Construction Environments" for more details.

`Precious(`*`target, ...`*`)`
***`env`*`.Precious(`*`target, ...`*`)`**

Marks each given `target` as precious so it is not deleted before it is rebuilt. Normally **scons** deletes a target before building it. Multiple targets can be passed in to a single call to `Precious`.

***`env`*`.Prepend(`*`key=val, [...]`*`)`**

Prepend values to construction variables in the current construction environment, Works like `env.Append` (see for details), except that values are added to the front, rather than the end, of any existing value of the construction variable

Example:

```
env.Prepend(CCFLAGS='-g ', FOO=['foo.yyy'])
```

See also `env.Append`, `env.AppendUnique` and `env.PrependUnique`.

***`env`*`.PrependENVPath(`*`name, newpath, [envname, sep, delete_existing]`*`)`**

Prepend new path elements to the given path in the specified external environment (`$ENV` by default). This will only add any particular path once (leaving the first one it encounters and ignoring the rest, to preserve path order), and to help assure this, will normalize all paths (using `os.path.normpath` and `os.path.normcase`). 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.

If *`delete_existing`* is `False`, then adding a path that already exists will not move it to the beginning; it will stay where it is in the list.

Example:

```
print('before:', env['ENV']['INCLUDE'])
include_path = '/foo/bar:/foo'
env.PrependENVPath('INCLUDE', include_path)
```

```
print('after:', env['ENV']['INCLUDE'])
```

Yields:

```
before: /biz:/foo
after: /foo/bar:/foo:/biz
```

***env*.PrependUnique(*key=val, delete_existing=False, [...]*)**

Prepend values to construction variables in the current construction environment, maintaining uniqueness. Works like `env.Append` (see for details), except that values are added to the front, rather than the end, of any existing value of the the construction variable, and values already present in the construction variable will not be added again. If *`delete_existing`* is `True`, the existing matching value is first removed, and the requested value is inserted, having the effect of moving such values to the front.

Example:

```
env.PrependUnique(CCFLAGS='-g', FOO=['foo.yyy'])
```

See also `env.Append`, `env.AppendUnique` and `env.Prepend`.

Progress(*callable, [interval]*)
Progress(*string, [interval, file, overwrite]*)
Progress(*list_of_strings, [interval, file, overwrite]*)

Allows SCons to show progress made during the build by displaying a string or calling a function while evaluating Nodes (e.g. files).

If the first specified argument is a Python callable (a function or an object that has a `__call__` method), the function will be called once every `interval` times a Node is evaluated (default `1`). The callable will be passed the evaluated Node as its only argument. (For future compatibility, it's a good idea to also add *`*args`* and *`**kwargs`* as arguments to your function or method signatures. This will prevent the code from breaking if SCons ever changes the interface to call the function with additional arguments in the future.)

An example of a simple custom progress function that prints a string containing the Node name every 10 Nodes:

```
def my_progress_function(node, *args, **kwargs):
    print('Evaluating node %s!' % node)
Progress(my_progress_function, interval=10)
```

A more complicated example of a custom progress display object that prints a string containing a count every 100 evaluated Nodes. Note the use of `\r` (a carriage return) at the end so that the string will overwrite itself on a display:

```
import sys
class ProgressCounter(object):
    count = 0
    def __call__(self, node, *args, **kw):
        self.count += 100
        sys.stderr.write('Evaluated %s nodes\r' % self.count)

Progress(ProgressCounter(), interval=100)
```

If the first argument to `Progress` is a string or list of strings, it is taken as text to be displayed every `interval` evaluated Nodes. If the first argument is a list of strings, then each string in the list will be displayed in rotating fashion every `interval` evaluated Nodes.

The default is to print the string on standard output. An alternate output stream may be specified with the *`file`* keyword argument, which the caller must pass already opened.

The following will print a series of dots on the error output, one dot for every 100 evaluated Nodes:

```
import sys
Progress('.', interval=100, file=sys.stderr)
```

If the string contains the verbatim substring `$TARGET;`, it will be replaced with the Node. Note that, for performance reasons, this is *not* a regular SCons variable substition, so you can not use other variables or use curly braces. The following example will print the name of every evaluated Node, using a carriage return) (`\r`) to cause each line to overwritten by the next line, and the *`overwrite`* keyword argument (default `False`) to make sure the previously-printed file name is overwritten with blank spaces:

```
import sys
Progress('$TARGET\r', overwrite=True)
```

A list of strings can be used to implement a "spinner" on the user's screen as follows, changing every five evaluated Nodes:

```
Progress(['-\r', '\\\r', '|\r', '/\r'], interval=5)
```

`Pseudo(`*`target, ...`*`)`
***`env`*`.Pseudo(`*`target, ...`*`)`**

This indicates that each given `target` should not be created by the build rule, and if the target is created, an error will be generated. This is similar to the gnu make .PHONY target. However, in the vast majority of cases, an `Alias` is more appropriate. Multiple targets can be passed in to a single call to `Pseudo`.

`PyPackageDir(`*`modulename`*`)`
***`env`*`.PyPackageDir(`*`modulename`*`)`**

This returns a Directory Node similar to Dir. The python module / package is looked up and if located the directory is returned for the location. *`modulename`* Is a named python package / module to lookup the directory for it's location.

If *`modulename`* is a list, SCons returns a list of Dir nodes. Construction variables are expanded in *`modulename`*.

***`env`*`.Replace(`*`key=val, [...]`*`)`**

Replaces construction variables in the Environment with the specified keyword arguments.

Example:

```
env.Replace(CCFLAGS='-g', FOO='foo.xxx')
```

`Repository(`*`directory`*`)`
***`env`*`.Repository(`*`directory`*`)`**

Specifies that *`directory`* is a repository to be searched for files. Multiple calls to `Repository` are legal, and each one adds to the list of repositories that will be searched.

To **scons**, a repository is a copy of the source tree, from the top-level directory on down, which may contain both source files and derived files that can be used to build targets in the local source tree. The canonical example would be an official source tree maintained by an integrator. If the repository contains derived files, then the derived files should have been built using **scons**, so that the repository contains the necessary signature information to allow **scons** to figure out when it is appropriate to use the repository copy of a derived file, instead of building one locally.

Note that if an up-to-date derived file already exists in a repository, **scons** will *not* make a copy in the local directory tree. In order to guarantee that a local copy will be made, use the `Local` method.

Requires(*target, prerequisite*)
***env*.Requires(*target, prerequisite*)**

Specifies an order-only relationship between the specified target file(s) and the specified prerequisite file(s). The prerequisite file(s) will be (re)built, if necessary, *before* the target file(s), but the target file(s) do not actually depend on the prerequisites and will not be rebuilt simply because the prerequisite file(s) change.

Example:

```
env.Requires('foo', 'file-that-must-be-built-before-foo')
```

Return(*[vars..., stop=True]*)

Return to the calling SConscript, optionally returning the values of variables named in `vars`. Multiple strings contaning variable names may be passed to `Return`. A string containing white space is split into individual variable names. Returns the value if one variable is specified, else returns a tuple of values. Returns an empty tuple if *`vars`* is omitted.

By default `Return` stops processing the current SConscript and returns immediately. The optional `stop` keyword argument may be set to a false value to continue processing the rest of the SConscript file after the `Return` call (this was the default behavior prior to SCons 0.98.) However, the values returned are still the values of the variables in the named `vars` at the point `Return` was called.

Examples:

```
# Returns no values (evaluates False)
Return()

# Returns the value of the 'foo' Python variable.
Return("foo")

# Returns the values of the Python variables 'foo' and 'bar'.
Return("foo", "bar")

# Returns the values of Python variables 'val1' and 'val2'.
Return('val1 val2')
```

Scanner(*function, [name, argument, skeys, path_function, node_class, node_factory, scan_check, recursive]*)
***env*.Scanner(*function, [name, argument, skeys, path_function, node_class, node_factory, scan_check, recursive]*)**

Creates a Scanner object for the specified *`function`*. See manpage section "Scanner Objects" for a complete explanation of the arguments and behavior.

SConscript(*scripts, [exports, variant_dir, duplicate, must_exist]*)
***env*.SConscript(*scripts, [exports, variant_dir, duplicate, must_exist]*)**
SConscript(*dirs=subdirs, [name=script, exports, variant_dir, duplicate, must_exist]*)
***env*.SConscript(*dirs=subdirs, [name=script, exports, variant_dir, duplicate, must_exist]*)**

Execute one or more subsidiary SConscript (configuration) files. There are two ways to call the `SConscript` function.

The first calling style is to explicitly specify one or more `scripts` as the first argument. A single script may be specified as a string; multiple scripts must be specified as a list (either explicitly or as created by a function like `Split`). Examples:

```
SConscript('SConscript')      # run SConscript in the current directory
SConscript('src/SConscript')  # run SConscript in the src directory
SConscript(['src/SConscript', 'doc/SConscript'])
config = SConscript('MyConfig.py')
```

The second way to call `SConscript` is to specify a list of (sub)directory names as a `dirs=`*`subdirs`* keyword argument. In this case, **scons** will execute a subsidiary configuration file named `SConscript` in each of the specified directories. You may specify a name other than `SConscript` by supplying an optional `name=`*`script`* keyword argument. The first three examples below have the same effect as the first three examples above:

```
SConscript(dirs='.')      # run SConscript in the current directory
SConscript(dirs='src')    # run SConscript in the src directory
SConscript(dirs=['src', 'doc'])
SConscript(dirs=['sub1', 'sub2'], name='MySConscript')
```

The optional `exports` argument provides a string or list of strings representing variable names, or a dictionary of named values, to export. These variables are locally exported only to the called SConscript file(s) and do not affect the global pool of variables managed by the `Export` function. The subsidiary SConscript files must use the `Import` function to import the variables. Examples:

```
foo = SConscript('sub/SConscript', exports='env')
SConscript('dir/SConscript', exports=['env', 'variable'])
SConscript(dirs='subdir', exports='env variable')
SConscript(dirs=['one', 'two', 'three'], exports='shared_info')
```

If the optional `variant_dir` argument is present, it causes an effect equivalent to the `VariantDir` function. The `variant_dir` argument is interpreted relative to the directory of the calling SConscript file. The optional `duplicate` argument is interpreted as for `VariantDir`. If `variant_dir` is omitted, the `duplicate` argument is ignored. See the description of `VariantDir` below for additional details and restrictions.

If `variant_dir` is present, the source directory is the directory in which the SConscript file resides and the SConscript file is evaluated as if it were in the `variant_dir` directory:

```
SConscript('src/SConscript', variant_dir='build')
```

is equivalent to

```
VariantDir('build', 'src')
SConscript('build/SConscript')
```

This later paradigm is often used when the sources are in the same directory as the `SConstruct`:

```
SConscript('SConscript', variant_dir='build')
```

is equivalent to

```
VariantDir('build', '.')
SConscript('build/SConscript')
```

If the optional `must_exist` is `True`, causes an exception to be raised if a requested SConscript file is not found. The current default is `False`, causing only a warning to be emitted, but this default is deprecated (*since 3.1*). For scripts which truly intend to be optional, transition to explicitly supplying `must_exist=False` to the `SConscript` call.

Here are some composite examples:

```
# collect the configuration information and use it to build src and doc
shared_info = SConscript('MyConfig.py')
SConscript('src/SConscript', exports='shared_info')
SConscript('doc/SConscript', exports='shared_info')
```

```
# build debugging and production versions.  SConscript
# can use Dir('.').path to determine variant.
SConscript('SConscript', variant_dir='debug', duplicate=0)
SConscript('SConscript', variant_dir='prod', duplicate=0)
```

```
# build debugging and production versions.  SConscript
# is passed flags to use.
opts = { 'CPPDEFINES' : ['DEBUG'], 'CCFLAGS' : '-pgdb' }
SConscript('SConscript', variant_dir='debug', duplicate=0, exports=opts)
opts = { 'CPPDEFINES' : ['NODEBUG'], 'CCFLAGS' : '-O' }
SConscript('SConscript', variant_dir='prod', duplicate=0, exports=opts)
```

```
# build common documentation and compile for different architectures
SConscript('doc/SConscript', variant_dir='build/doc', duplicate=0)
SConscript('src/SConscript', variant_dir='build/x86', duplicate=0)
SConscript('src/SConscript', variant_dir='build/ppc', duplicate=0)
```

`SConscript` returns the values of any variables named by the executed SConscript(s) in arguments to the `Return` function (see above for details). If a single `SConscript` call causes multiple scripts to be executed, the return value is a tuple containing the returns of all of the scripts. If an executed script does not explicitly call `Return`, it returns `None`.

SConscriptChdir(*value*)
***env*.SConscriptChdir(*value*)**

By default, **scons** changes its working directory to the directory in which each subsidiary SConscript file lives. This behavior may be disabled by specifying either:

```
SConscriptChdir(0)
env.SConscriptChdir(0)
```

in which case **scons** will stay in the top-level directory while reading all SConscript files. (This may be necessary when building from repositories, when all the directories in which SConscript files may be found don't necessarily exist locally.) You may enable and disable this ability by calling SConscriptChdir() multiple times.

Example:

```
env = Environment()
SConscriptChdir(0)
SConscript('foo/SConscript') # will not chdir to foo
env.SConscriptChdir(1)
SConscript('bar/SConscript') # will chdir to bar
```

`SConsignFile(`*`[name, dbm_module]`*`)`
***`env`*`.SConsignFile(`*`[name, dbm_module]`*`)`**

Specify where to store the SCons file signature database, and which database format to use. This may be useful to specify alternate database files and/or file locations for different types of builds.

The optional *`name`* argument is the base name of the database file(s). If not an absolute path name, these are placed relative to the directory containing the top-level `SConstruct` file. The default is `.sconsign`. The actual database file(s) stored on disk may have an appropriate suffix appended by the chosen *`dbm_module`*

The optional *`dbm_module`* argument specifies which Python database module to use for reading/writing the file. The module must be imported first; then the imported module name is passed as the argument. The default is a custom `SCons.dblite` module that uses pickled Python data structures, which works on all Python versions. See documentation of the Python `dbm` module for other available types.

If called with no arguments, the database will default to `.sconsign.dblite` in the top directory of the project, which is also the default if if `SConsignFile` is not called.

The setting is global, so the only difference between the global function and the environment method form is variable expansion on *`name`*. There should only be one active call to this function/method in a given build setup.

If *`name`* is set to `None`, **scons** will store file signatures in a separate `.sconsign` file in each directory, not in a single combined database file. This is a backwards-compatibility meaure to support what was the default behavior prior to SCons 0.97 (i.e. before 2008). Use of this mode is discouraged and may be deprecated in a future SCons release.

Examples:

```
# Explicitly stores signatures in ".sconsign.dblite"
# in the top-level SConstruct directory (the default behavior).
SConsignFile()

# Stores signatures in the file "etc/scons-signatures"
# relative to the top-level SConstruct directory.
# SCons will add a database suffix to this name.
SConsignFile("etc/scons-signatures")

# Stores signatures in the specified absolute file name.
# SCons will add a database suffix to this name.
SConsignFile("/home/me/SCons/signatures")

# Stores signatures in a separate .sconsign file
# in each directory.
SConsignFile(None)

# Stores signatures in a GNU dbm format .sconsign file
import dbm.gnu
SConsignFile(dbm_module=dbm.gnu)
```

***env*.SetDefault(*key=val, [...]*)**

Sets construction variables to default values specified with the keyword arguments if (and only if) the variables are not already set. The following statements are equivalent:

```
env.SetDefault(FOO='foo')
if 'FOO' not in env:
    env['FOO'] = 'foo'
```

SetOption(*name, value*)
***env*.SetOption(*name, value*)**

Sets **scons** option variable *name* to *value*. These options are all also settable via command-line options but the variable name may differ from the command-line option name - see the table for correspondences. A value set via command-line option will take precedence over one set with `SetOption`, which allows setting a project default in the scripts and temporarily overriding it via command line. `SetOption` calls can also be placed in the `site_init.py` file.

See the documentation in the manpage for the corresponding command line option for information about each specific option. The *value* parameter is mandatory, for option values which are boolean in nature (that is, the command line option does not take an argument) use a *value* which evaluates to true (e.g. `True`, `1`) or false (e.g. `False`, `0`).

Options which affect the reading and processing of SConscript files are not settable using `SetOption` since those files must be read in order to find the `SetOption` call in the first place.

The settable variables with their associated command-line options are:

Settable name	Command-line options	Notes
`clean`	`-c`, `--clean`, `--remove`	
`diskcheck`	`--diskcheck`	
`duplicate`	`--duplicate`	
`experimental`	`--experimental`	*since 4.2*
`hash_chunksize`	`--hash-chunksize`	Actually sets `md5_chunksize`. *since 4.2*
`hash_format`	`--hash-format`	*since 4.2*
`help`	`-h`, `--help`	
`implicit_cache`	`--implicit-cache`	
`implicit_deps_changed`	`--implicit-deps-changed`	Also sets `implicit_cache`. *(settable since 4.2)*
`implicit_deps_unchanged`	`--implicit-deps-unchanged`	Also sets `implicit_cache`. *(settable since 4.2)*
`max_drift`	`--max-drift`	
`md5_chunksize`	`--md5-chunksize`	
`no_exec`	`-n`, `--no-exec`, `--just-print`, `--dry-run`, `--recon`	
`no_progress`	`-Q`	See [a]
`num_jobs`	`-j`, `--jobs`	
`random`	`--random`	

Settable name	Command-line options	Notes
`silent`	`-s`, `--silent`, `--quiet`	
`stack_size`	`--stack-size`	
`warn`	`--warn`	

[a]If `no_progress` is set via `SetOption` in an SConscript file (but not if set in a `site_init.py` file) there will still be an initial status message about reading SConscript files since SCons has to start reading them before it can see the `SetOption`.

Example:

```
SetOption('max_drift', 0)
```

`SideEffect(`*`side_effect, target`*`)`
***`env`*`.SideEffect(`*`side_effect, target`*`)`**

Declares *`side_effect`* as a side effect of building *`target`*. Both *`side_effect`* and *`target`* can be a list, a file name, or a node. A side effect is a target file that is created or updated as a side effect of building other targets. For example, a Windows PDB file is created as a side effect of building the .obj files for a static library, and various log files are created updated as side effects of various TeX commands. If a target is a side effect of multiple build commands, **scons** will ensure that only one set of commands is executed at a time. Consequently, you only need to use this method for side-effect targets that are built as a result of multiple build commands.

Because multiple build commands may update the same side effect file, by default the *`side_effect`* target is *not* automatically removed when the *`target`* is removed by the `-c` option. (Note, however, that the *`side_effect`* might be removed as part of cleaning the directory in which it lives.) If you want to make sure the *`side_effect`* is cleaned whenever a specific *`target`* is cleaned, you must specify this explicitly with the `Clean` or `env.Clean` function.

This function returns the list of side effect Node objects that were successfully added. If the list of side effects contained any side effects that had already been added, they are not added and included in the returned list.

`Split(`*`arg`*`)`
***`env`*`.Split(`*`arg`*`)`**

If *`arg`* is a string, splits on whitespace and returns a list of strings without whitespace. This mode is the most common case, and can be used to split a list of filenames (for example) rather than having to type them as a list of individually quoted words. If *`arg`* is a list or tuple returns the list or tuple unchanged. If *`arg`* is any other type of object, returns a list containing just the object. These non-string cases do not actually do any spliting, but allow an argument variable to be passed to `Split` without having to first check its type.

Example:

```
files = Split("f1.c f2.c f3.c")
files = env.Split("f4.c f5.c f6.c")
files = Split("""
    f7.c
    f8.c
    f9.c
""")
```

***`env`*`.subst(`*`input, [raw, target, source, conv]`*`)`**

Performs construction variable interpolation on *`input`*, which can be a string or a sequence.

By default, leading or trailing white space will be removed from the result, and all sequences of white space will be compressed to a single space character. Additionally, any `$(` and `$)` character sequences will be stripped from the returned string, The optional *`raw`* argument may be set to `1` if you want to preserve white space and `$(-$)`

sequences. The *raw* argument may be set to `2` if you want to additionally discard all characters between any `$(` and `$)` pairs (as is done for signature calculation).

If the input is a sequence (list or tuple), the individual elements of the sequence will be expanded, and the results will be returned as a list.

The optional *target* and *source* keyword arguments must be set to lists of target and source nodes, respectively, if you want the `$TARGET`, `$TARGETS`, `$SOURCE` and `$SOURCES` to be available for expansion. This is usually necessary if you are calling `env.subst` from within a Python function used as an SCons action.

Returned string values or sequence elements are converted to their string representation by default. The optional *conv* argument may specify a conversion function that will be used in place of the default. For example, if you want Python objects (including SCons Nodes) to be returned as Python objects, you can use a Python lambda expression to pass in an unnamed function that simply returns its unconverted argument.

Example:

```
print(env.subst("The C compiler is: $CC"))

def compile(target, source, env):
    sourceDir = env.subst(
        "${SOURCE.srcdir}",
        target=target,
        source=source
    )

source_nodes = env.subst('$EXPAND_TO_NODELIST', conv=lambda x: x)
```

Tag(*node, tags*)

Annotates file or directory Nodes with information about how the `Package` Builder should package those files or directories. All Node-level tags are optional.

Examples:

```
# makes sure the built library will be installed with 644 file access mode
Tag(Library('lib.c'), UNIX_ATTR="0o644")

# marks file2.txt to be a documentation file
Tag('file2.txt', DOC)
```

Tool(*name, [toolpath, **kwargs]*)
***env*.Tool(*name, [toolpath, **kwargs]*)**

Locates the tool specification module *name* and returns a callable tool object for that tool. The tool module is searched for in standard locations and in any paths specified by the optional *toolpath* parameter. The standard locations are SCons' own internal path for tools plus the toolpath, if any (see the **Tools** section in the manual page for more details). Any additional keyword arguments *kwargs* are passed to the tool module's `generate` function during tool object construction.

When called, the tool object updates a construction environment with construction variables and arranges any other initialization needed to use the mechanisms that tool describes.

When the `env.Tool` form is used, the tool object is automatically called to update `env` and the value of *tool* is appended to the `$TOOLS` construction variable in that environment.

Examples:

```
env.Tool('gcc')
env.Tool('opengl', toolpath=['build/tools'])
```

When the global function `Tool` form is used, the tool object is constructed but not called, as it lacks the context of an environment to update. The tool object can be passed to an `Environment` or `Clone` call as part of the *`tools`* keyword argument, in which case the tool is applied to the environment being constructed, or it can be called directly, in which case a construction environment to update must be passed as the argument. Either approach will also update the `$TOOLS` construction variable.

Examples:

```
env = Environment(tools=[Tool('msvc')])

env = Environment()
msvctool = Tool('msvc')
msvctool(env)  # adds 'msvc' to the TOOLS variable
gltool = Tool('opengl', toolpath = ['tools'])
gltool(env)  # adds 'opengl' to the TOOLS variable
```

Changed in SCons 4.2: `env.Tool` now returns the tool object, previously it did not return (i.e. returned `None`).

`Value`(*`value, [built_value], [name]`*)
***`env`*.`Value`(*`value, [built_value], [name]`*)**

Returns a Node object representing the specified Python value. Value Nodes can be used as dependencies of targets. If the result of calling `str`(*`value`*) changes between SCons runs, any targets depending on `Value`(*`value`*) will be rebuilt. (This is true even when using timestamps to decide if files are up-to-date.) When using timestamp source signatures, Value Nodes' timestamps are equal to the system time when the Node is created. *`name`* can be provided as an alternative name for the resulting `Value` node; this is advised if the *`value`* parameter can't be converted to a string.

The returned Value Node object has a `write`() method that can be used to "build" a Value Node by setting a new value. The optional *`built_value`* argument can be specified when the Value Node is created to indicate the Node should already be considered "built." There is a corresponding `read`() method that will return the built value of the Node.

Examples:

```
env = Environment()

def create(target, source, env):
    # A function that will write a 'prefix=$SOURCE'
    # string into the file name specified as the
    # $TARGET.
    with open(str(target[0]), 'wb') as f:
        f.write('prefix=' + source[0].get_contents())

# Fetch the prefix= argument, if any, from the command
# line, and use /usr/local as the default.
prefix = ARGUMENTS.get('prefix', '/usr/local')

# Attach a .Config() builder for the above function action
# to the construction environment.
```

```
env['BUILDERS']['Config'] = Builder(action = create)
env.Config(target = 'package-config', source = Value(prefix))

def build_value(target, source, env):
    # A function that "builds" a Python Value by updating
    # the the Python value with the contents of the file
    # specified as the source of the Builder call ($SOURCE).
    target[0].write(source[0].get_contents())

output = env.Value('before')
input = env.Value('after')

# Attach a .UpdateValue() builder for the above function
# action to the construction environment.
env['BUILDERS']['UpdateValue'] = Builder(action = build_value)
env.UpdateValue(target = Value(output), source = Value(input))
```

VariantDir(*variant_dir, src_dir, [duplicate]*)
***env*.VariantDir(*variant_dir, src_dir, [duplicate]*)**

Sets up an alternate build location. When building in the *variant_dir*, SCons backfills as needed with files from *src_dir* to create a complete build directory. `VariantDir` can be called multiple times with the same *src_dir* to set up multiple builds with different options (*variants*).

The *variant* location must be in or underneath the project top directory, and *src_dir* may not be underneath *variant_dir*.

By default, SCons physically duplicates the source files and SConscript files as needed into the variant tree. Thus, a build performed in the variant tree is guaranteed to be identical to a build performed in the source tree even if intermediate source files are generated during the build, or if preprocessors or other scanners search for included files relative to the source file, or if individual compilers or other invoked tools are hard-coded to put derived files in the same directory as source files. Only the files SCons calculates are needed for the build are duplicated into *variant_dir*.

If possible on the platform, the duplication is performed by linking rather than copying. This behavior is affected by the `--duplicate` command-line option.

Duplicating the source files may be disabled by setting the *duplicate* argument to `False`. This will cause SCons to invoke Builders using the path names of source files in *src_dir* and the path names of derived files within *variant_dir*. This is more efficient than `duplicate=True`, and is safe for most builds; revert to `True` if it causes problems.

`VariantDir` works most naturally with used with a subsidiary SConscript file. The subsidiary SConscript file is called as if it were in *variant_dir*, regardless of the value of *duplicate*. This is how you tell **scons** which variant of a source tree to build:

```

# run src/SConscript in two variant directories
VariantDir('build/variant1', 'src')
SConscript('build/variant1/SConscript')
VariantDir('build/variant2', 'src')
SConscript('build/variant2/SConscript')
```

See also the `SConscript` function, described above, for another way to specify a variant directory in conjunction with calling a subsidiary SConscript file.

Examples:

```
# use names in the build directory, not the source directory
VariantDir('build', 'src', duplicate=0)
Program('build/prog', 'build/source.c')

# this builds both the source and docs in a separate subtree
VariantDir('build', '.', duplicate=0)
SConscript(dirs=['build/src','build/doc'])

# same as previous example, but only uses SConscript
SConscript(dirs='src', variant_dir='build/src', duplicate=0)
SConscript(dirs='doc', variant_dir='build/doc', duplicate=0)
```

`WhereIs(`*`program, [path, pathext, reject]`*`)`
***`env`*`.WhereIs(`*`program, [path, pathext, reject]`*`)`**

Searches for the specified executable *`program`*, returning the full path to the program or `None`.

When called as a construction environment method, searches the paths in the *`path`* keyword argument, or if `None` (the default) the paths listed in the construction environment (`env['ENV']['PATH']`). The external environment's path list (`os.environ['PATH']`) is used as a fallback if the key `env['ENV']['PATH']` does not exist.

On Windows systems, searches for executable programs with any of the file extensions listed in the *`pathext`* keyword argument, or if `None` (the default) the pathname extensions listed in the construction environment (`env['ENV']['PATHEXT']`). The external environment's pathname extensions list (`os.environ['PATHEXT']`) is used as a fallback if the key `env['ENV']['PATHEXT']` does not exist.

When called as a global function, uses the external environment's path `os.environ['PATH']` and path extensions `os.environ['PATHEXT']`, respectively, if *`path`* and *`pathext`* are `None`.

Will not select any path name or names in the optional *`reject`* list.

SConscript Variables

In addition to the global functions and methods, **scons** supports a number of variables that can be used in SConscript files to affect how you want the build to be performed.

`ARGLIST`

A list of the *keyword=value* arguments specified on the command line. Each element in the list is a tuple containing the argument. The separate *keyword* and *value* elements of the tuple can be accessed by subscripting for elements **[0]** and **[1]** of the tuple, or, more readably, by using tuple unpacking. Example:

```
print("first keyword, value =", ARGLIST[0][0], ARGLIST[0][1])
print("second keyword, value =", ARGLIST[1][0], ARGLIST[1][1])
key, value = ARGLIST[2]
print("third keyword, value =", key, value)
for key, value in ARGLIST:
    # process key and value
```

`ARGUMENTS`

A dictionary of all the *keyword=value* arguments specified on the command line. The dictionary is not in order, and if a given keyword has more than one value assigned to it on the command line, the last (right-most) value is the one in the `ARGUMENTS` dictionary.

Example:

```
if ARGUMENTS.get('debug', 0):
    env = Environment(CCFLAGS='-g')
else:
    env = Environment()
```

BUILD_TARGETS

A list of the targets which **scons** has been asked to build. The contents will be either those targets listed on the command line, or, if none, those targets set via calls to the `Default` function. It does *not* contain any dependent targets that **scons** selects for building as a result of making the sure the specified targets are up to date, if those targets did not appear on the command line. The list is empty if neither command line targets or `Default` calls are present.

The elements of this list may be strings *or* nodes, so you should run the list through the Python `str` function to make sure any Node path names are converted to strings.

Because this list may be taken from the list of targets specified using the `Default` function, the contents of the list may change on each successive call to `Default`. See the `DEFAULT_TARGETS` list, below, for additional information.

Example:

```
if 'foo' in BUILD_TARGETS:
    print("Don't forget to test the `foo' program!")
if 'special/program' in BUILD_TARGETS:
    SConscript('special')
```

COMMAND_LINE_TARGETS

A list of the targets explicitly specified on the command line. If there are command line targets, this list will have the same contents as `BUILD_TARGETS`. If there are no targets specified on the command line, the list is empty. The elements of this list are strings. This can be used, for example, to take specific actions only when certain targets are explicitly being built.

Example:

```
if 'foo' in COMMAND_LINE_TARGETS:
    print("Don't forget to test the `foo' program!")
if 'special/program' in COMMAND_LINE_TARGETS:
    SConscript('special')
```

DEFAULT_TARGETS

A list of the target *nodes* that have been specified using the `Default` function. If there are no command line targets, this list will have the same contents as `BUILD_TARGETS`. Since the elements of the list are nodes, you need to call the Python `str` function on them to get the path name for each Node.

Example:

```
print(str(DEFAULT_TARGETS[0]))
if 'foo' in [str(t) for t in DEFAULT_TARGETS]:
    print("Don't forget to test the `foo' program!")
```

The contents of the `DEFAULT_TARGETS` list change on on each successive call to the `Default` function:

```
print([str(t) for t in DEFAULT_TARGETS])   # originally []
Default('foo')
print([str(t) for t in DEFAULT_TARGETS])   # now a node ['foo']
Default('bar')
print([str(t) for t in DEFAULT_TARGETS])   # now a node ['foo', 'bar']
Default(None)
print([str(t) for t in DEFAULT_TARGETS])   # back to []
```

Consequently, be sure to use `DEFAULT_TARGETS` only after you've made all of your `Default`() calls, or else simply be careful of the order of these statements in your SConscript files so that you don't look for a specific default target before it's actually been added to the list.

These variables may be accessed from custom Python modules that you import into an SConscript file by adding the following to the Python module:

```
from SCons.Script import *
```

Construction Variables

A construction environment has an associated dictionary of *construction variables* that are used by built-in or user-supplied build rules. Construction variable naming must follow the same rules as Python identifier naming: the initial character must be an underscore or letter, followed by any number of underscores, letters, or digits. A construction environment is not a Python dictionary itself, but it can be indexed like one to access a construction variable:

```
env["CC"] = "cc"
flags = env.get("CPPDEFINES", [])
```

Construction variables can also be retrieved and set by using the `Dictionary` method of the construction environment to create an actual dictionary:

```
cvars = env.Dictionary()
cvars["CC"] = "cc"
```

Construction variables can also be passed to the construction environment constructor:

```
env = Environment(CC="cc")
```

or when copying a construction environment using the `Clone` method:

```
env2 = env.Clone(CC="cl.exe")
```

Construction variables can also be supplied as keyword arguments to a builder, in which case those settings affect only the work done by that builder call, and not the construction environment as a whole. This concept is called an *override*:

```
env.Program('hello', 'hello.c', LIBS=['gl', 'glut'])
```

A number of useful construction variables are automatically defined by scons for each supported platform, and you can modify these or define any additional construction variables for your own use, taking care not to overwrite ones which SCons is using. The following is a list of the possible automatically defined construction variables.

Note the actual list available at execution time will never include all of these, as the ones detected as not being useful (wrong platform, necessary external command or files not installed, etc.) will not be set up. Correct build setups should be resilient to the possible absence of certain construction variables before using them, for example by using a Python dictionary `get` method to retrieve the value and taking alternative action if the return indicates the variable is unset. The `env.Dump` method can be called to examine the construction variables set in a particular environment.

`__LDMODULEVERSIONFLAGS`

This construction variable automatically introduces `$_LDMODULEVERSIONFLAGS` if `$LDMODULEVERSION` is set. Othervise it evaluates to an empty string.

`__NINJA_NO`

Internal flag. Used to tell SCons whether or not to try to import pypi's ninja python package. This is set to True when being called by Ninja?

`__SHLIBVERSIONFLAGS`

This construction variable automatically introduces `$_SHLIBVERSIONFLAGS` if `$SHLIBVERSION` is set. Othervise it evaluates to an empty string.

`APPLELINK_COMPATIBILITY_VERSION`

On Mac OS X this is used to set the linker flag: -compatibility_version

The value is specified as X[.Y[.Z]] where X is between 1 and 65535, Y can be omitted or between 1 and 255, Z can be omitted or between 1 and 255. This value will be derived from `$SHLIBVERSION` if not specified. The lowest digit will be dropped and replaced by a 0.

If the `$APPLELINK_NO_COMPATIBILITY_VERSION` is set then no -compatibility_version will be output.

See MacOS's ld manpage for more details

`_APPLELINK_COMPATIBILITY_VERSION`

A macro (by default a generator function) used to create the linker flags to specify apple's linker's -compatibility_version flag. The default generator uses `$APPLELINK_COMPATIBILITY_VERSION` and `$APPLELINK_NO_COMPATIBILITY_VERSION` and `$SHLIBVERSION` to determine the correct flag.

`APPLELINK_CURRENT_VERSION`

On Mac OS X this is used to set the linker flag: -current_version

The value is specified as X[.Y[.Z]] where X is between 1 and 65535, Y can be omitted or between 1 and 255, Z can be omitted or between 1 and 255. This value will be set to `$SHLIBVERSION` if not specified.

If the `$APPLELINK_NO_CURRENT_VERSION` is set then no -current_version will be output.

See MacOS's ld manpage for more details

`_APPLELINK_CURRENT_VERSION`

A macro (by default a generator function) used to create the linker flags to specify apple's linker's -current_version flag. The default generator uses `$APPLELINK_CURRENT_VERSION` and `$APPLELINK_NO_CURRENT_VERSION` and `$SHLIBVERSION` to determine the correct flag.

`APPLELINK_NO_COMPATIBILITY_VERSION`

Set this to any True (1|True|non-empty string) value to disable adding -compatibility_version flag when generating versioned shared libraries.

This overrides `$APPLELINK_COMPATIBILITY_VERSION`.

`APPLELINK_NO_CURRENT_VERSION`

Set this to any True (1|True|non-empty string) value to disable adding -current_version flag when generating versioned shared libraries.

This overrides `$APPLELINK_CURRENT_VERSION`.

AR

The static library archiver.

ARCHITECTURE

Specifies the system architecture for which the package is being built. The default is the system architecture of the machine on which SCons is running. This is used to fill in the `Architecture:` field in an Ipkg `control` file, and the `BuildArch:` field in the RPM `.spec` file, as well as forming part of the name of a generated RPM package file.

See the `Package` builder.

ARCOM

The command line used to generate a static library from object files.

ARCOMSTR

The string displayed when a static library is generated from object files. If this is not set, then `$ARCOM` (the command line) is displayed.

```
env = Environment(ARCOMSTR = "Archiving $TARGET")
```

ARFLAGS

General options passed to the static library archiver.

AS

The assembler.

ASCOM

The command line used to generate an object file from an assembly-language source file.

ASCOMSTR

The string displayed when an object file is generated from an assembly-language source file. If this is not set, then `$ASCOM` (the command line) is displayed.

```
env = Environment(ASCOMSTR = "Assembling $TARGET")
```

ASFLAGS

General options passed to the assembler.

ASPPCOM

The command line used to assemble an assembly-language source file into an object file after first running the file through the C preprocessor. Any options specified in the `$ASFLAGS` and `$CPPFLAGS` construction variables are included on this command line.

ASPPCOMSTR

The string displayed when an object file is generated from an assembly-language source file after first running the file through the C preprocessor. If this is not set, then `$ASPPCOM` (the command line) is displayed.

```
env = Environment(ASPPCOMSTR = "Assembling $TARGET")
```

ASPPFLAGS

General options when an assembling an assembly-language source file into an object file after first running the file through the C preprocessor. The default is to use the value of `$ASFLAGS`.

BIBTEX

The bibliography generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

BIBTEXCOM

The command line used to call the bibliography generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

BIBTEXCOMSTR

The string displayed when generating a bibliography for TeX or LaTeX. If this is not set, then `$BIBTEXCOM` (the command line) is displayed.

```
env = Environment(BIBTEXCOMSTR = "Generating bibliography $TARGET")
```

BIBTEXFLAGS

General options passed to the bibliography generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

BUILDERS

A dictionary mapping the names of the builders available through the construction environment to underlying Builder objects. Custom builders need to be added to this to make them available.

A platform-dependent default list of builders such as `Program`, `Library` etc. is used to populate this construction variable when the construction environment is initialized via the presence/absence of the tools those builders depend on. `$BUILDERS` can be examined to learn which builders will actually be available at run-time.

Note that if you initialize this construction variable through assignment when the construction environment is created, that value for `$BUILDERS` will override any defaults:

```
bld = Builder(action='foobuild < $SOURCE > $TARGET')
env = Environment(BUILDERS={'NewBuilder': bld})
```

To instead use a new Builder object in addition to the default Builders, add your new Builder object like this:

```
env = Environment()
env.Append(BUILDERS={'NewBuilder': bld})
```

or this:

```
env = Environment()
env['BUILDERS']['NewBuilder'] = bld
```

CACHEDIR_CLASS

The class type that SCons should use when instantiating a new `CacheDir` for the given environment. It must be a subclass of the SCons.CacheDir.CacheDir class.

CC

The C compiler.

CCCOM

The command line used to compile a C source file to a (static) object file. Any options specified in the `$CFLAGS`, `$CCFLAGS` and `$CPPFLAGS` construction variables are included on this command line. See also `$SHCCCOM` for compiling to shared objects.

CCCOMSTR

If set, the string displayed when a C source file is compiled to a (static) object file. If not set, then `$CCCOM` (the command line) is displayed. See also `$SHCCCOMSTR` for compiling to shared objects.

```
env = Environment(CCCOMSTR = "Compiling static object $TARGET")
```

CCFLAGS

General options that are passed to the C and C++ compilers. See also `$SHCCFLAGS` for compiling to shared objects.

CCPCHFLAGS

Options added to the compiler command line to support building with precompiled headers. The default value expands expands to the appropriate Microsoft Visual C++ command-line options when the `$PCH` construction variable is set.

CCPDBFLAGS

Options added to the compiler command line to support storing debugging information in a Microsoft Visual C++ PDB file. The default value expands expands to appropriate Microsoft Visual C++ command-line options when the `$PDB` construction variable is set.

The Visual C++ compiler option that SCons uses by default to generate PDB information is `/Z7`. This works correctly with parallel (`-j`) builds because it embeds the debug information in the intermediate object files, as opposed to sharing a single PDB file between multiple object files. This is also the only way to get debug information embedded into a static library. Using the `/Zi` instead may yield improved link-time performance, although parallel builds will no longer work.

You can generate PDB files with the `/Zi` switch by overriding the default `$CCPDBFLAGS` variable as follows:

```
env['CCPDBFLAGS'] = ['${(PDB and "/Zi /Fd%s" % File(PDB)) or ""}']
```

An alternative would be to use the `/Zi` to put the debugging information in a separate `.pdb` file for each object file by overriding the `$CCPDBFLAGS` variable as follows:

```
env['CCPDBFLAGS'] = '/Zi /Fd${TARGET}.pdb'
```

CCVERSION

The version number of the C compiler. This may or may not be set, depending on the specific C compiler being used.

CFILESUFFIX

The suffix for C source files. This is used by the internal CFile builder when generating C files from Lex (.l) or YACC (.y) input files. The default suffix, of course, is `.c` (lower case). On case-insensitive systems (like Windows), SCons also treats `.C` (upper case) files as C files.

CFLAGS

General options that are passed to the C compiler (C only; not C++). See also `$SHCFLAGS` for compiling to shared objects.

CHANGE_SPECFILE

A hook for modifying the file that controls the packaging build (the `.spec` for RPM, the `control` for Ipkg, the `.wxs` for MSI). If set, the function will be called after the SCons template for the file has been written.

See the `Package` builder.

CHANGED_SOURCES

A reserved variable name that may not be set or used in a construction environment. (See the manpage section "Variable Substitution" for more information).

CHANGED_TARGETS

A reserved variable name that may not be set or used in a construction environment. (See the manpage section "Variable Substitution" for more information).

CHANGELOG

The name of a file containing the change log text to be included in the package. This is included as the `%changelog` section of the RPM `.spec` file.

See the `Package` builder.

COMPILATIONDB_COMSTR

The string displayed when the `CompilationDatabase` builder's action is run.

COMPILATIONDB_PATH_FILTER

A string which instructs `CompilationDatabase` to only include entries where the `output` member matches the pattern in the filter string using fnmatch, which uses glob style wildcards.

The default value is an empty string '', which disables filtering.

COMPILATIONDB_USE_ABSPATH

A boolean flag to instruct `CompilationDatabase` whether to write the `file` and `output` members in the compilation database using absolute or relative paths.

The default value is False (use relative paths)

_concat

A function used to produce variables like `$_CPPINCFLAGS`. It takes four mandatory arguments, and up to 4 additional optional arguments: 1) a prefix to concatenate onto each element, 2) a list of elements, 3) a suffix to concatenate onto each element, 4) an environment for variable interpolation, 5) an optional function that will be called to transform the list before concatenation, 6) an optionally specified target (Can use TARGET), 7) an optionally specified source (Can use SOURCE), 8) optional *`affect_signature`* flag which will wrap non-empty returned value with $(and $) to indicate the contents should not affect the signature of the generated command line.

```
env['_CPPINCFLAGS'] = '${_concat(INCPREFIX, CPPPATH, INCSUFFIX, __env__, RDirs,
```

CONFIGUREDIR

The name of the directory in which Configure context test files are written. The default is `.sconf_temp` in the top-level directory containing the `SConstruct` file.

CONFIGURELOG

The name of the `Configure` context log file. The default is `config.log` in the top-level directory containing the `SConstruct` file.

_CPPDEFFLAGS

An automatically-generated construction variable containing the C preprocessor command-line options to define values. The value of `$_CPPDEFFLAGS` is created by respectively prepending and appending `$CPPDEFPREFIX` and `$CPPDEFSUFFIX` to each definition in `$CPPDEFINES`.

CPPDEFINES

A platform independent specification of C preprocessor macro definitions. The definitions will be added to command lines through the automatically-generated `$_CPPDEFFLAGS` construction variable (see above), which is constructed according to the type of value of `$CPPDEFINES`:

If $CPPDEFINES is a string, the values of the $CPPDEFPREFIX and $CPPDEFSUFFIX construction variables will be respectively prepended and appended to each definition in $CPPDEFINES.

```
# Will add -Dxyz to POSIX compiler command lines,
# and /Dxyz to Microsoft Visual C++ command lines.
env = Environment(CPPDEFINES='xyz')
```

If $CPPDEFINES is a list, the values of the $CPPDEFPREFIX and $CPPDEFSUFFIX construction variables will be respectively prepended and appended to each element in the list. If any element is a list or tuple, then the first item is the name being defined and the second item is its value:

```
# Will add -DB=2 -DA to POSIX compiler command lines,
# and /DB=2 /DA to Microsoft Visual C++ command lines.
env = Environment(CPPDEFINES=[('B', 2), 'A'])
```

If $CPPDEFINES is a dictionary, the values of the $CPPDEFPREFIX and $CPPDEFSUFFIX construction variables will be respectively prepended and appended to each item from the dictionary. The key of each dictionary item is a name being defined to the dictionary item's corresponding value; if the value is None, then the name is defined without an explicit value. Note that the resulting flags are sorted by keyword to ensure that the order of the options on the command line is consistent each time **scons** is run.

```
# Will add -DA -DB=2 to POSIX compiler command lines,
# and /DA /DB=2 to Microsoft Visual C++ command lines.
env = Environment(CPPDEFINES={'B':2, 'A':None})
```

CPPDEFPREFIX

The prefix used to specify preprocessor macro definitions on the C compiler command line. This will be prepended to each definition in the $CPPDEFINES construction variable when the $_CPPDEFFLAGS variable is automatically generated.

CPPDEFSUFFIX

The suffix used to specify preprocessor macro definitions on the C compiler command line. This will be appended to each definition in the $CPPDEFINES construction variable when the $_CPPDEFFLAGS variable is automatically generated.

CPPFLAGS

User-specified C preprocessor options. These will be included in any command that uses the C preprocessor, including not just compilation of C and C++ source files via the $CCCOM, $SHCCCOM, $CXXCOM and $SHCXXCOM command lines, but also the $FORTRANPPCOM, $SHFORTRANPPCOM, $F77PPCOM and $SHF77PPCOM command lines used to compile a Fortran source file, and the $ASPPCOM command line used to assemble an assembly language source file, after first running each file through the C preprocessor. Note that this variable does *not* contain -I (or similar) include search path options that scons generates automatically from $CPPPATH. See $_CPPINCFLAGS, below, for the variable that expands to those options.

_CPPINCFLAGS

An automatically-generated construction variable containing the C preprocessor command-line options for specifying directories to be searched for include files. The value of $_CPPINCFLAGS is created by respectively prepending and appending $INCPREFIX and $INCSUFFIX to each directory in $CPPPATH.

CPPPATH

The list of directories that the C preprocessor will search for include directories. The C/C++ implicit dependency scanner will search these directories for include files. In general it's not advised to put include directory directives directly into `$CCFLAGS` or `$CXXFLAGS` as the result will be non-portable and the directories will not be searched by the dependency scanner. `$CPPPATH` should be a list of path strings, or a single string, not a pathname list joined by Python's `os.sep`.

Note: directory names in `$CPPPATH` will be looked-up relative to the directory of the SConscript file when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use the # prefix:

```
env = Environment(CPPPATH='#/include')
```

The directory look-up can also be forced using the `Dir` function:

```
include = Dir('include')
env = Environment(CPPPATH=include)
```

The directory list will be added to command lines through the automatically-generated `$_CPPINCFLAGS` construction variable, which is constructed by respectively prepending and appending the values of the `$INCPREFIX` and `$INCSUFFIX` construction variables to each directory in `$CPPPATH`. Any command lines you define that need the `$CPPPATH` directory list should include `$_CPPINCFLAGS`:

```
env = Environment(CCCOM="my_compiler $_CPPINCFLAGS -c -o $TARGET $SOURCE")
```

CPPSUFFIXES

The list of suffixes of files that will be scanned for C preprocessor implicit dependencies (#include lines). The default list is:

```
[".c", ".C", ".cxx", ".cpp", ".c++", ".cc",
 ".h", ".H", ".hxx", ".hpp", ".hh",
 ".F", ".fpp", ".FPP",
 ".m", ".mm",
 ".S", ".spp", ".SPP"]
```

CXX

The C++ compiler. See also `$SHCXX` for compiling to shared objects..

CXXCOM

The command line used to compile a C++ source file to an object file. Any options specified in the `$CXXFLAGS` and `$CPPFLAGS` construction variables are included on this command line. See also `$SHCXXCOM` for compiling to shared objects..

CXXCOMSTR

If set, the string displayed when a C++ source file is compiled to a (static) object file. If not set, then `$CXXCOM` (the command line) is displayed. See also `$SHCXXCOMSTR` for compiling to shared objects..

```
env = Environment(CXXCOMSTR = "Compiling static object $TARGET")
```

CXXFILESUFFIX

The suffix for C++ source files. This is used by the internal CXXFile builder when generating C++ files from Lex (.ll) or YACC (.yy) input files. The default suffix is `.cc`. SCons also treats files with the suffixes `.cpp`, `.cxx`, `.c++`, and `.C++` as C++ files, and files with `.mm` suffixes as Objective C++ files. On case-sensitive systems (Linux, UNIX, and other POSIX-alikes), SCons also treats `.C` (upper case) files as C++ files.

CXXFLAGS

General options that are passed to the C++ compiler. By default, this includes the value of `$CCFLAGS`, so that setting `$CCFLAGS` affects both C and C++ compilation. If you want to add C++-specific flags, you must set or override the value of `$CXXFLAGS`. See also `$SHCXXFLAGS` for compiling to shared objects..

CXXVERSION

The version number of the C++ compiler. This may or may not be set, depending on the specific C++ compiler being used.

DC

The D compiler to use. See also `$SHDC` for compiling to shared objects.

DCOM

The command line used to compile a D file to an object file. Any options specified in the `$DFLAGS` construction variable is included on this command line. See also `$SHDCOM` for compiling to shared objects.

DCOMSTR

If set, the string displayed when a D source file is compiled to a (static) object file. If not set, then `$DCOM` (the command line) is displayed. See also `$SHDCOMSTR` for compiling to shared objects.

DDEBUG

List of debug tags to enable when compiling.

DDEBUGPREFIX

DDEBUGPREFIX.

DDEBUGSUFFIX

DDEBUGSUFFIX.

DESCRIPTION

A long description of the project being packaged. This is included in the relevant section of the file that controls the packaging build.

See the `Package` builder.

DESCRIPTION_lang

A language-specific long description for the specified `lang`. This is used to populate a `%description -l` section of an RPM `.spec` file.

See the `Package` builder.

DFILESUFFIX

DFILESUFFIX.

DFLAGPREFIX

DFLAGPREFIX.

DFLAGS

General options that are passed to the D compiler.

DFLAGSUFFIX

DFLAGSUFFIX.

DINCPREFIX

DINCPREFIX.

DINCSUFFIX

DLIBFLAGSUFFIX.

Dir
A function that converts a string into a Dir instance relative to the target being built.

Dirs
A function that converts a list of strings into a list of Dir instances relative to the target being built.

DLIB
Name of the lib tool to use for D codes.

DLIBCOM
The command line to use when creating libraries.

DLIBDIRPREFIX
DLIBLINKPREFIX.

DLIBDIRSUFFIX
DLIBLINKSUFFIX.

DLIBFLAGPREFIX
DLIBFLAGPREFIX.

DLIBFLAGSUFFIX
DLIBFLAGSUFFIX.

DLIBLINKPREFIX
DLIBLINKPREFIX.

DLIBLINKSUFFIX
DLIBLINKSUFFIX.

DLINK
Name of the linker to use for linking systems including D sources. See also `$SHDLINK` for linking shared objects.

DLINKCOM
The command line to use when linking systems including D sources. See also `$SHDLINKCOM` for linking shared objects.

DLINKFLAGPREFIX
DLINKFLAGPREFIX.

DLINKFLAGS
List of linker flags. See also `$SHDLINKFLAGS` for linking shared objects.

DLINKFLAGSUFFIX
DLINKFLAGSUFFIX.

DOCBOOK_DEFAULT_XSL_EPUB
The default XSLT file for the `DocbookEpub` builder within the current environment, if no other XSLT gets specified via keyword.

DOCBOOK_DEFAULT_XSL_HTML
The default XSLT file for the `DocbookHtml` builder within the current environment, if no other XSLT gets specified via keyword.

DOCBOOK_DEFAULT_XSL_HTMLCHUNKED
The default XSLT file for the `DocbookHtmlChunked` builder within the current environment, if no other XSLT gets specified via keyword.

DOCBOOK_DEFAULT_XSL_HTMLHELP
The default XSLT file for the `DocbookHtmlhelp` builder within the current environment, if no other XSLT gets specified via keyword.

DOCBOOK_DEFAULT_XSL_MAN
The default XSLT file for the `DocbookMan` builder within the current environment, if no other XSLT gets specified via keyword.

DOCBOOK_DEFAULT_XSL_PDF
The default XSLT file for the `DocbookPdf` builder within the current environment, if no other XSLT gets specified via keyword.

DOCBOOK_DEFAULT_XSL_SLIDESHTML
The default XSLT file for the `DocbookSlidesHtml` builder within the current environment, if no other XSLT gets specified via keyword.

DOCBOOK_DEFAULT_XSL_SLIDESPDF
The default XSLT file for the `DocbookSlidesPdf` builder within the current environment, if no other XSLT gets specified via keyword.

DOCBOOK_FOP
The path to the PDF renderer `fop` or `xep`, if one of them is installed (`fop` gets checked first).

DOCBOOK_FOPCOM
The full command-line for the PDF renderer `fop` or `xep`.

DOCBOOK_FOPCOMSTR
The string displayed when a renderer like `fop` or `xep` is used to create PDF output from an XML file.

DOCBOOK_FOPFLAGS
Additonal command-line flags for the PDF renderer `fop` or `xep`.

DOCBOOK_XMLLINT
The path to the external executable `xmllint`, if it's installed. Note, that this is only used as last fallback for resolving XIncludes, if no lxml Python binding can be imported in the current system.

DOCBOOK_XMLLINTCOM
The full command-line for the external executable `xmllint`.

DOCBOOK_XMLLINTCOMSTR
The string displayed when `xmllint` is used to resolve XIncludes for a given XML file.

DOCBOOK_XMLLINTFLAGS
Additonal command-line flags for the external executable `xmllint`.

DOCBOOK_XSLTPROC
The path to the external executable `xsltproc` (or `saxon`, `xalan`), if one of them is installed. Note, that this is only used as last fallback for XSL transformations, if no lxml Python binding can be imported in the current system.

DOCBOOK_XSLTPROCCOM
The full command-line for the external executable `xsltproc` (or `saxon`, `xalan`).

DOCBOOK_XSLTPROCCOMSTR
The string displayed when `xsltproc` is used to transform an XML file via a given XSLT stylesheet.

DOCBOOK_XSLTPROCFLAGS
Additonal command-line flags for the external executable `xsltproc` (or `saxon`, `xalan`).

DOCBOOK_XSLTPROCPARAMS

Additonal parameters that are not intended for the XSLT processor executable, but the XSL processing itself. By default, they get appended at the end of the command line for `saxon` and `saxon-xslt`, respectively.

DPATH

List of paths to search for import modules.

DRPATHPREFIX

DRPATHPREFIX.

DRPATHSUFFIX

DRPATHSUFFIX.

DSUFFIXES

The list of suffixes of files that will be scanned for imported D package files. The default list is `['.d']`.

DVERPREFIX

DVERPREFIX.

DVERSIONS

List of version tags to enable when compiling.

DVERSUFFIX

DVERSUFFIX.

DVIPDF

The TeX DVI file to PDF file converter.

DVIPDFCOM

The command line used to convert TeX DVI files into a PDF file.

DVIPDFCOMSTR

The string displayed when a TeX DVI file is converted into a PDF file. If this is not set, then `$DVIPDFCOM` (the command line) is displayed.

DVIPDFFLAGS

General options passed to the TeX DVI file to PDF file converter.

DVIPS

The TeX DVI file to PostScript converter.

DVIPSFLAGS

General options passed to the TeX DVI file to PostScript converter.

ENV

A dictionary of environment variables to use when invoking commands. When `$ENV` is used in a command all list values will be joined using the path separator and any other non-string values will simply be coerced to a string. Note that, by default, **scons** does *not* propagate the environment in effect when you execute **scons** to the commands used to build target files. This is so that builds will be guaranteed repeatable regardless of the environment variables set at the time **scons** is invoked.

If you want to propagate your environment variables to the commands executed to build target files, you must do so explicitly:

```
import os
```

```
env = Environment(ENV=os.environ.copy())
```

Note that you can choose only to propagate certain environment variables. A common example is the system `PATH` environment variable, so that **scons** uses the same utilities as the invoking shell (or other process):

```
import os
env = Environment(ENV={'PATH': os.environ['PATH']})
```

`ESCAPE`

A function that will be called to escape shell special characters in command lines. The function should take one argument: the command line string to escape; and should return the escaped command line.

`F03`

The Fortran 03 compiler. You should normally set the `$FORTRAN` variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set `$F03` if you need to use a specific compiler or compiler version for Fortran 03 files.

`F03COM`

The command line used to compile a Fortran 03 source file to an object file. You only need to set `$F03COM` if you need to use a specific command line for Fortran 03 files. You should normally set the `$FORTRANCOM` variable, which specifies the default command line for all Fortran versions.

`F03COMSTR`

If set, the string displayed when a Fortran 03 source file is compiled to an object file. If not set, then `$F03COM` or `$FORTRANCOM` (the command line) is displayed.

`F03FILESUFFIXES`

The list of file extensions for which the F03 dialect will be used. By default, this is `['.f03']`

`F03FLAGS`

General user-specified options that are passed to the Fortran 03 compiler. Note that this variable does *not* contain `-I` (or similar) include search path options that scons generates automatically from `$F03PATH`. See `$_F03INCFLAGS` below, for the variable that expands to those options. You only need to set `$F03FLAGS` if you need to define specific user options for Fortran 03 files. You should normally set the `$FORTRANFLAGS` variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

`_F03INCFLAGS`

An automatically-generated construction variable containing the Fortran 03 compiler command-line options for specifying directories to be searched for include files. The value of `$_F03INCFLAGS` is created by appending `$INCPREFIX` and `$INCSUFFIX` to the beginning and end of each directory in `$F03PATH`.

`F03PATH`

The list of directories that the Fortran 03 compiler will search for include directories. The implicit dependency scanner will search these directories for include files. Don't explicitly put include directory arguments in `$F03FLAGS` because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in `$F03PATH` will be looked-up relative to the SConscript directory when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use #: You only need to set `$F03PATH` if you need to define a specific include path for Fortran 03 files. You should normally set the `$FORTRANPATH` variable, which specifies the include path for the default Fortran compiler for all Fortran versions.

```
env = Environment(F03PATH='#/include')
```

The directory look-up can also be forced using the `Dir`() function:

```
include = Dir('include')
env = Environment(F03PATH=include)
```

The directory list will be added to command lines through the automatically-generated `$_F03INCFLAGS` construction variable, which is constructed by appending the values of the `$INCPREFIX` and `$INCSUFFIX` construction variables to the beginning and end of each directory in `$F03PATH`. Any command lines you define that need the F03PATH directory list should include `$_F03INCFLAGS`:

```
env = Environment(F03COM="my_compiler $_F03INCFLAGS -c -o $TARGET $SOURCE")
```

F03PPCOM
The command line used to compile a Fortran 03 source file to an object file after first running the file through the C preprocessor. Any options specified in the `$F03FLAGS` and `$CPPFLAGS` construction variables are included on this command line. You only need to set `$F03PPCOM` if you need to use a specific C-preprocessor command line for Fortran 03 files. You should normally set the `$FORTRANPPCOM` variable, which specifies the default C-preprocessor command line for all Fortran versions.

F03PPCOMSTR
If set, the string displayed when a Fortran 03 source file is compiled to an object file after first running the file through the C preprocessor. If not set, then `$F03PPCOM` or `$FORTRANPPCOM` (the command line) is displayed.

F03PPFILESUFFIXES
The list of file extensions for which the compilation + preprocessor pass for F03 dialect will be used. By default, this is empty.

F08
The Fortran 08 compiler. You should normally set the `$FORTRAN` variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set `$F08` if you need to use a specific compiler or compiler version for Fortran 08 files.

F08COM
The command line used to compile a Fortran 08 source file to an object file. You only need to set `$F08COM` if you need to use a specific command line for Fortran 08 files. You should normally set the `$FORTRANCOM` variable, which specifies the default command line for all Fortran versions.

F08COMSTR
If set, the string displayed when a Fortran 08 source file is compiled to an object file. If not set, then `$F08COM` or `$FORTRANCOM` (the command line) is displayed.

F08FILESUFFIXES
The list of file extensions for which the F08 dialect will be used. By default, this is `['.f08']`

F08FLAGS
General user-specified options that are passed to the Fortran 08 compiler. Note that this variable does *not* contain `-I` (or similar) include search path options that scons generates automatically from `$F08PATH`. See `$_F08INCFLAGS` below, for the variable that expands to those options. You only need to set `$F08FLAGS` if you need to define specific user options for Fortran 08 files. You should normally set the `$FORTRANFLAGS` variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

_F08INCFLAGS
An automatically-generated construction variable containing the Fortran 08 compiler command-line options for specifying directories to be searched for include files. The value of `$_F08INCFLAGS` is created by appending `$INCPREFIX` and `$INCSUFFIX` to the beginning and end of each directory in `$F08PATH`.

F08PATH

The list of directories that the Fortran 08 compiler will search for include directories. The implicit dependency scanner will search these directories for include files. Don't explicitly put include directory arguments in `$F08FLAGS` because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in `$F08PATH` will be looked-up relative to the SConscript directory when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use #: You only need to set `$F08PATH` if you need to define a specific include path for Fortran 08 files. You should normally set the `$FORTRANPATH` variable, which specifies the include path for the default Fortran compiler for all Fortran versions.

```
env = Environment(F08PATH='#/include')
```

The directory look-up can also be forced using the `Dir`() function:

```
include = Dir('include')
env = Environment(F08PATH=include)
```

The directory list will be added to command lines through the automatically-generated `$_F08INCFLAGS` construction variable, which is constructed by appending the values of the `$INCPREFIX` and `$INCSUFFIX` construction variables to the beginning and end of each directory in `$F08PATH`. Any command lines you define that need the F08PATH directory list should include `$_F08INCFLAGS`:

```
env = Environment(F08COM="my_compiler $_F08INCFLAGS -c -o $TARGET $SOURCE")
```

F08PPCOM

The command line used to compile a Fortran 08 source file to an object file after first running the file through the C preprocessor. Any options specified in the `$F08FLAGS` and `$CPPFLAGS` construction variables are included on this command line. You only need to set `$F08PPCOM` if you need to use a specific C-preprocessor command line for Fortran 08 files. You should normally set the `$FORTRANPPCOM` variable, which specifies the default C-preprocessor command line for all Fortran versions.

F08PPCOMSTR

If set, the string displayed when a Fortran 08 source file is compiled to an object file after first running the file through the C preprocessor. If not set, then `$F08PPCOM` or `$FORTRANPPCOM` (the command line) is displayed.

F08PPFILESUFFIXES

The list of file extensions for which the compilation + preprocessor pass for F08 dialect will be used. By default, this is empty.

F77

The Fortran 77 compiler. You should normally set the `$FORTRAN` variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set `$F77` if you need to use a specific compiler or compiler version for Fortran 77 files.

F77COM

The command line used to compile a Fortran 77 source file to an object file. You only need to set `$F77COM` if you need to use a specific command line for Fortran 77 files. You should normally set the `$FORTRANCOM` variable, which specifies the default command line for all Fortran versions.

F77COMSTR

If set, the string displayed when a Fortran 77 source file is compiled to an object file. If not set, then `$F77COM` or `$FORTRANCOM` (the command line) is displayed.

F77FILESUFFIXES

The list of file extensions for which the F77 dialect will be used. By default, this is `['.f77']`

F77FLAGS

General user-specified options that are passed to the Fortran 77 compiler. Note that this variable does *not* contain `-I` (or similar) include search path options that scons generates automatically from `$F77PATH`. See `$_F77INCFLAGS` below, for the variable that expands to those options. You only need to set `$F77FLAGS` if you need to define specific user options for Fortran 77 files. You should normally set the `$FORTRANFLAGS` variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

_F77INCFLAGS

An automatically-generated construction variable containing the Fortran 77 compiler command-line options for specifying directories to be searched for include files. The value of `$_F77INCFLAGS` is created by appending `$INCPREFIX` and `$INCSUFFIX` to the beginning and end of each directory in `$F77PATH`.

F77PATH

The list of directories that the Fortran 77 compiler will search for include directories. The implicit dependency scanner will search these directories for include files. Don't explicitly put include directory arguments in `$F77FLAGS` because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in `$F77PATH` will be looked-up relative to the SConscript directory when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use #: You only need to set `$F77PATH` if you need to define a specific include path for Fortran 77 files. You should normally set the `$FORTRANPATH` variable, which specifies the include path for the default Fortran compiler for all Fortran versions.

```
env = Environment(F77PATH='#/include')
```

The directory look-up can also be forced using the `Dir`() function:

```
include = Dir('include')
env = Environment(F77PATH=include)
```

The directory list will be added to command lines through the automatically-generated `$_F77INCFLAGS` construction variable, which is constructed by appending the values of the `$INCPREFIX` and `$INCSUFFIX` construction variables to the beginning and end of each directory in `$F77PATH`. Any command lines you define that need the F77PATH directory list should include `$_F77INCFLAGS`:

```
env = Environment(F77COM="my_compiler $_F77INCFLAGS -c -o $TARGET $SOURCE")
```

F77PPCOM

The command line used to compile a Fortran 77 source file to an object file after first running the file through the C preprocessor. Any options specified in the `$F77FLAGS` and `$CPPFLAGS` construction variables are included on this command line. You only need to set `$F77PPCOM` if you need to use a specific C-preprocessor command line for Fortran 77 files. You should normally set the `$FORTRANPPCOM` variable, which specifies the default C-preprocessor command line for all Fortran versions.

F77PPCOMSTR

If set, the string displayed when a Fortran 77 source file is compiled to an object file after first running the file through the C preprocessor. If not set, then `$F77PPCOM` or `$FORTRANPPCOM` (the command line) is displayed.

F77PPFILESUFFIXES

The list of file extensions for which the compilation + preprocessor pass for F77 dialect will be used. By default, this is empty.

F90

The Fortran 90 compiler. You should normally set the `$FORTRAN` variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set `$F90` if you need to use a specific compiler or compiler version for Fortran 90 files.

F90COM

The command line used to compile a Fortran 90 source file to an object file. You only need to set `$F90COM` if you need to use a specific command line for Fortran 90 files. You should normally set the `$FORTRANCOM` variable, which specifies the default command line for all Fortran versions.

F90COMSTR

If set, the string displayed when a Fortran 90 source file is compiled to an object file. If not set, then `$F90COM` or `$FORTRANCOM` (the command line) is displayed.

F90FILESUFFIXES

The list of file extensions for which the F90 dialect will be used. By default, this is `['.f90']`

F90FLAGS

General user-specified options that are passed to the Fortran 90 compiler. Note that this variable does *not* contain `-I` (or similar) include search path options that scons generates automatically from `$F90PATH`. See `$_F90INCFLAGS` below, for the variable that expands to those options. You only need to set `$F90FLAGS` if you need to define specific user options for Fortran 90 files. You should normally set the `$FORTRANFLAGS` variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

_F90INCFLAGS

An automatically-generated construction variable containing the Fortran 90 compiler command-line options for specifying directories to be searched for include files. The value of `$_F90INCFLAGS` is created by appending `$INCPREFIX` and `$INCSUFFIX` to the beginning and end of each directory in `$F90PATH`.

F90PATH

The list of directories that the Fortran 90 compiler will search for include directories. The implicit dependency scanner will search these directories for include files. Don't explicitly put include directory arguments in `$F90FLAGS` because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in `$F90PATH` will be looked-up relative to the SConscript directory when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use #: You only need to set `$F90PATH` if you need to define a specific include path for Fortran 90 files. You should normally set the `$FORTRANPATH` variable, which specifies the include path for the default Fortran compiler for all Fortran versions.

```
env = Environment(F90PATH='#/include')
```

The directory look-up can also be forced using the `Dir`() function:

```
include = Dir('include')
env = Environment(F90PATH=include)
```

The directory list will be added to command lines through the automatically-generated `$_F90INCFLAGS` construction variable, which is constructed by appending the values of the `$INCPREFIX` and `$INCSUFFIX` construction variables to the beginning and end of each directory in `$F90PATH`. Any command lines you define that need the F90PATH directory list should include `$_F90INCFLAGS`:

```
env = Environment(F90COM="my_compiler $_F90INCFLAGS -c -o $TARGET $SOURCE")
```

F90PPCOM

The command line used to compile a Fortran 90 source file to an object file after first running the file through the C preprocessor. Any options specified in the `$F90FLAGS` and `$CPPFLAGS` construction variables are included on this command line. You only need to set `$F90PPCOM` if you need to use a specific C-preprocessor command line for Fortran 90 files. You should normally set the `$FORTRANPPCOM` variable, which specifies the default C-preprocessor command line for all Fortran versions.

F90PPCOMSTR

If set, the string displayed when a Fortran 90 source file is compiled after first running the file through the C preprocessor. If not set, then `$F90PPCOM` or `$FORTRANPPCOM` (the command line) is displayed.

F90PPFILESUFFIXES

The list of file extensions for which the compilation + preprocessor pass for F90 dialect will be used. By default, this is empty.

F95

The Fortran 95 compiler. You should normally set the `$FORTRAN` variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set `$F95` if you need to use a specific compiler or compiler version for Fortran 95 files.

F95COM

The command line used to compile a Fortran 95 source file to an object file. You only need to set `$F95COM` if you need to use a specific command line for Fortran 95 files. You should normally set the `$FORTRANCOM` variable, which specifies the default command line for all Fortran versions.

F95COMSTR

If set, the string displayed when a Fortran 95 source file is compiled to an object file. If not set, then `$F95COM` or `$FORTRANCOM` (the command line) is displayed.

F95FILESUFFIXES

The list of file extensions for which the F95 dialect will be used. By default, this is `['.f95']`

F95FLAGS

General user-specified options that are passed to the Fortran 95 compiler. Note that this variable does *not* contain `-I` (or similar) include search path options that scons generates automatically from `$F95PATH`. See `$_F95INCFLAGS` below, for the variable that expands to those options. You only need to set `$F95FLAGS` if you need to define specific user options for Fortran 95 files. You should normally set the `$FORTRANFLAGS` variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

_F95INCFLAGS

An automatically-generated construction variable containing the Fortran 95 compiler command-line options for specifying directories to be searched for include files. The value of `$_F95INCFLAGS` is created by appending `$INCPREFIX` and `$INCSUFFIX` to the beginning and end of each directory in `$F95PATH`.

F95PATH

The list of directories that the Fortran 95 compiler will search for include directories. The implicit dependency scanner will search these directories for include files. Don't explicitly put include directory arguments in `$F95FLAGS` because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in `$F95PATH` will be looked-up relative to the SConscript directory when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use #: You only need to set `$F95PATH` if you need to define a specific include path for Fortran 95 files. You should normally set the `$FORTRANPATH` variable, which specifies the include path for the default Fortran compiler for all Fortran versions.

```
env = Environment(F95PATH='#/include')
```

The directory look-up can also be forced using the `Dir`() function:

```
include = Dir('include')
env = Environment(F95PATH=include)
```

The directory list will be added to command lines through the automatically-generated `$_F95INCFLAGS` construction variable, which is constructed by appending the values of the `$INCPREFIX` and `$INCSUFFIX` construction variables to the beginning and end of each directory in `$F95PATH`. Any command lines you define that need the F95PATH directory list should include `$_F95INCFLAGS`:

```
env = Environment(F95COM="my_compiler $_F95INCFLAGS -c -o $TARGET $SOURCE")
```

F95PPCOM
The command line used to compile a Fortran 95 source file to an object file after first running the file through the C preprocessor. Any options specified in the `$F95FLAGS` and `$CPPFLAGS` construction variables are included on this command line. You only need to set `$F95PPCOM` if you need to use a specific C-preprocessor command line for Fortran 95 files. You should normally set the `$FORTRANPPCOM` variable, which specifies the default C-preprocessor command line for all Fortran versions.

F95PPCOMSTR
If set, the string displayed when a Fortran 95 source file is compiled to an object file after first running the file through the C preprocessor. If not set, then `$F95PPCOM` or `$FORTRANPPCOM` (the command line) is displayed.

F95PPFILESUFFIXES
The list of file extensions for which the compilation + preprocessor pass for F95 dialect will be used. By default, this is empty.

File
A function that converts a string into a File instance relative to the target being built.

FORTRAN
The default Fortran compiler for all versions of Fortran.

FORTRANCOM
The command line used to compile a Fortran source file to an object file. By default, any options specified in the `$FORTRANFLAGS`, `$CPPFLAGS`, `$_CPPDEFFLAGS`, `$_FORTRANMODFLAG`, and `$_FORTRANINCFLAGS` construction variables are included on this command line.

FORTRANCOMSTR
If set, the string displayed when a Fortran source file is compiled to an object file. If not set, then `$FORTRANCOM` (the command line) is displayed.

FORTRANFILESUFFIXES
The list of file extensions for which the FORTRAN dialect will be used. By default, this is `['.f', '.for', '.ftn']`

FORTRANFLAGS
General user-specified options that are passed to the Fortran compiler. Note that this variable does *not* contain `-I` (or similar) include or module search path options that scons generates automatically from `$FORTRANPATH`. See `$_FORTRANINCFLAGS` and `$_FORTRANMODFLAG`, below, for the variables that expand those options.

_FORTRANINCFLAGS
An automatically-generated construction variable containing the Fortran compiler command-line options for specifying directories to be searched for include files and module files. The value of `$_FORTRANINCFLAGS` is

created by respectively prepending and appending `$INCPREFIX` and `$INCSUFFIX` to the beginning and end of each directory in `$FORTRANPATH`.

FORTRANMODDIR
Directory location where the Fortran compiler should place any module files it generates. This variable is empty, by default. Some Fortran compilers will internally append this directory in the search path for module files, as well.

FORTRANMODDIRPREFIX
The prefix used to specify a module directory on the Fortran compiler command line. This will be prepended to the beginning of the directory in the `$FORTRANMODDIR` construction variables when the `$_FORTRANMODFLAG` variables is automatically generated.

FORTRANMODDIRSUFFIX
The suffix used to specify a module directory on the Fortran compiler command line. This will be appended to the end of the directory in the `$FORTRANMODDIR` construction variables when the `$_FORTRANMODFLAG` variables is automatically generated.

_FORTRANMODFLAG
An automatically-generated construction variable containing the Fortran compiler command-line option for specifying the directory location where the Fortran compiler should place any module files that happen to get generated during compilation. The value of `$_FORTRANMODFLAG` is created by respectively prepending and appending `$FORTRANMODDIRPREFIX` and `$FORTRANMODDIRSUFFIX` to the beginning and end of the directory in `$FORTRANMODDIR`.

FORTRANMODPREFIX
The module file prefix used by the Fortran compiler. SCons assumes that the Fortran compiler follows the quasi-standard naming convention for module files of `module_name.mod`. As a result, this variable is left empty, by default. For situations in which the compiler does not necessarily follow the normal convention, the user may use this variable. Its value will be appended to every module file name as scons attempts to resolve dependencies.

FORTRANMODSUFFIX
The module file suffix used by the Fortran compiler. SCons assumes that the Fortran compiler follows the quasi-standard naming convention for module files of `module_name.mod`. As a result, this variable is set to ".mod", by default. For situations in which the compiler does not necessarily follow the normal convention, the user may use this variable. Its value will be appended to every module file name as scons attempts to resolve dependencies.

FORTRANPATH
The list of directories that the Fortran compiler will search for include files and (for some compilers) module files. The Fortran implicit dependency scanner will search these directories for include files (but not module files since they are autogenerated and, as such, may not actually exist at the time the scan takes place). Don't explicitly put include directory arguments in FORTRANFLAGS because the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in FORTRANPATH will be looked-up relative to the SConscript directory when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use #:

```
env = Environment(FORTRANPATH='#/include')
```

The directory look-up can also be forced using the `Dir`() function:

```
include = Dir('include')
env = Environment(FORTRANPATH=include)
```

The directory list will be added to command lines through the automatically-generated `$_FORTRANINCFLAGS` construction variable, which is constructed by respectively prepending and appending the values of the

$INCPREFIX and $INCSUFFIX construction variables to the beginning and end of each directory in $FORTRANPATH. Any command lines you define that need the FORTRANPATH directory list should include $_FORTRANINCFLAGS:

```
env = Environment(FORTRANCOM="my_compiler $_FORTRANINCFLAGS -c -o $TARGET $SOURCE")
```

FORTRANPPCOM
The command line used to compile a Fortran source file to an object file after first running the file through the C preprocessor. By default, any options specified in the $FORTRANFLAGS, $CPPFLAGS, $_CPPDEFFLAGS, $_FORTRANMODFLAG, and $_FORTRANINCFLAGS construction variables are included on this command line.

FORTRANPPCOMSTR
If set, the string displayed when a Fortran source file is compiled to an object file after first running the file through the C preprocessor. If not set, then $FORTRANPPCOM (the command line) is displayed.

FORTRANPPFILESUFFIXES
The list of file extensions for which the compilation + preprocessor pass for FORTRAN dialect will be used. By default, this is `['.fpp', '.FPP']`

FORTRANSUFFIXES
The list of suffixes of files that will be scanned for Fortran implicit dependencies (INCLUDE lines and USE statements). The default list is:

```
[".f", ".F", ".for", ".FOR", ".ftn", ".FTN", ".fpp", ".FPP",
".f77", ".F77", ".f90", ".F90", ".f95", ".F95"]
```

FRAMEWORKPATH
On Mac OS X with gcc, a list containing the paths to search for frameworks. Used by the compiler to find framework-style includes like #include <Fmwk/Header.h>. Used by the linker to find user-specified frameworks when linking (see $FRAMEWORKS). For example:

```
env.AppendUnique(FRAMEWORKPATH='#myframeworkdir')
```

will add

```
... -Fmyframeworkdir
```

to the compiler and linker command lines.

_FRAMEWORKPATH
On Mac OS X with gcc, an automatically-generated construction variable containing the linker command-line options corresponding to $FRAMEWORKPATH.

FRAMEWORKPATHPREFIX
On Mac OS X with gcc, the prefix to be used for the FRAMEWORKPATH entries. (see $FRAMEWORKPATH). The default value is `-F`.

FRAMEWORKPREFIX
On Mac OS X with gcc, the prefix to be used for linking in frameworks (see $FRAMEWORKS). The default value is `-framework`.

FRAMEWORKS

On Mac OS X with gcc, a list of the framework names to be linked into a program or shared library or bundle. The default value is the empty list. For example:

```
env.AppendUnique(FRAMEWORKS=Split('System Cocoa SystemConfiguration'))
```

_FRAMEWORKS

On Mac OS X with gcc, an automatically-generated construction variable containing the linker command-line options for linking with FRAMEWORKS.

FRAMEWORKSFLAGS

On Mac OS X with gcc, general user-supplied frameworks options to be added at the end of a command line building a loadable module. (This has been largely superseded by the `$FRAMEWORKPATH`, `$FRAMEWORKPATHPREFIX`, `$FRAMEWORKPREFIX` and `$FRAMEWORKS` variables described above.)

GS

The Ghostscript program used to, for example, convert PostScript to PDF files.

GSCOM

The full Ghostscript command line used for the conversion process. Its default value is “`$GS $GSFLAGS -sOutputFile=$TARGET $SOURCES`”.

GSCOMSTR

The string displayed when Ghostscript is called for the conversion process. If this is not set (the default), then `$GSCOM` (the command line) is displayed.

GSFLAGS

General options passed to the Ghostscript program, when converting PostScript to PDF files for example. Its default value is “`-dNOPAUSE -dBATCH -sDEVICE=pdfwrite`”

HOST_ARCH

The name of the host hardware architecture used to create this construction environment. The platform code sets this when initializing (see `$PLATFORM` and the *`platform`* argument to `Environment`). Note the detected name of the architecture may not be identical to that returned by the Python `platform.machine` method.

On the `win32` platform, if the Microsoft Visual C++ compiler is available, `msvc` tool setup is done using `$HOST_ARCH` and `$TARGET_ARCH`. Changing the values at any later time will not cause the tool to be reinitialized. Valid host arch values are `x86` and `arm` for 32-bit hosts and `amd64` and `x86_64` for 64-bit hosts.

Should be considered immutable. `$HOST_ARCH` is not currently used by other platforms, but the option is reserved to do so in future

HOST_OS

The name of the host operating system for the platform used to create this construction environment. The platform code sets this when initializing (see `$PLATFORM` and the *`platform`* argument to `Environment`).

Should be considered immutable. `$HOST_OS` is not currently used by SCons, but the option is reserved to do so in future

IDLSUFFIXES

The list of suffixes of files that will be scanned for IDL implicit dependencies (#include or import lines). The default list is:

```
[".idl", ".IDL"]
```

IMPLIBNOVERSIONSYMLINKS

Used to override $SHLIBNOVERSIONSYMLINKS/$LDMODULENOVERSIONSYMLINKS when creating versioned import library for a shared library/loadable module. If not defined, then $SHLIBNOVERSIONSYMLINKS/$LDMODULENOVERSIONSYMLINKS is used to determine whether to disable symlink generation or not.

IMPLIBPREFIX

The prefix used for import library names. For example, cygwin uses import libraries (libfoo.dll.a) in pair with dynamic libraries (cygfoo.dll). The cyglink linker sets $IMPLIBPREFIX to 'lib' and $SHLIBPREFIX to 'cyg'.

IMPLIBSUFFIX

The suffix used for import library names. For example, cygwin uses import libraries (libfoo.dll.a) in pair with dynamic libraries (cygfoo.dll). The cyglink linker sets $IMPLIBSUFFIX to '.dll.a' and $SHLIBSUFFIX to '.dll'.

IMPLIBVERSION

Used to override $SHLIBVERSION/$LDMODULEVERSION when generating versioned import library for a shared library/loadable module. If undefined, the $SHLIBVERSION/$LDMODULEVERSION is used to determine the version of versioned import library.

IMPLICIT_COMMAND_DEPENDENCIES

Controls whether or not SCons will add implicit dependencies for the commands executed to build targets.

By default, SCons will add to each target an implicit dependency on the command represented by the first argument of any command line it executes (which is typically the command itself). By setting such a dependency, SCons can determine that a target should be rebuilt if the command changes, such as when a compiler is upgraded to a new version. The specific file for the dependency is found by searching the PATH variable in the ENV dictionary in the construction environment used to execute the command. The default is the same as setting the construction variable $IMPLICIT_COMMAND_DEPENDENCIES to a True-like value (“true”, “yes”, or “1” - but not a number greater than one, as that has a different meaning).

Action strings can be segmented by the use of an AND operator, &&. In a segemented string, each segment is a separate “command line”, these are run sequentially until one fails or the entire sequence has been executed. If an action string is segmented, then the selected behavior of $IMPLICIT_COMMAND_DEPENDENCIES is applied to each segment.

If $IMPLICIT_COMMAND_DEPENDENCIES is set to a False-like value (“none”, “false”, “no”, “0”, etc.), then the implicit dependency will not be added to the targets built with that construction environment.

If $IMPLICIT_COMMAND_DEPENDENCIES is set to “2” or higher, then that number of arguments in the command line will be scanned for relative or absolute paths. If any are present, they will be added as implicit dependencies to the targets built with that construction environment. The first argument in the command line will be searched for using the PATH variable in the ENV dictionary in the construction environment used to execute the command. The other arguments will only be found if they are absolute paths or valid paths relative to the working directory.

If $IMPLICIT_COMMAND_DEPENDENCIES is set to “all”, then all arguments in the command line will be scanned for relative or absolute paths. If any are present, they will be added as implicit dependencies to the targets built with that construction environment. The first argument in the command line will be searched for using the PATH variable in the ENV dictionary in the construction environment used to execute the command. The other arguments will only be found if they are absolute paths or valid paths relative to the working directory.

```
env = Environment(IMPLICIT_COMMAND_DEPENDENCIES=False)
```

INCPREFIX
The prefix used to specify an include directory on the C compiler command line. This will be prepended to each directory in the `$CPPPATH` and `$FORTRANPATH` construction variables when the `$_CPPINCFLAGS` and `$_FORTRANINCFLAGS` variables are automatically generated.

INCSUFFIX
The suffix used to specify an include directory on the C compiler command line. This will be appended to each directory in the `$CPPPATH` and `$FORTRANPATH` construction variables when the `$_CPPINCFLAGS` and `$_FORTRANINCFLAGS` variables are automatically generated.

INSTALL
A function to be called to install a file into a destination file name. The default function copies the file into the destination (and sets the destination file's mode and permission bits to match the source file's). The function takes the following arguments:

```
def install(dest, source, env):
```

`dest` is the path name of the destination file. `source` is the path name of the source file. `env` is the construction environment (a dictionary of construction values) in force for this file installation.

INSTALLSTR
The string displayed when a file is installed into a destination file name. The default is:

```
Install file: "$SOURCE" as "$TARGET"
```

INTEL_C_COMPILER_VERSION
Set by the `intelc` Tool to the major version number of the Intel C compiler selected for use.

JAR
The Java archive tool.

JARCHDIR
The directory to which the Java archive tool should change (using the `-C` option).

JARCOM
The command line used to call the Java archive tool.

JARCOMSTR
The string displayed when the Java archive tool is called If this is not set, then `$JARCOM` (the command line) is displayed.

```
env = Environment(JARCOMSTR="JARchiving $SOURCES into $TARGET")
```

JARFLAGS
General options passed to the Java archive tool. By default this is set to `cf` to create the necessary **jar** file.

JARSUFFIX
The suffix for Java archives: `.jar` by default.

JAVABOOTCLASSPATH
Specifies the list of directories that will be added to the javac command line via the `-bootclasspath` option. The individual directory names will be separated by the operating system's path separate character (`:` on UNIX/Linux/POSIX, `;` on Windows).

JAVAC

The Java compiler.

JAVACCOM

The command line used to compile a directory tree containing Java source files to corresponding Java class files. Any options specified in the `$JAVACFLAGS` construction variable are included on this command line.

JAVACCOMSTR

The string displayed when compiling a directory tree of Java source files to corresponding Java class files. If this is not set, then `$JAVACCOM` (the command line) is displayed.

```
env = Environment(JAVACCOMSTR="Compiling class files $TARGETS from $SOURCES")
```

JAVACFLAGS

General options that are passed to the Java compiler.

JAVACLASSDIR

The directory in which Java class files may be found. This is stripped from the beginning of any Java .class file names supplied to the `JavaH` builder.

JAVACLASSPATH

Specifies the list of directories that will be searched for Java `.class` file. The directories in this list will be added to the javac and javah command lines via the `-classpath` option. The individual directory names will be separated by the operating system's path separate character (`:` on UNIX/Linux/POSIX, `;` on Windows).

Note that this currently just adds the specified directory via the `-classpath` option. SCons does not currently search the `$JAVACLASSPATH` directories for dependency `.class` files.

JAVACLASSSUFFIX

The suffix for Java class files; `.class` by default.

JAVAH

The Java generator for C header and stub files.

JAVAHCOM

The command line used to generate C header and stub files from Java classes. Any options specified in the `$JAVAHFLAGS` construction variable are included on this command line.

JAVAHCOMSTR

The string displayed when C header and stub files are generated from Java classes. If this is not set, then `$JAVAHCOM` (the command line) is displayed.

```
env = Environment(JAVAHCOMSTR="Generating header/stub file(s) $TARGETS from $SOURCES")
```

JAVAHFLAGS

General options passed to the C header and stub file generator for Java classes.

JAVAINCLUDES

Include path for Java header files (such as jni.h)

JAVASOURCEPATH

Specifies the list of directories that will be searched for input `.java` file. The directories in this list will be added to the javac command line via the `-sourcepath` option. The individual directory names will be separated by the operating system's path separate character (`:` on UNIX/Linux/POSIX, `;` on Windows).

Note that this currently just adds the specified directory via the `-sourcepath` option. SCons does not currently search the `$JAVASOURCEPATH` directories for dependency `.java` files.

JAVASUFFIX
The suffix for Java files; `.java` by default.

JAVAVERSION
Specifies the Java version being used by the `Java` builder. Set this to specify the version of Java targeted by the javac compiler. This is sometimes necessary because Java 1.5 changed the file names that are created for nested anonymous inner classes, which can cause a mismatch with the files that SCons expects will be generated by the javac compiler. Setting `$JAVAVERSION` to a version greater than `1.4` makes SCons realize that a build with such a compiler is actually up to date. The default is `1.4`.

While this is *not* primarily intended for selecting one version of the Java compiler vs. another, it does have that effect on the Windows platform. A more precise approach is to set `$JAVAC` (and related construction variables for related utilities) to the path to the specific Java compiler you want, if that is not the default compiler. On non-Windows platforms, the `alternatives` system may provide a way to adjust the default Java compiler without having to specify explicit paths.

LATEX
The LaTeX structured formatter and typesetter.

LATEXCOM
The command line used to call the LaTeX structured formatter and typesetter.

LATEXCOMSTR
The string displayed when calling the LaTeX structured formatter and typesetter. If this is not set, then `$LATEXCOM` (the command line) is displayed.

```
env = Environment(LATEXCOMSTR = "Building $TARGET from LaTeX input $SOURCES")
```

LATEXFLAGS
General options passed to the LaTeX structured formatter and typesetter.

LATEXRETRIES
The maximum number of times that LaTeX will be re-run if the `.log` generated by the `$LATEXCOM` command indicates that there are undefined references. The default is to try to resolve undefined references by re-running LaTeX up to three times.

LATEXSUFFIXES
The list of suffixes of files that will be scanned for LaTeX implicit dependencies (`\include` or `\import` files). The default list is:

```
[".tex", ".ltx", ".latex"]
```

LDMODULE
The linker for building loadable modules. By default, this is the same as `$SHLINK`.

LDMODULECOM
The command line for building loadable modules. On Mac OS X, this uses the `$LDMODULE`, `$LDMODULEFLAGS` and `$FRAMEWORKSFLAGS` variables. On other systems, this is the same as `$SHLINK`.

LDMODULECOMSTR
If set, the string displayed when building loadable modules. If not set, then `$LDMODULECOM` (the command line) is displayed.

LDMODULEEMITTER

Contains the emitter specification for the `LoadableModule` builder. The manpage section "Builder Objects" contains general information on specifying emitters.

LDMODULEFLAGS

General user options passed to the linker for building loadable modules.

LDMODULENOVERSIONSYMLINKS

Instructs the `LoadableModule` builder to not automatically create symlinks for versioned modules. Defaults to `$SHLIBNOVERSIONSYMLINKS`

LDMODULEPREFIX

The prefix used for loadable module file names. On Mac OS X, this is null; on other systems, this is the same as `$SHLIBPREFIX`.

_LDMODULESONAME

A macro that automatically generates loadable module's SONAME based on $TARGET, $LDMODULEVERSION and $LDMODULESUFFIX. Used by `LoadableModule` builder when the linker tool supports SONAME (e.g. `gnulink`).

LDMODULESUFFIX

The suffix used for loadable module file names. On Mac OS X, this is null; on other systems, this is the same as $SHLIBSUFFIX.

LDMODULEVERSION

When this construction variable is defined, a versioned loadable module is created by `LoadableModule` builder. This activates the `$_LDMODULEVERSIONFLAGS` and thus modifies the `$LDMODULECOM` as required, adds the version number to the library name, and creates the symlinks that are needed. `$LDMODULEVERSION` versions should exist in the same format as `$SHLIBVERSION`.

_LDMODULEVERSIONFLAGS

This macro automatically introduces extra flags to `$LDMODULECOM` when building versioned `LoadableModule` (that is when `$LDMODULEVERSION` is set). `_LDMODULEVERSIONFLAGS` usually adds `$SHLIBVERSIONFLAGS` and some extra dynamically generated options (such as `-Wl,-soname=$_LDMODULESONAME`). It is unused by plain (unversioned) loadable modules.

LDMODULEVERSIONFLAGS

Extra flags added to `$LDMODULECOM` when building versioned `LoadableModule`. These flags are only used when `$LDMODULEVERSION` is set.

LEX

The lexical analyzer generator.

LEXCOM

The command line used to call the lexical analyzer generator to generate a source file.

LEXCOMSTR

The string displayed when generating a source file using the lexical analyzer generator. If this is not set, then `$LEXCOM` (the command line) is displayed.

```
env = Environment(LEXCOMSTR = "Lex'ing $TARGET from $SOURCES")
```

LEXFLAGS

General options passed to the lexical analyzer generator.

LEXUNISTD

Used only on windows environments to set a lex flag to prevent 'unistd.h' from being included. The default value is '--nounistd'.

_LIBDIRFLAGS

An automatically-generated construction variable containing the linker command-line options for specifying directories to be searched for library. The value of `$_LIBDIRFLAGS` is created by respectively prepending and appending `$LIBDIRPREFIX` and `$LIBDIRSUFFIX` to each directory in `$LIBPATH`.

LIBDIRPREFIX

The prefix used to specify a library directory on the linker command line. This will be prepended to each directory in the `$LIBPATH` construction variable when the `$_LIBDIRFLAGS` variable is automatically generated.

LIBDIRSUFFIX

The suffix used to specify a library directory on the linker command line. This will be appended to each directory in the `$LIBPATH` construction variable when the `$_LIBDIRFLAGS` variable is automatically generated.

LIBEMITTER

Contains the emitter specification for the `StaticLibrary` builder. The manpage section "Builder Objects" contains general information on specifying emitters.

_LIBFLAGS

An automatically-generated construction variable containing the linker command-line options for specifying libraries to be linked with the resulting target. The value of `$_LIBFLAGS` is created by respectively prepending and appending `$LIBLINKPREFIX` and `$LIBLINKSUFFIX` to each filename in `$LIBS`.

LIBLINKPREFIX

The prefix used to specify a library to link on the linker command line. This will be prepended to each library in the `$LIBS` construction variable when the `$_LIBFLAGS` variable is automatically generated.

LIBLINKSUFFIX

The suffix used to specify a library to link on the linker command line. This will be appended to each library in the `$LIBS` construction variable when the `$_LIBFLAGS` variable is automatically generated.

LIBPATH

The list of directories that will be searched for libraries specified by the `$LIBS` construction variable. `$LIBPATH` should be a list of path strings, or a single string, not a pathname list joined by Python's `os.sep`. Do not put library search directives directly into `$LINKFLAGS` or `$SHLINKFLAGS` as the result will be non-portable.

Note: directory names in `$LIBPATH` will be looked-up relative to the directory of the SConscript file when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use the # prefix:

```
env = Environment(LIBPATH='#/libs')
```

The directory look-up can also be forced using the `Dir` function:

```
libs = Dir('libs')
env = Environment(LIBPATH=libs)
```

The directory list will be added to command lines through the automatically-generated `$_LIBDIRFLAGS` construction variable, which is constructed by respectively prepending and appending the values of the `$LIBDIRPREFIX` and `$LIBDIRSUFFIX` construction variables to each directory in `$LIBPATH`. Any command lines you define that need the `$LIBPATH` directory list should include `$_LIBDIRFLAGS`:

```
env = Environment(LINKCOM="my_linker $_LIBDIRFLAGS $_LIBFLAGS -o $TARGET $SOURCE")
```

LIBPREFIX

The prefix used for (static) library file names. A default value is set for each platform (posix, win32, os2, etc.), but the value is overridden by individual tools (ar, mslib, sgiar, sunar, tlib, etc.) to reflect the names of the libraries they create.

LIBPREFIXES

A list of all legal prefixes for library file names. When searching for library dependencies, SCons will look for files with these prefixes, the base library name, and suffixes from the `$LIBSUFFIXES` list.

LIBS

A list of one or more libraries that will be added to the link line for linking with any executable program, shared library, or loadable module created by the construction environment or override.

String-valued library names should include only the library base names, without prefixes such as `lib` or suffixes such as `.so` or `.dll`. The library list will be added to command lines through the automatically-generated `$_LIBFLAGS` construction variable which is constructed by respectively prepending and appending the values of the `$LIBLINKPREFIX` and `$LIBLINKSUFFIX` construction variables to each library name in `$LIBS`. Library name strings should not include a path component, instead the compiler will be directed to look for libraries in the paths specified by `$LIBPATH`.

Any command lines you define that need the `$LIBS` library list should include `$_LIBFLAGS`:

```
env = Environment(LINKCOM="my_linker $_LIBDIRFLAGS $_LIBFLAGS -o $TARGET $SOURCE")
```

If you add a `File` object to the `$LIBS` list, the name of that file will be added to `$_LIBFLAGS`, and thus to the link line, as-is, without `$LIBLINKPREFIX` or `$LIBLINKSUFFIX`. For example:

```
env.Append(LIBS=File('/tmp/mylib.so'))
```

In all cases, scons will add dependencies from the executable program to all the libraries in this list.

LIBSUFFIX

The suffix used for (static) library file names. A default value is set for each platform (posix, win32, os2, etc.), but the value is overridden by individual tools (ar, mslib, sgiar, sunar, tlib, etc.) to reflect the names of the libraries they create.

LIBSUFFIXES

A list of all legal suffixes for library file names. When searching for library dependencies, SCons will look for files with prefixes from the `$LIBPREFIXES` list, the base library name, and these suffixes.

LICENSE

The abbreviated name, preferably the SPDX code, of the license under which this project is released (GPL-3.0, LGPL-2.1, BSD-2-Clause etc.). See http://www.opensource.org/licenses/alphabetical [http://www.opensource.org/licenses/alphabetical] for a list of license names and SPDX codes.

See the `Package` builder.

LINESEPARATOR

The separator used by the `Substfile` and `Textfile` builders. This value is used between sources when constructing the target. It defaults to the current system line separator.

LINGUAS_FILE

The `$LINGUAS_FILE` defines file(s) containing list of additional linguas to be processed by `POInit`, `POUpdate` or `MOFiles` builders. It also affects `Translate` builder. If the variable contains a string, it defines name of the list file. The `$LINGUAS_FILE` may be a list of file names as well. If `$LINGUAS_FILE` is set to `True` (or non-zero numeric value), the list will be read from default file named `LINGUAS`.

LINK

The linker. See also `$SHLINK` for linking shared objects.

On POSIX systems (those using the `link` tool), you should normally not change this value as it defaults to a "smart" linker tool which selects a compiler driver matching the type of source files in use. So for example, if you set `$CXX` to a specific compiler name, and are compiling C++ sources, the smartlink function will automatically select the same compiler for linking.

LINKCOM

The command line used to link object files into an executable. See also `$SHLINKCOM` for linking shared objects.

LINKCOMSTR

If set, the string displayed when object files are linked into an executable. If not set, then `$LINKCOM` (the command line) is displayed. See also `$SHLINKCOMSTR`. for linking shared objects.

```
env = Environment(LINKCOMSTR = "Linking $TARGET")
```

LINKFLAGS

General user options passed to the linker. Note that this variable should *not* contain `-l` (or similar) options for linking with the libraries listed in `$LIBS`, nor `-L` (or similar) library search path options that scons generates automatically from `$LIBPATH`. See `$_LIBFLAGS` above, for the variable that expands to library-link options, and `$_LIBDIRFLAGS` above, for the variable that expands to library search path options. See also `$SHLINKFLAGS`. for linking shared objects.

M4

The M4 macro preprocessor.

M4COM

The command line used to pass files through the M4 macro preprocessor.

M4COMSTR

The string displayed when a file is passed through the M4 macro preprocessor. If this is not set, then `$M4COM` (the command line) is displayed.

M4FLAGS

General options passed to the M4 macro preprocessor.

MAKEINDEX

The makeindex generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

MAKEINDEXCOM

The command line used to call the makeindex generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

MAKEINDEXCOMSTR

The string displayed when calling the makeindex generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter. If this is not set, then `$MAKEINDEXCOM` (the command line) is displayed.

MAKEINDEXFLAGS

General options passed to the makeindex generator for the TeX formatter and typesetter and the LaTeX structured formatter and typesetter.

MAXLINELENGTH

The maximum number of characters allowed on an external command line. On Win32 systems, link lines longer than this many characters are linked via a temporary file name.

MIDL
The Microsoft IDL compiler.

MIDLCOM
The command line used to pass files to the Microsoft IDL compiler.

MIDLCOMSTR
The string displayed when the Microsoft IDL compiler is called. If this is not set, then `$MIDLCOM` (the command line) is displayed.

MIDLFLAGS
General options passed to the Microsoft IDL compiler.

MOSUFFIX
Suffix used for `MO` files (default: `'.mo'`). See `msgfmt` tool and `MOFiles` builder.

MSGFMT
Absolute path to **msgfmt(1)** binary, found by `Detect()`. See `msgfmt` tool and `MOFiles` builder.

MSGFMTCOM
Complete command line to run **msgfmt(1)** program. See `msgfmt` tool and `MOFiles` builder.

MSGFMTCOMSTR
String to display when **msgfmt(1)** is invoked (default: `''`, which means ``print `$MSGFMTCOM`"). See `msgfmt` tool and `MOFiles` builder.

MSGFMTFLAGS
Additional flags to **msgfmt(1)**. See `msgfmt` tool and `MOFiles` builder.

MSGINIT
Path to **msginit(1)** program (found via `Detect()`). See `msginit` tool and `POInit` builder.

MSGINITCOM
Complete command line to run **msginit(1)** program. See `msginit` tool and `POInit` builder.

MSGINITCOMSTR
String to display when **msginit(1)** is invoked (default: `''`, which means ``print `$MSGINITCOM`"). See `msginit` tool and `POInit` builder.

MSGINITFLAGS
List of additional flags to **msginit(1)** (default: `[]`). See `msginit` tool and `POInit` builder.

_MSGINITLOCALE
Internal ``macro". Computes locale (language) name based on target filename (default: `'${TARGET.filebase}'`).

See `msginit` tool and `POInit` builder.

MSGMERGE
Absolute path to **msgmerge(1)** binary as found by `Detect()`. See `msgmerge` tool and `POUpdate` builder.

MSGMERGECOM
Complete command line to run **msgmerge(1)** command. See `msgmerge` tool and `POUpdate` builder.

MSGMERGECOMSTR
String to be displayed when **msgmerge(1)** is invoked (default: `''`, which means ``print `$MSGMERGECOM`"). See `msgmerge` tool and `POUpdate` builder.

MSGMERGEFLAGS

Additional flags to **msgmerge(1)** command. See `msgmerge` tool and `POUpdate` builder.

MSSDK_DIR

The directory containing the Microsoft SDK (either Platform SDK or Windows SDK) to be used for compilation.

MSSDK_VERSION

The version string of the Microsoft SDK (either Platform SDK or Windows SDK) to be used for compilation. Supported versions include `6.1`, `6.0A`, `6.0`, `2003R2` and `2003R1`.

MSVC_BATCH

When set to any true value, specifies that SCons should batch compilation of object files when calling the Microsoft Visual C/C++ compiler. All compilations of source files from the same source directory that generate target files in a same output directory and were configured in SCons using the same construction environment will be built in a single call to the compiler. Only source files that have changed since their object files were built will be passed to each compiler invocation (via the `$CHANGED_SOURCES` construction variable). Any compilations where the object (target) file base name (minus the `.obj`) does not match the source file base name will be compiled separately.

MSVC_USE_SCRIPT

Use a batch script to set up the Microsoft Visual C++ compiler.

If set to the name of a Visual Studio `.bat` file (e.g. `vcvars.bat`), SCons will run that batch file instead of the auto-detected one, and extract the relevant variables from the result (typically `%INCLUDE%`, `%LIB%`, and `%PATH%`) for supplying to the build. This can be useful to force the use of a compiler version that SCons does not detect.

Setting `$MSVC_USE_SCRIPT` to `None` bypasses the Visual Studio autodetection entirely; use this if you are running SCons in a Visual Studio **cmd** window and importing the shell's environment variables - that is, if you are sure everything is set correctly already and you don't want SCons to change anything.

`$MSVC_USE_SCRIPT` overrides `$MSVC_VERSION` and `$TARGET_ARCH`.

MSVC_UWP_APP

Build libraries for a Universal Windows Platform (UWP) Application.

If `$MSVC_UWP_APP` is set, the Visual C++ environment will be set up to point to the Windows Store compatible libraries and Visual C++ runtimes. In doing so, any libraries that are built will be able to be used in a UWP App and published to the Windows Store. This flag will only have an effect with Visual Studio 2015 or later. This variable must be passed as an argument to the Environment() constructor; setting it later has no effect.

Valid values are '1' or '0'

MSVC_VERSION

Sets the preferred version of Microsoft Visual C/C++ to use.

If `$MSVC_VERSION` is not set, SCons will (by default) select the latest version of Visual C/C++ installed on your system. If the specified version isn't installed, tool initialization will fail. This variable must be passed as an argument to the `Environment` constructor; setting it later has no effect.

Valid values for Windows are `14.3`, `14.2`, `14.1`, `14.1Exp`, `14.0`, `14.0Exp`, `12.0`, `12.0Exp`, `11.0`, `11.0Exp`, `10.0`, `10.0Exp`, `9.0`, `9.0Exp`, `8.0`, `8.0Exp`, `7.1`, `7.0`, and `6.0`. Versions ending in `Exp` refer to "Express" or "Express for Desktop" editions.

MSVS

When the Microsoft Visual Studio tools are initialized, they set up this dictionary with the following keys:

VERSION

the version of MSVS being used (can be set via `$MSVS_VERSION`)

VERSIONS
: the available versions of MSVS installed

VCINSTALLDIR
: installed directory of Visual C++

VSINSTALLDIR
: installed directory of Visual Studio

FRAMEWORKDIR
: installed directory of the .NET framework

FRAMEWORKVERSIONS
: list of installed versions of the .NET framework, sorted latest to oldest.

FRAMEWORKVERSION
: latest installed version of the .NET framework

FRAMEWORKSDKDIR
: installed location of the .NET SDK.

PLATFORMSDKDIR
: installed location of the Platform SDK.

PLATFORMSDK_MODULES
: dictionary of installed Platform SDK modules, where the dictionary keys are keywords for the various modules, and the values are 2-tuples where the first is the release date, and the second is the version number.

If a value is not set, it was not available in the registry.

`MSVS_ARCH`

Sets the architecture for which the generated project(s) should build.

The default value is `x86`. `amd64` is also supported by SCons for most Visual Studio versions. Since Visual Studio 2015 `arm` is supported, and since Visual Studio 2017 `arm64` is supported. Trying to set `$MSVS_ARCH` to an architecture that's not supported for a given Visual Studio version will generate an error.

`MSVS_PROJECT_GUID`

The string placed in a generated Microsoft Visual Studio project file as the value of the `ProjectGUID` attribute. There is no default value. If not defined, a new GUID is generated.

`MSVS_SCC_AUX_PATH`

The path name placed in a generated Microsoft Visual Studio project file as the value of the `SccAuxPath` attribute if the `MSVS_SCC_PROVIDER` construction variable is also set. There is no default value.

`MSVS_SCC_CONNECTION_ROOT`

The root path of projects in your SCC workspace, i.e the path under which all project and solution files will be generated. It is used as a reference path from which the relative paths of the generated Microsoft Visual Studio project and solution files are computed. The relative project file path is placed as the value of the `SccLocalPath` attribute of the project file and as the values of the `SccProjectFilePathRelativizedFromConnection[i]` (where [i] ranges from 0 to the number of projects in the solution) attributes of the `GlobalSection(SourceCodeControl)` section of the Microsoft Visual Studio solution file. Similarly the relative solution file path is placed as the values of the `SccLocalPath[i]` (where [i] ranges from 0 to the number of projects in the solution) attributes of the `GlobalSection(SourceCodeControl)` section of the Microsoft Visual Studio solution file. This is used only if the `MSVS_SCC_PROVIDER` construction variable is also set. The default value is the current working directory.

MSVS_SCC_PROJECT_NAME

The project name placed in a generated Microsoft Visual Studio project file as the value of the `SccProjectName` attribute if the `MSVS_SCC_PROVIDER` construction variable is also set. In this case the string is also placed in the `SccProjectName0` attribute of the `GlobalSection(SourceCodeControl)` section of the Microsoft Visual Studio solution file. There is no default value.

MSVS_SCC_PROVIDER

The string placed in a generated Microsoft Visual Studio project file as the value of the `SccProvider` attribute. The string is also placed in the `SccProvider0` attribute of the `GlobalSection(SourceCodeControl)` section of the Microsoft Visual Studio solution file. There is no default value.

MSVS_VERSION

Sets the preferred version of Microsoft Visual Studio to use.

If `$MSVS_VERSION` is not set, SCons will (by default) select the latest version of Visual Studio installed on your system. So, if you have version 6 and version 7 (MSVS .NET) installed, it will prefer version 7. You can override this by specifying the `MSVS_VERSION` variable in the Environment initialization, setting it to the appropriate version ('6.0' or '7.0', for example). If the specified version isn't installed, tool initialization will fail.

This is obsolete: use `$MSVC_VERSION` instead. If `$MSVS_VERSION` is set and `$MSVC_VERSION` is not, `$MSVC_VERSION` will be set automatically to `$MSVS_VERSION`. If both are set to different values, scons will raise an error.

MSVSBUILDCOM

The build command line placed in a generated Microsoft Visual Studio project file. The default is to have Visual Studio invoke SCons with any specified build targets.

MSVSCLEANCOM

The clean command line placed in a generated Microsoft Visual Studio project file. The default is to have Visual Studio invoke SCons with the -c option to remove any specified targets.

MSVSENCODING

The encoding string placed in a generated Microsoft Visual Studio project file. The default is encoding `Windows-1252`.

MSVSPROJECTCOM

The action used to generate Microsoft Visual Studio project files.

MSVSPROJECTSUFFIX

The suffix used for Microsoft Visual Studio project (DSP) files. The default value is `.vcproj` when using Visual Studio version 7.x (.NET) or later version, and `.dsp` when using earlier versions of Visual Studio.

MSVSREBUILDCOM

The rebuild command line placed in a generated Microsoft Visual Studio project file. The default is to have Visual Studio invoke SCons with any specified rebuild targets.

MSVSSCONS

The SCons used in generated Microsoft Visual Studio project files. The default is the version of SCons being used to generate the project file.

MSVSSCONSCOM

The default SCons command used in generated Microsoft Visual Studio project files.

MSVSSCONSCRIPT

The sconscript file (that is, `SConstruct` or `SConscript` file) that will be invoked by Visual Studio project files (through the `$MSVSSCONSCOM` variable). The default is the same sconscript file that contains the call to `MSVSProject` to build the project file.

MSVSSCONSFLAGS
The SCons flags used in generated Microsoft Visual Studio project files.

MSVSSOLUTIONCOM
The action used to generate Microsoft Visual Studio solution files.

MSVSSOLUTIONSUFFIX
The suffix used for Microsoft Visual Studio solution (DSW) files. The default value is `.sln` when using Visual Studio version 7.x (.NET), and `.dsw` when using earlier versions of Visual Studio.

MT
The program used on Windows systems to embed manifests into DLLs and EXEs. See also `$WINDOWS_EMBED_MANIFEST`.

MTEXECOM
The Windows command line used to embed manifests into executables. See also `$MTSHLIBCOM`.

MTFLAGS
Flags passed to the `$MT` manifest embedding program (Windows only).

MTSHLIBCOM
The Windows command line used to embed manifests into shared libraries (DLLs). See also `$MTEXECOM`.

MWCW_VERSION
The version number of the MetroWerks CodeWarrior C compiler to be used.

MWCW_VERSIONS
A list of installed versions of the MetroWerks CodeWarrior C compiler on this system.

NAME
Specfies the name of the project to package.

See the `Package` builder.

NINJA_ALIAS_NAME
Name of the Alias() which is will cause SCons to create the `ninja.build` file, and then (optionally) run ninja.

NINJA_COMPDB_EXPAND
Boolean value (True|False) to instruct ninja to expand the command line arguments normally put into response files. This prevents lines in the compilation database like “gcc @rsp_file” and instead yields “gcc -c -o myfile.o myfile.c -Ia -DXYZ”

Ninja's compdb tool added the “-x” flag in Ninja V1.9.0

NINJA_DIR
This propagates directly into the generated `ninja.build` file. From Ninja's docs:

> builddir A directory for some Ninja output files. ... (You can also store other build output in this directory.)

NINJA_DISABLE_AUTO_RUN
Boolean (True|False). Default: False When True, SCons will not run ninja automatically after creating the `ninja.build` file. If not set, this will be set to True if “--disable_execute_ninja” or `SetOption('disable_execute_ninja', True)`

NINJA_ENV_VAR_CACHE
A string that sets the environment for any environment variables that differ between the OS environment and the SCons command ENV. It will be compatible with the default shell of the operating system. If not explicitly

specified, SCons will generate this dynamically from the Environment()'s 'ENV' “env['ENV']” where those values differ from the existing shell..

NINJA_FILE_NAME

The filename for the generated Ninja build file defaults to `ninja.build`

NINJA_FORCE_SCONS_BUILD

When NINJA_FORCE_SCONS_BUILD is True, this will cause the build nodes to callback to scons instead of using ninja to build them. This is intended to be passed to the environment on the builder invocation. It is useful if you have a build node which does something which is not easily translated into ninja.

NINJA_GENERATED_SOURCE_SUFFIXES

The list of source file suffixes which are generated by SCons build steps. All source files which match these suffixes will be added to the _generated_sources alias in the output `ninja.build` file. Then all other source files will be made to depend on this in the `ninja.build` file, forcing the generated sources to be built first.

NINJA_MSVC_DEPS_PREFIX

This propagates directly into the generated `ninja.build` file. From Ninja's docs “defines the string which should be stripped from msvc’s /showIncludes output”

NINJA_POOL

Set the “ninja_pool” for this or all targets in scope for this env var.

NINJA_REGENERATE_DEPS

A generator function used to create a ninja depsfile which includes all the files which would require SCons to be invoked if they change. Or a list of said files.

_NINJA_REGENERATE_DEPS_FUNC

Internal value used to specify the function to call with argument env to generate the list of files which if changed would require the ninja file to be regenerated.

NINJA_SYNTAX

Theres also NINJA_SYNTAX which is the path to a custom ninja_syntax.py file which is used in generation. The tool currently assumes you have ninja installed through pip, and grabs the syntax file from that installation if none specified.

no_import_lib

When set to non-zero, suppresses creation of a corresponding Windows static import lib by the `SharedLibrary` builder when used with MinGW, Microsoft Visual Studio or Metrowerks. This also suppresses creation of an export (`.exp`) file when using Microsoft Visual Studio.

OBJPREFIX

The prefix used for (static) object file names.

OBJSUFFIX

The suffix used for (static) object file names.

PACKAGEROOT

Specifies the directory where all files in resulting archive will be placed if applicable. The default value is “`$NAME-$VERSION`”.

See the `Package` builder.

PACKAGETYPE

Selects the package type to build when using the `Package` builder. May be a string or list of strings. See the docuentation for the builder for the currently supported types.

`$PACKAGETYPE` may be overridden with the `--package-type` command line option.

See the `Package` builder.

PACKAGEVERSION

The version of the package (not the underlying project). This is currently only used by the rpm packager and should reflect changes in the packaging, not the underlying project code itself.

See the `Package` builder.

PCH

The Microsoft Visual C++ precompiled header that will be used when compiling object files. This variable is ignored by tools other than Microsoft Visual C++. When this variable is defined SCons will add options to the compiler command line to cause it to use the precompiled header, and will also set up the dependencies for the PCH file. Example:

```
env['PCH'] = File('StdAfx.pch')
```

PCHCOM

The command line used by the `PCH` builder to generated a precompiled header.

PCHCOMSTR

The string displayed when generating a precompiled header. If this is not set, then `$PCHCOM` (the command line) is displayed.

PCHPDBFLAGS

A construction variable that, when expanded, adds the `/yD` flag to the command line only if the `$PDB` construction variable is set.

PCHSTOP

This variable specifies how much of a source file is precompiled. This variable is ignored by tools other than Microsoft Visual C++, or when the PCH variable is not being used. When this variable is define it must be a string that is the name of the header that is included at the end of the precompiled portion of the source files, or the empty string if the "#pragma hrdstop" construct is being used:

```
env['PCHSTOP'] = 'StdAfx.h'
```

PDB

The Microsoft Visual C++ PDB file that will store debugging information for object files, shared libraries, and programs. This variable is ignored by tools other than Microsoft Visual C++. When this variable is defined SCons will add options to the compiler and linker command line to cause them to generate external debugging information, and will also set up the dependencies for the PDB file. Example:

```
env['PDB'] = 'hello.pdb'
```

The Visual C++ compiler switch that SCons uses by default to generate PDB information is `/Z7`. This works correctly with parallel (`-j`) builds because it embeds the debug information in the intermediate object files, as opposed to sharing a single PDB file between multiple object files. This is also the only way to get debug information embedded into a static library. Using the `/Zi` instead may yield improved link-time performance, although parallel builds will no longer work. You can generate PDB files with the `/Zi` switch by overriding the default `$CCPDBFLAGS` variable; see the entry for that variable for specific examples.

PDFLATEX

The pdflatex utility.

PDFLATEXCOM

The command line used to call the pdflatex utility.

PDFLATEXCOMSTR

The string displayed when calling the pdflatex utility. If this is not set, then `$PDFLATEXCOM` (the command line) is displayed.

```
env = Environment(PDFLATEX;COMSTR = "Building $TARGET from LaTeX input $SOURCES")
```

PDFLATEXFLAGS

General options passed to the pdflatex utility.

PDFPREFIX

The prefix used for PDF file names.

PDFSUFFIX

The suffix used for PDF file names.

PDFTEX

The pdftex utility.

PDFTEXCOM

The command line used to call the pdftex utility.

PDFTEXCOMSTR

The string displayed when calling the pdftex utility. If this is not set, then `$PDFTEXCOM` (the command line) is displayed.

```
env = Environment(PDFTEXCOMSTR = "Building $TARGET from TeX input $SOURCES")
```

PDFTEXFLAGS

General options passed to the pdftex utility.

PKGCHK

On Solaris systems, the package-checking program that will be used (along with `$PKGINFO`) to look for installed versions of the Sun PRO C++ compiler. The default is `/usr/sbin/pgkchk`.

PKGINFO

On Solaris systems, the package information program that will be used (along with `$PKGCHK`) to look for installed versions of the Sun PRO C++ compiler. The default is `pkginfo`.

PLATFORM

The name of the platform used to create this construction environment. SCons sets this when initializing the platform, which by default is auto-detected (see the *`platform`* argument to `Environment`).

```
env = Environment(tools=[])
if env['PLATFORM'] == 'cygwin':
    Tool('mingw')(env)
else:
    Tool('msvc')(env)
```

POAUTOINIT

The `$POAUTOINIT` variable, if set to `True` (on non-zero numeric value), let the `msginit` tool to automatically initialize *missing* `PO` files with **msginit(1)**. This applies to both, `POInit` and `POUpdate` builders (and others that use any of them).

POCREATE_ALIAS

Common alias for all PO files created with POInit builder (default: 'po-create'). See msginit tool and POInit builder.

POSUFFIX

Suffix used for PO files (default: '.po') See msginit tool and POInit builder.

POTDOMAIN

The $POTDOMAIN defines default domain, used to generate POT filename as $POTDOMAIN.pot when no POT file name is provided by the user. This applies to POTUpdate, POInit and POUpdate builders (and builders, that use them, e.g. Translate). Normally (if $POTDOMAIN is not defined), the builders use messages.pot as default POT file name.

POTSUFFIX

Suffix used for PO Template files (default: '.pot'). See xgettext tool and POTUpdate builder.

POTUPDATE_ALIAS

Name of the common phony target for all PO Templates created with POUpdate (default: 'pot-update'). See xgettext tool and POTUpdate builder.

POUPDATE_ALIAS

Common alias for all PO files being defined with POUpdate builder (default: 'po-update'). See msgmerge tool and POUpdate builder.

PRINT_CMD_LINE_FUNC

A Python function used to print the command lines as they are executed (assuming command printing is not disabled by the -q or -s options or their equivalents). The function should take four arguments: s, the command being executed (a string), target, the target being built (file node, list, or string name(s)), source, the source(s) used (file node, list, or string name(s)), and env, the environment being used.

The function must do the printing itself. The default implementation, used if this variable is not set or is None, is:

```
def print_cmd_line(s, target, source, env):
  sys.stdout.write(s + "\n")
```

Here's an example of a more interesting function:

```
def print_cmd_line(s, target, source, env):
   sys.stdout.write("Building %s -> %s...\n" %
    (' and '.join([str(x) for x in source]),
     ' and '.join([str(x) for x in target])))
env=Environment(PRINT_CMD_LINE_FUNC=print_cmd_line)
env.Program('foo', 'foo.c')
```

This just prints "Building targetname from sourcename..." instead of the actual commands. Such a function could also log the actual commands to a log file, for example.

PROGEMITTER

Contains the emitter specification for the Program builder. The manpage section "Builder Objects" contains general information on specifying emitters.

PROGPREFIX

The prefix used for executable file names.

PROGSUFFIX
The suffix used for executable file names.

PSCOM
The command line used to convert TeX DVI files into a PostScript file.

PSCOMSTR
The string displayed when a TeX DVI file is converted into a PostScript file. If this is not set, then `$PSCOM` (the command line) is displayed.

PSPREFIX
The prefix used for PostScript file names.

PSSUFFIX
The prefix used for PostScript file names.

QT_AUTOSCAN
Turn off scanning for mocable files. Use the `Moc` Builder to explicitly specify files to run **moc** on.

QT_BINPATH
The path where the Qt binaries are installed. The default value is '`$QTDIR/bin`'.

QT_CPPPATH
The path where the Qt header files are installed. The default value is '`$QTDIR`/include'. Note: If you set this variable to `None`, the tool won't change the `$CPPPATH` construction variable.

QT_DEBUG
Prints lots of debugging information while scanning for moc files.

QT_LIB
Default value is `'qt'`. You may want to set this to `'qt-mt'`. Note: If you set this variable to `None`, the tool won't change the `$LIBS` variable.

QT_LIBPATH
The path where the Qt libraries are installed. The default value is '`$QTDIR/lib`'. Note: If you set this variable to `None`, the tool won't change the `$LIBPATH` construction variable.

QT_MOC
Default value is '`$QT_BINPATH/moc`'.

QT_MOCCXXPREFIX
Default value is `''`. Prefix for **moc** output files when source is a C++ file.

QT_MOCCXXSUFFIX
Default value is `'.moc'`. Suffix for **moc** output files when source is a C++ file.

QT_MOCFROMCXXCOM
Command to generate a moc file from a C++ file.

QT_MOCFROMCXXCOMSTR
The string displayed when generating a moc file from a C++ file. If this is not set, then `$QT_MOCFROMCXXCOM` (the command line) is displayed.

QT_MOCFROMCXXFLAGS
Default value is `'-i'`. These flags are passed to **moc** when moccing a C++ file.

QT_MOCFROMHCOM
Command to generate a moc file from a header.

QT_MOCFROMHCOMSTR
The string displayed when generating a moc file from a C++ file. If this is not set, then `$QT_MOCFROMHCOM` (the command line) is displayed.

QT_MOCFROMHFLAGS
Default value is `''`. These flags are passed to **moc** when moccing a header file.

QT_MOCHPREFIX
Default value is `'moc_'`. Prefix for **moc** output files when source is a header.

QT_MOCHSUFFIX
Default value is '`$CXXFILESUFFIX`'. Suffix for moc output files when source is a header.

QT_UIC
Default value is '`$QT_BINPATH/uic`'.

QT_UICCOM
Command to generate header files from `.ui` files.

QT_UICCOMSTR
The string displayed when generating header files from `.ui` files. If this is not set, then `$QT_UICCOM` (the command line) is displayed.

QT_UICDECLFLAGS
Default value is ''. These flags are passed to **uic** when creating a header file from a `.ui` file.

QT_UICDECLPREFIX
Default value is `''`. Prefix for **uic** generated header files.

QT_UICDECLSUFFIX
Default value is `'.h'`. Suffix for **uic** generated header files.

QT_UICIMPLFLAGS
Default value is `''`. These flags are passed to **uic** when creating a C++ file from a `.ui` file.

QT_UICIMPLPREFIX
Default value is `'uic_'`. Prefix for uic generated implementation files.

QT_UICIMPLSUFFIX
Default value is '`$CXXFILESUFFIX`'. Suffix for uic generated implementation files.

QT_UISUFFIX
Default value is `'.ui'`. Suffix of designer input files.

QTDIR
The path to the Qt installation to build against. If not already set, `qt` tool tries to obtain this from `os.environ`; if not found there, it tries to make a guess.

RANLIB
The archive indexer.

RANLIBCOM
The command line used to index a static library archive.

RANLIBCOMSTR

The string displayed when a static library archive is indexed. If this is not set, then `$RANLIBCOM` (the command line) is displayed.

```
env = Environment(RANLIBCOMSTR = "Indexing $TARGET")
```

RANLIBFLAGS

General options passed to the archive indexer.

RC

The resource compiler used to build a Microsoft Visual C++ resource file.

RCCOM

The command line used to build a Microsoft Visual C++ resource file.

RCCOMSTR

The string displayed when invoking the resource compiler to build a Microsoft Visual C++ resource file. If this is not set, then `$RCCOM` (the command line) is displayed.

RCFLAGS

The flags passed to the resource compiler by the `RES` builder.

RCINCFLAGS

An automatically-generated construction variable containing the command-line options for specifying directories to be searched by the resource compiler. The value of `$RCINCFLAGS` is created by respectively prepending and appending `$RCINCPREFIX` and `$RCINCSUFFIX` to the beginning and end of each directory in `$CPPPATH`.

RCINCPREFIX

The prefix (flag) used to specify an include directory on the resource compiler command line. This will be prepended to the beginning of each directory in the `$CPPPATH` construction variable when the `$RCINCFLAGS` variable is expanded.

RCINCSUFFIX

The suffix used to specify an include directory on the resource compiler command line. This will be appended to the end of each directory in the `$CPPPATH` construction variable when the `$RCINCFLAGS` variable is expanded.

RDirs

A function that converts a string into a list of Dir instances by searching the repositories.

REGSVR

The program used on Windows systems to register a newly-built DLL library whenever the `SharedLibrary` builder is passed a keyword argument of `register=True`.

REGSVRCOM

The command line used on Windows systems to register a newly-built DLL library whenever the `SharedLibrary` builder is passed a keyword argument of `register=True`.

REGSVRCOMSTR

The string displayed when registering a newly-built DLL file. If this is not set, then `$REGSVRCOM` (the command line) is displayed.

REGSVRFLAGS

Flags passed to the DLL registration program on Windows systems when a newly-built DLL library is registered. By default, this includes the `/s` that prevents dialog boxes from popping up and requiring user attention.

RMIC

The Java RMI stub compiler.

RMICCOM

The command line used to compile stub and skeleton class files from Java classes that contain RMI implementations. Any options specified in the `$RMICFLAGS` construction variable are included on this command line.

RMICCOMSTR

The string displayed when compiling stub and skeleton class files from Java classes that contain RMI implementations. If this is not set, then `$RMICCOM` (the command line) is displayed.

```
env = Environment(RMICCOMSTR = "Generating stub/skeleton class files $TARGETS from $SOU
```

RMICFLAGS

General options passed to the Java RMI stub compiler.

RPATH

A list of paths to search for shared libraries when running programs. Currently only used in the GNU (gnulink), IRIX (sgilink) and Sun (sunlink) linkers. Ignored on platforms and toolchains that don't support it. Note that the paths added to RPATH are not transformed by **scons** in any way: if you want an absolute path, you must make it absolute yourself.

_RPATH

An automatically-generated construction variable containing the rpath flags to be used when linking a program with shared libraries. The value of `$_RPATH` is created by respectively prepending `$RPATHPREFIX` and appending `$RPATHSUFFIX` to the beginning and end of each directory in `$RPATH`.

RPATHPREFIX

The prefix used to specify a directory to be searched for shared libraries when running programs. This will be prepended to the beginning of each directory in the `$RPATH` construction variable when the `$_RPATH` variable is automatically generated.

RPATHSUFFIX

The suffix used to specify a directory to be searched for shared libraries when running programs. This will be appended to the end of each directory in the `$RPATH` construction variable when the `$_RPATH` variable is automatically generated.

RPCGEN

The RPC protocol compiler.

RPCGENCLIENTFLAGS

Options passed to the RPC protocol compiler when generating client side stubs. These are in addition to any flags specified in the `$RPCGENFLAGS` construction variable.

RPCGENFLAGS

General options passed to the RPC protocol compiler.

RPCGENHEADERFLAGS

Options passed to the RPC protocol compiler when generating a header file. These are in addition to any flags specified in the `$RPCGENFLAGS` construction variable.

RPCGENSERVICEFLAGS

Options passed to the RPC protocol compiler when generating server side stubs. These are in addition to any flags specified in the `$RPCGENFLAGS` construction variable.

RPCGENXDRFLAGS

Options passed to the RPC protocol compiler when generating XDR routines. These are in addition to any flags specified in the `$RPCGENFLAGS` construction variable.

SCANNERS

A list of the available implicit dependency scanners. New file scanners may be added by appending to this list, although the more flexible approach is to associate scanners with a specific Builder. See the manpage sections "Builder Objects" and "Scanner Objects" for more information.

SCONS_HOME

The (optional) path to the SCons library directory, initialized from the external environment. If set, this is used to construct a shorter and more efficient search path in the `$MSVSSCONS` command line executed from Microsoft Visual Studio project files.

SHCC

The C compiler used for generating shared-library objects. See also `$CC` for compiling to static objects.

SHCCCOM

The command line used to compile a C source file to a shared-library object file. Any options specified in the `$SHCFLAGS`, `$SHCCFLAGS` and `$CPPFLAGS` construction variables are included on this command line. See also `$CCCOM` for compiling to static objects.

SHCCCOMSTR

If set, the string displayed when a C source file is compiled to a shared object file. If not set, then `$SHCCCOM` (the command line) is displayed. See also `$CCCOMSTR` for compiling to static objects.

```
env = Environment(SHCCCOMSTR = "Compiling shared object $TARGET")
```

SHCCFLAGS

Options that are passed to the C and C++ compilers to generate shared-library objects. See also `$CCFLAGS` for compiling to static objects.

SHCFLAGS

Options that are passed to the C compiler (only; not C++) to generate shared-library objects. See also `$CFLAGS` for compiling to static objects.

SHCXX

The C++ compiler used for generating shared-library objects. See also `$CXX` for compiling to static objects.

SHCXXCOM

The command line used to compile a C++ source file to a shared-library object file. Any options specified in the `$SHCXXFLAGS` and `$CPPFLAGS` construction variables are included on this command line. See also `$CXXCOM` for compiling to static objects.

SHCXXCOMSTR

If set, the string displayed when a C++ source file is compiled to a shared object file. If not set, then `$SHCXXCOM` (the command line) is displayed. See also `$CXXCOMSTR` for compiling to static objects.

```
env = Environment(SHCXXCOMSTR = "Compiling shared object $TARGET")
```

SHCXXFLAGS

Options that are passed to the C++ compiler to generate shared-library objects. See also `$CXXFLAGS` for compiling to static objects.

SHDC

The name of the compiler to use when compiling D source destined to be in a shared objects. See also `$DC` for compiling to static objects.

SHDCOM

The command line to use when compiling code to be part of shared objects. See also `$DCOM` for compiling to static objects.

SHDCOMSTR

If set, the string displayed when a D source file is compiled to a (shared) object file. If not set, then `$SHDCOM` (the command line) is displayed. See also `$DCOMSTR` for compiling to static objects.

SHDLIBVERSIONFLAGS

Extra flags added to `$SHDLINKCOM` when building versioned `SharedLibrary`. These flags are only used when `$SHLIBVERSION` is set.

SHDLINK

The linker to use when creating shared objects for code bases include D sources. See also `$DLINK` for linking static objects.

SHDLINKCOM

The command line to use when generating shared objects. See also `$DLINKCOM` for linking static objects.

SHDLINKFLAGS

The list of flags to use when generating a shared object. See also `$DLINKFLAGS` for linking static objects.

SHELL

A string naming the shell program that will be passed to the `$SPAWN` function. See the `$SPAWN` construction variable for more information.

SHF03

The Fortran 03 compiler used for generating shared-library objects. You should normally set the `$SHFORTRAN` variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set `$SHF03` if you need to use a specific compiler or compiler version for Fortran 03 files.

SHF03COM

The command line used to compile a Fortran 03 source file to a shared-library object file. You only need to set `$SHF03COM` if you need to use a specific command line for Fortran 03 files. You should normally set the `$SHFORTRANCOM` variable, which specifies the default command line for all Fortran versions.

SHF03COMSTR

If set, the string displayed when a Fortran 03 source file is compiled to a shared-library object file. If not set, then `$SHF03COM` or `$SHFORTRANCOM` (the command line) is displayed.

SHF03FLAGS

Options that are passed to the Fortran 03 compiler to generated shared-library objects. You only need to set `$SHF03FLAGS` if you need to define specific user options for Fortran 03 files. You should normally set the `$SHFORTRANFLAGS` variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

SHF03PPCOM

The command line used to compile a Fortran 03 source file to a shared-library object file after first running the file through the C preprocessor. Any options specified in the `$SHF03FLAGS` and `$CPPFLAGS` construction variables are included on this command line. You only need to set `$SHF03PPCOM` if you need to use a specific C-preprocessor command line for Fortran 03 files. You should normally set the `$SHFORTRANPPCOM` variable, which specifies the default C-preprocessor command line for all Fortran versions.

SHF03PPCOMSTR

If set, the string displayed when a Fortran 03 source file is compiled to a shared-library object file after first running the file through the C preprocessor. If not set, then `$SHF03PPCOM` or `$SHFORTRANPPCOM` (the command line) is displayed.

SHF08

The Fortran 08 compiler used for generating shared-library objects. You should normally set the `$SHFORTRAN` variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set `$SHF08` if you need to use a specific compiler or compiler version for Fortran 08 files.

SHF08COM

The command line used to compile a Fortran 08 source file to a shared-library object file. You only need to set `$SHF08COM` if you need to use a specific command line for Fortran 08 files. You should normally set the `$SHFORTRANCOM` variable, which specifies the default command line for all Fortran versions.

SHF08COMSTR

If set, the string displayed when a Fortran 08 source file is compiled to a shared-library object file. If not set, then `$SHF08COM` or `$SHFORTRANCOM` (the command line) is displayed.

SHF08FLAGS

Options that are passed to the Fortran 08 compiler to generated shared-library objects. You only need to set `$SHF08FLAGS` if you need to define specific user options for Fortran 08 files. You should normally set the `$SHFORTRANFLAGS` variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

SHF08PPCOM

The command line used to compile a Fortran 08 source file to a shared-library object file after first running the file through the C preprocessor. Any options specified in the `$SHF08FLAGS` and `$CPPFLAGS` construction variables are included on this command line. You only need to set `$SHF08PPCOM` if you need to use a specific C-preprocessor command line for Fortran 08 files. You should normally set the `$SHFORTRANPPCOM` variable, which specifies the default C-preprocessor command line for all Fortran versions.

SHF08PPCOMSTR

If set, the string displayed when a Fortran 08 source file is compiled to a shared-library object file after first running the file through the C preprocessor. If not set, then `$SHF08PPCOM` or `$SHFORTRANPPCOM` (the command line) is displayed.

SHF77

The Fortran 77 compiler used for generating shared-library objects. You should normally set the `$SHFORTRAN` variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set `$SHF77` if you need to use a specific compiler or compiler version for Fortran 77 files.

SHF77COM

The command line used to compile a Fortran 77 source file to a shared-library object file. You only need to set `$SHF77COM` if you need to use a specific command line for Fortran 77 files. You should normally set the `$SHFORTRANCOM` variable, which specifies the default command line for all Fortran versions.

SHF77COMSTR

If set, the string displayed when a Fortran 77 source file is compiled to a shared-library object file. If not set, then `$SHF77COM` or `$SHFORTRANCOM` (the command line) is displayed.

SHF77FLAGS

Options that are passed to the Fortran 77 compiler to generated shared-library objects. You only need to set `$SHF77FLAGS` if you need to define specific user options for Fortran 77 files. You should normally set the `$SHFORTRANFLAGS` variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

SHF77PPCOM

The command line used to compile a Fortran 77 source file to a shared-library object file after first running the file through the C preprocessor. Any options specified in the `$SHF77FLAGS` and `$CPPFLAGS` construction variables are included on this command line. You only need to set `$SHF77PPCOM` if you need to use a specific

C-preprocessor command line for Fortran 77 files. You should normally set the `$SHFORTRANPPCOM` variable, which specifies the default C-preprocessor command line for all Fortran versions.

SHF77PPCOMSTR
If set, the string displayed when a Fortran 77 source file is compiled to a shared-library object file after first running the file through the C preprocessor. If not set, then `$SHF77PPCOM` or `$SHFORTRANPPCOM` (the command line) is displayed.

SHF90
The Fortran 90 compiler used for generating shared-library objects. You should normally set the `$SHFORTRAN` variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set `$SHF90` if you need to use a specific compiler or compiler version for Fortran 90 files.

SHF90COM
The command line used to compile a Fortran 90 source file to a shared-library object file. You only need to set `$SHF90COM` if you need to use a specific command line for Fortran 90 files. You should normally set the `$SHFORTRANCOM` variable, which specifies the default command line for all Fortran versions.

SHF90COMSTR
If set, the string displayed when a Fortran 90 source file is compiled to a shared-library object file. If not set, then `$SHF90COM` or `$SHFORTRANCOM` (the command line) is displayed.

SHF90FLAGS
Options that are passed to the Fortran 90 compiler to generated shared-library objects. You only need to set `$SHF90FLAGS` if you need to define specific user options for Fortran 90 files. You should normally set the `$SHFORTRANFLAGS` variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

SHF90PPCOM
The command line used to compile a Fortran 90 source file to a shared-library object file after first running the file through the C preprocessor. Any options specified in the `$SHF90FLAGS` and `$CPPFLAGS` construction variables are included on this command line. You only need to set `$SHF90PPCOM` if you need to use a specific C-preprocessor command line for Fortran 90 files. You should normally set the `$SHFORTRANPPCOM` variable, which specifies the default C-preprocessor command line for all Fortran versions.

SHF90PPCOMSTR
If set, the string displayed when a Fortran 90 source file is compiled to a shared-library object file after first running the file through the C preprocessor. If not set, then `$SHF90PPCOM` or `$SHFORTRANPPCOM` (the command line) is displayed.

SHF95
The Fortran 95 compiler used for generating shared-library objects. You should normally set the `$SHFORTRAN` variable, which specifies the default Fortran compiler for all Fortran versions. You only need to set `$SHF95` if you need to use a specific compiler or compiler version for Fortran 95 files.

SHF95COM
The command line used to compile a Fortran 95 source file to a shared-library object file. You only need to set `$SHF95COM` if you need to use a specific command line for Fortran 95 files. You should normally set the `$SHFORTRANCOM` variable, which specifies the default command line for all Fortran versions.

SHF95COMSTR
If set, the string displayed when a Fortran 95 source file is compiled to a shared-library object file. If not set, then `$SHF95COM` or `$SHFORTRANCOM` (the command line) is displayed.

SHF95FLAGS
Options that are passed to the Fortran 95 compiler to generated shared-library objects. You only need to set `$SHF95FLAGS` if you need to define specific user options for Fortran 95 files. You should normally set the

`$SHFORTRANFLAGS` variable, which specifies the user-specified options passed to the default Fortran compiler for all Fortran versions.

SHF95PPCOM
The command line used to compile a Fortran 95 source file to a shared-library object file after first running the file through the C preprocessor. Any options specified in the `$SHF95FLAGS` and `$CPPFLAGS` construction variables are included on this command line. You only need to set `$SHF95PPCOM` if you need to use a specific C-preprocessor command line for Fortran 95 files. You should normally set the `$SHFORTRANPPCOM` variable, which specifies the default C-preprocessor command line for all Fortran versions.

SHF95PPCOMSTR
If set, the string displayed when a Fortran 95 source file is compiled to a shared-library object file after first running the file through the C preprocessor. If not set, then `$SHF95PPCOM` or `$SHFORTRANPPCOM` (the command line) is displayed.

SHFORTRAN
The default Fortran compiler used for generating shared-library objects.

SHFORTRANCOM
The command line used to compile a Fortran source file to a shared-library object file.

SHFORTRANCOMSTR
If set, the string displayed when a Fortran source file is compiled to a shared-library object file. If not set, then `$SHFORTRANCOM` (the command line) is displayed.

SHFORTRANFLAGS
Options that are passed to the Fortran compiler to generate shared-library objects.

SHFORTRANPPCOM
The command line used to compile a Fortran source file to a shared-library object file after first running the file through the C preprocessor. Any options specified in the `$SHFORTRANFLAGS` and `$CPPFLAGS` construction variables are included on this command line.

SHFORTRANPPCOMSTR
If set, the string displayed when a Fortran source file is compiled to a shared-library object file after first running the file through the C preprocessor. If not set, then `$SHFORTRANPPCOM` (the command line) is displayed.

SHLIBEMITTER
Contains the emitter specification for the `SharedLibrary` builder. The manpage section "Builder Objects" contains general information on specifying emitters.

SHLIBNOVERSIONSYMLINKS
Instructs the `SharedLibrary` builder to not create symlinks for versioned shared libraries.

SHLIBPREFIX
The prefix used for shared library file names.

_SHLIBSONAME
A macro that automatically generates shared library's SONAME based on $TARGET, $SHLIBVERSION and $SHLIBSUFFIX. Used by `SharedLibrary` builder when the linker tool supports SONAME (e.g. `gnulink`).

SHLIBSUFFIX
The suffix used for shared library file names.

SHLIBVERSION
When this construction variable is defined, a versioned shared library is created by the `SharedLibrary` builder. This activates the `$_SHLIBVERSIONFLAGS` and thus modifies the `$SHLINKCOM` as required, adds the

version number to the library name, and creates the symlinks that are needed. `$SHLIBVERSION` versions should exist as alpha-numeric, decimal-delimited values as defined by the regular expression "\w+[\.\w+]*". Example `$SHLIBVERSION` values include '1', '1.2.3', and '1.2.gitaa412c8b'.

_SHLIBVERSIONFLAGS

This macro automatically introduces extra flags to `$SHLINKCOM` when building versioned `SharedLibrary` (that is when `$SHLIBVERSION` is set). `_SHLIBVERSIONFLAGS` usually adds `$SHLIBVERSIONFLAGS` and some extra dynamically generated options (such as `-Wl,-soname=$_SHLIBSONAME`. It is unused by "plain" (unversioned) shared libraries.

SHLIBVERSIONFLAGS

Extra flags added to `$SHLINKCOM` when building versioned `SharedLibrary`. These flags are only used when `$SHLIBVERSION` is set.

SHLINK

The linker for programs that use shared libraries. See also `$LINK` for linking static objects.

On POSIX systems (those using the `link` tool), you should normally not change this value as it defaults to a "smart" linker tool which selects a compiler driver matching the type of source files in use. So for example, if you set `$SHCXX` to a specific compiler name, and are compiling C++ sources, the smartlink function will automatically select the same compiler for linking.

SHLINKCOM

The command line used to link programs using shared libraries. See also `$LINKCOM` for linking static objects.

SHLINKCOMSTR

The string displayed when programs using shared libraries are linked. If this is not set, then `$SHLINKCOM` (the command line) is displayed. See also `$LINKCOMSTR` for linking static objects.

```
env = Environment(SHLINKCOMSTR = "Linking shared $TARGET")
```

SHLINKFLAGS

General user options passed to the linker for programs using shared libraries. Note that this variable should *not* contain `-l` (or similar) options for linking with the libraries listed in `$LIBS`, nor `-L` (or similar) include search path options that scons generates automatically from `$LIBPATH`. See `$_LIBFLAGS` above, for the variable that expands to library-link options, and `$_LIBDIRFLAGS` above, for the variable that expands to library search path options. See also `$LINKFLAGS` for linking static objects.

SHOBJPREFIX

The prefix used for shared object file names.

SHOBJSUFFIX

The suffix used for shared object file names.

SONAME

Variable used to hard-code SONAME for versioned shared library/loadable module.

```
env.SharedLibrary('test', 'test.c', SHLIBVERSION='0.1.2', SONAME='libtest.so.2')
```

The variable is used, for example, by `gnulink` linker tool.

SOURCE

A reserved variable name that may not be set or used in a construction environment. (See the manpage section "Variable Substitution" for more information).

SOURCE_URL

The URL (web address) of the location from which the project was retrieved. This is used to fill in the `Source:` field in the controlling information for Ipkg and RPM packages.

See the `Package` builder.

SOURCES

A reserved variable name that may not be set or used in a construction environment. (See the manpage section "Variable Substitution" for more information).

SOVERSION

This will construct the `SONAME` using on the base library name (*`test`* in the example below) and use specified `SOVERSION` to create `SONAME`.

```
env.SharedLibrary('test', 'test.c', SHLIBVERSION='0.1.2', SOVERSION='2')
```

The variable is used, for example, by `gnulink` linker tool.

In the example above `SONAME` would be `libtest.so.2` which would be a symlink and point to `libtest.so.0.1.2`

SPAWN

A command interpreter function that will be called to execute command line strings. The function must expect the following arguments:

```
def spawn(shell, escape, cmd, args, env):
```

`sh` is a string naming the shell program to use. `escape` is a function that can be called to escape shell special characters in the command line. `cmd` is the path to the command to be executed. `args` is the arguments to the command. `env` is a dictionary of the environment variables in which the command should be executed.

STATIC_AND_SHARED_OBJECTS_ARE_THE_SAME

When this variable is true, static objects and shared objects are assumed to be the same; that is, SCons does not check for linking static objects into a shared library.

SUBST_DICT

The dictionary used by the `Substfile` or `Textfile` builders for substitution values. It can be anything acceptable to the `dict()` constructor, so in addition to a dictionary, lists of tuples are also acceptable.

SUBSTFILEPREFIX

The prefix used for `Substfile` file names, an empty string by default.

SUBSTFILESUFFIX

The suffix used for `Substfile` file names, an empty string by default.

SUMMARY

A short summary of what the project is about. This is used to fill in the `Summary:` field in the controlling information for Ipkg and RPM packages, and as the `Description:` field in MSI packages.

See the `Package` builder.

SWIG

The scripting language wrapper and interface generator.

SWIGCFILESUFFIX

The suffix that will be used for intermediate C source files generated by the scripting language wrapper and interface generator. The default value is `_wrap$CFILESUFFIX`. By default, this value is used whenever the `-c++` option is *not* specified as part of the `$SWIGFLAGS` construction variable.

SWIGCOM

The command line used to call the scripting language wrapper and interface generator.

SWIGCOMSTR

The string displayed when calling the scripting language wrapper and interface generator. If this is not set, then `$SWIGCOM` (the command line) is displayed.

SWIGCXXFILESUFFIX

The suffix that will be used for intermediate C++ source files generated by the scripting language wrapper and interface generator. The default value is `_wrap$CFILESUFFIX`. By default, this value is used whenever the `-c++` option is specified as part of the `$SWIGFLAGS` construction variable.

SWIGDIRECTORSUFFIX

The suffix that will be used for intermediate C++ header files generated by the scripting language wrapper and interface generator. These are only generated for C++ code when the SWIG 'directors' feature is turned on. The default value is `_wrap.h`.

SWIGFLAGS

General options passed to the scripting language wrapper and interface generator. This is where you should set `-python`, `-perl5`, `-tcl`, or whatever other options you want to specify to SWIG. If you set the `-c++` option in this variable, **scons** will, by default, generate a C++ intermediate source file with the extension that is specified as the `$CXXFILESUFFIX` variable.

_SWIGINCFLAGS

An automatically-generated construction variable containing the SWIG command-line options for specifying directories to be searched for included files. The value of `$_SWIGINCFLAGS` is created by respectively prepending and appending `$SWIGINCPREFIX` and `$SWIGINCSUFFIX` to the beginning and end of each directory in `$SWIGPATH`.

SWIGINCPREFIX

The prefix used to specify an include directory on the SWIG command line. This will be prepended to the beginning of each directory in the `$SWIGPATH` construction variable when the `$_SWIGINCFLAGS` variable is automatically generated.

SWIGINCSUFFIX

The suffix used to specify an include directory on the SWIG command line. This will be appended to the end of each directory in the `$SWIGPATH` construction variable when the `$_SWIGINCFLAGS` variable is automatically generated.

SWIGOUTDIR

Specifies the output directory in which the scripting language wrapper and interface generator should place generated language-specific files. This will be used by SCons to identify the files that will be generated by the swig call, and translated into the `swig -outdir` option on the command line.

SWIGPATH

The list of directories that the scripting language wrapper and interface generate will search for included files. The SWIG implicit dependency scanner will search these directories for include files. The default value is an empty list.

Don't explicitly put include directory arguments in SWIGFLAGS; the result will be non-portable and the directories will not be searched by the dependency scanner. Note: directory names in SWIGPATH will be looked-up relative to the SConscript directory when they are used in a command. To force **scons** to look-up a directory relative to the root of the source tree use #:

```
env = Environment(SWIGPATH='#/include')
```

The directory look-up can also be forced using the `Dir`() function:

```
include = Dir('include')
env = Environment(SWIGPATH=include)
```

The directory list will be added to command lines through the automatically-generated `$_SWIGINCFLAGS` construction variable, which is constructed by respectively prepending and appending the values of the `$SWIGINCPREFIX` and `$SWIGINCSUFFIX` construction variables to the beginning and end of each directory in `$SWIGPATH`. Any command lines you define that need the SWIGPATH directory list should include `$_SWIGINCFLAGS`:

```
env = Environment(SWIGCOM="my_swig -o $TARGET $_SWIGINCFLAGS $SOURCES")
```

SWIGVERSION
The version number of the SWIG tool.

TAR
The tar archiver.

TARCOM
The command line used to call the tar archiver.

TARCOMSTR
The string displayed when archiving files using the tar archiver. If this is not set, then `$TARCOM` (the command line) is displayed.

```
env = Environment(TARCOMSTR = "Archiving $TARGET")
```

TARFLAGS
General options passed to the tar archiver.

TARGET
A reserved variable name that may not be set or used in a construction environment. (See the manpage section "Variable Substitution" for more information).

TARGET_ARCH
The name of the hardware architecture that objects created using this construction environment should target. Can be set when creating a construction environment by passing as a keyword argument in the `Environment` call.

On the `win32` platform, if the Microsoft Visual C++ compiler is available, `msvc` tool setup is done using `$HOST_ARCH` and `$TARGET_ARCH`. If a value is not specified, will be set to the same value as `$HOST_ARCH`. Changing the value after the environment is initialized will not cause the tool to be reinitialized. Compiled objects will be in the target architecture if the compilation system supports generating for that target. The latest compiler which can fulfill the requirement will be selected, unless a different version is directed by the value of the `$MSVC_VERSION` construction variable.

On the win32/msvc combination, valid target arch values are `x86`, `arm`, `i386` for 32-bit targets and `amd64`, `arm64`, `x86_64` and `ia64` (Itanium) for 64-bit targets. For example, if you want to compile 64-bit binaries, you would set `TARGET_ARCH='x86_64'` when creating the construction environment. Note that not all target architectures are supported for all Visual Studio / MSVC versions. Check the relevant Microsoft documentation.

`$TARGET_ARCH` is not currently used by other compilation tools, but the option is reserved to do so in future

TARGET_OS

The name of the operating system that objects created using this construction environment should target. Can be set when creating a construction environment by passing as a keyword argument in the `Environment` call;.

`$TARGET_OS` is not currently used by SCons but the option is reserved to do so in future

TARGETS

A reserved variable name that may not be set or used in a construction environment. (See the manpage section "Variable Substitution" for more information).

TARSUFFIX

The suffix used for tar file names.

TEMPFILE

A callable object used to handle overly long command line strings, since operations which call out to a shell will fail if the line is longer than the shell can accept. This tends to particularly impact linking. The tempfile object stores the command line in a temporary file in the appropriate format, and returns an alternate command line so the invoked tool will make use of the contents of the temporary file. If you need to replace the default tempfile object, the callable should take into account the settings of `$MAXLINELENGTH`, `$TEMPFILEPREFIX`, `$TEMPFILESUFFIX`, `$TEMPFILEARGJOIN`, `$TEMPFILEDIR` and `$TEMPFILEARGESCFUNC`.

TEMPFILEARGESCFUNC

The default argument escape function is `SCons.Subst.quote_spaces`. If you need to apply extra operations on a command argument (to fix Windows slashes, normalize paths, etc.) before writing to the temporary file, you can set the `$TEMPFILEARGESCFUNC` variable to a custom function. Such a function takes a single string argument and returns a new string with any modifications applied. Example:

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

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

def tempfile_arg_esc_func(arg):
    arg = quote_spaces(arg)
    if sys.platform != "win32":
        return arg
    # GCC requires double Windows slashes, let's use UNIX separator
    return WINPATHSEP_RE.sub(r"/\1", arg)

env["TEMPFILEARGESCFUNC"] = tempfile_arg_esc_func
```

TEMPFILEARGJOIN

The string to use to join the arguments passed to `$TEMPFILE` when the command line exceeds the limit set by `$MAXLINELENGTH`. The default value is a space. However for MSVC, MSLINK the default is a line separator as defined by `os.linesep`. Note this value is used literally and not expanded by the subst logic.

TEMPFILEDIR

The directory to create the long-lines temporary file in.

TEMPFILEPREFIX
The prefix for the name of the temporary file used to store command lines exceeding `$MAXLINELENGTH`. The default prefix is `'@'`, which works for the Microsoft and GNU toolchains on Windows. Set this appropriately for other toolchains, for example `'-@'` for the diab compiler or `'-via'` for ARM toolchain.

TEMPFILESUFFIX
The suffix for the name of the temporary file used to store command lines exceeding `$MAXLINELENGTH`. The suffix should include the dot ('.') if one is wanted as it will not be added automatically. The default is `.lnk`.

TEX
The TeX formatter and typesetter.

TEXCOM
The command line used to call the TeX formatter and typesetter.

TEXCOMSTR
The string displayed when calling the TeX formatter and typesetter. If this is not set, then `$TEXCOM` (the command line) is displayed.

```
env = Environment(TEXCOMSTR = "Building $TARGET from TeX input $SOURCES")
```

TEXFLAGS
General options passed to the TeX formatter and typesetter.

TEXINPUTS
List of directories that the LaTeX program will search for include directories. The LaTeX implicit dependency scanner will search these directories for \include and \import files.

TEXTFILEPREFIX
The prefix used for `Textfile` file names, an empty string by default.

TEXTFILESUFFIX
The suffix used for `Textfile` file names; `.txt` by default.

TOOLS
A list of the names of the Tool specifications that are part of this construction environment.

UNCHANGED_SOURCES
A reserved variable name that may not be set or used in a construction environment. (See the manpage section "Variable Substitution" for more information).

UNCHANGED_TARGETS
A reserved variable name that may not be set or used in a construction environment. (See the manpage section "Variable Substitution" for more information).

VENDOR
The person or organization who supply the packaged software. This is used to fill in the `Vendor:` field in the controlling information for RPM packages, and the `Manufacturer:` field in the controlling information for MSI packages.

See the `Package` builder.

VERSION
The version of the project, specified as a string.

See the `Package` builder.

VSWHERE

Specify the location of `vswhere.exe`.

The `vswhere.exe` executable is distributed with Microsoft Visual Studio and Build Tools since the 2017 edition, but is also available standalone. It provides full information about installations of 2017 and later editions. With the `-legacy` argument, `vswhere.exe` can detect installations of the 2010 through 2015 editions with limited data returned. If `VSWHERE` is set, SCons will use that location.

Otherwise SCons will look in the following locations and set `VSWHERE` to the path of the first `vswhere.exe` located.

- `%ProgramFiles(x86)%\Microsoft Visual Studio\Installer`
- `%ProgramFiles%\Microsoft Visual Studio\Installer`
- `%ChocolateyInstall%\bin`

Note that `VSWHERE` must be set at the same time or prior to any of `msvc`, `msvs` , and/or `mslink` `Tool` being initialized. Either set it as follows

```

env = Environment(VSWHERE='c:/my/path/to/vswhere')
```

or if your construction environment is created specifying an empty tools list (or a list of tools which omits all of default, msvs, msvc, and mslink), and also before `env.Tool` is called to ininitialize any of those tools:

```

    env = Environment(tools=[])
    env['VSWHERE'] = r'c:/my/vswhere/install/location/vswhere.exe'
    env.Tool('msvc')
    env.Tool('mslink')
    env.Tool('msvs')
```

WINDOWS_EMBED_MANIFEST

Set to `True` to embed the compiler-generated manifest (normally `${TARGET}.manifest`) into all Windows executables and DLLs built with this environment, as a resource during their link step. This is done using `$MT` and `$MTEXECOM` and `$MTSHLIBCOM`. See also `$WINDOWS_INSERT_MANIFEST`.

WINDOWS_INSERT_DEF

If set to true, a library build of a Windows shared library (`.dll` file) will include a reference to the corresponding module-definition file at the same time, if a module-definition file is not already listed as a build target. The name of the module-definition file will be constructed from the base name of the library and the construction variables `$WINDOWSDEFSUFFIX` and `$WINDOWSDEFPREFIX`. The default is to not add a module-definition file. The module-definition file is not created by this directive, and must be supplied by the developer.

WINDOWS_INSERT_MANIFEST

If set to true, **scons** will add the manifest file generated by Microsoft Visual C++ 8.0 and later to the target list so SCons will be aware they were generated. In the case of an executable, the manifest file name is constructed using `$WINDOWSPROGMANIFESTSUFFIX` and `$WINDOWSPROGMANIFESTPREFIX`. In the case of a shared library, the manifest file name is constructed using `$WINDOWSSHLIBMANIFESTSUFFIX` and `$WINDOWSSHLIBMANIFESTPREFIX`. See also `$WINDOWS_EMBED_MANIFEST`.

WINDOWSDEFPREFIX

The prefix used for a Windows linker module-definition file name. Defaults to empty.

WINDOWSDEFSUFFIX
The suffix used for a Windows linker module-definition file name. Defaults to `.def`.

WINDOWSEXPPREFIX
The prefix used for Windows linker exports file names. Defaults to empty.

WINDOWSEXPSUFFIX
The suffix used for Windows linker exports file names. Defaults to `.exp`.

WINDOWSPROGMANIFESTPREFIX
The prefix used for executable program manifest files generated by Microsoft Visual C/C++. Defaults to empty.

WINDOWSPROGMANIFESTSUFFIX
The suffix used for executable program manifest files generated by Microsoft Visual C/C++. Defaults to `.manifest`.

WINDOWSSHLIBMANIFESTPREFIX
The prefix used for shared library manifest files generated by Microsoft Visual C/C++. Defaults to empty.

WINDOWSSHLIBMANIFESTSUFFIX
The suffix used for shared library manifest files generated by Microsoft Visual C/C++. Defaults to `.manifest`.

X_IPK_DEPENDS
This is used to fill in the `Depends:` field in the controlling information for Ipkg packages.

See the `Package` builder.

X_IPK_DESCRIPTION
This is used to fill in the `Description:` field in the controlling information for Ipkg packages. The default value is “`$SUMMARY`\n`$DESCRIPTION`”

X_IPK_MAINTAINER
This is used to fill in the `Maintainer:` field in the controlling information for Ipkg packages.

X_IPK_PRIORITY
This is used to fill in the `Priority:` field in the controlling information for Ipkg packages.

X_IPK_SECTION
This is used to fill in the `Section:` field in the controlling information for Ipkg packages.

X_MSI_LANGUAGE
This is used to fill in the `Language:` attribute in the controlling information for MSI packages.

See the `Package` builder.

X_MSI_LICENSE_TEXT
The text of the software license in RTF format. Carriage return characters will be replaced with the RTF equivalent \\par.

See the `Package` builder.

X_MSI_UPGRADE_CODE
TODO

X_RPM_AUTOREQPROV
This is used to fill in the `AutoReqProv:` field in the RPM `.spec` file.

See the `Package` builder.

`X_RPM_BUILD`
internal, but overridable

`X_RPM_BUILDREQUIRES`
This is used to fill in the `BuildRequires:` field in the RPM `.spec` file. Note this should only be used on a host managed by rpm as the dependencies will not be resolvable at build time otherwise.

`X_RPM_BUILDROOT`
internal, but overridable

`X_RPM_CLEAN`
internal, but overridable

`X_RPM_CONFLICTS`
This is used to fill in the `Conflicts:` field in the RPM `.spec` file.

`X_RPM_DEFATTR`
This value is used as the default attributes for the files in the RPM package. The default value is “(-,root,root)”.

`X_RPM_DISTRIBUTION`
This is used to fill in the `Distribution:` field in the RPM `.spec` file.

`X_RPM_EPOCH`
This is used to fill in the `Epoch:` field in the RPM `.spec` file.

`X_RPM_EXCLUDEARCH`
This is used to fill in the `ExcludeArch:` field in the RPM `.spec` file.

`X_RPM_EXLUSIVEARCH`
This is used to fill in the `ExclusiveArch:` field in the RPM `.spec` file.

`X_RPM_EXTRADEFS`
A list used to supply extra defintions or flags to be added to the RPM `.spec` file. Each item is added as-is with a carriage return appended. This is useful if some specific RPM feature not otherwise anticipated by SCons needs to be turned on or off. Note if this variable is omitted, SCons will by default supply the value `'%global debug_package %{nil}'` to disable debug package generation. To enable debug package generation, include this variable set either to None, or to a custom list that does not include the default line. Added in version 3.1.

```
env.Package(
    NAME="foo",
    ...
    X_RPM_EXTRADEFS=[
        "%define _unpackaged_files_terminate_build 0"
        "%define _missing_doc_files_terminate_build 0"
    ],
    ...
)
```

`X_RPM_GROUP`
This is used to fill in the `Group:` field in the RPM `.spec` file.

`X_RPM_GROUP_lang`
This is used to fill in the `Group(lang):` field in the RPM `.spec` file. Note that `lang` is not literal and should be replaced by the appropriate language code.

X_RPM_ICON
: This is used to fill in the `Icon:` field in the RPM `.spec` file.

X_RPM_INSTALL
: internal, but overridable

X_RPM_PACKAGER
: This is used to fill in the `Packager:` field in the RPM `.spec` file.

X_RPM_POSTINSTALL
: This is used to fill in the `%post:` section in the RPM `.spec` file.

X_RPM_POSTUNINSTALL
: This is used to fill in the `%postun:` section in the RPM `.spec` file.

X_RPM_PREFIX
: This is used to fill in the `Prefix:` field in the RPM `.spec` file.

X_RPM_PREINSTALL
: This is used to fill in the `%pre:` section in the RPM `.spec` file.

X_RPM_PREP
: internal, but overridable

X_RPM_PREUNINSTALL
: This is used to fill in the `%preun:` section in the RPM `.spec` file.

X_RPM_PROVIDES
: This is used to fill in the `Provides:` field in the RPM `.spec` file.

X_RPM_REQUIRES
: This is used to fill in the `Requires:` field in the RPM `.spec` file.

X_RPM_SERIAL
: This is used to fill in the `Serial:` field in the RPM `.spec` file.

X_RPM_URL
: This is used to fill in the `Url:` field in the RPM `.spec` file.

XGETTEXT
: Path to **xgettext(1)** program (found via `Detect()`). See `xgettext` tool and `POTUpdate` builder.

XGETTEXTCOM
: Complete xgettext command line. See `xgettext` tool and `POTUpdate` builder.

XGETTEXTCOMSTR
: A string that is shown when **xgettext(1)** command is invoked (default: `''`, which means "print `$XGETTEXTCOM`"). See `xgettext` tool and `POTUpdate` builder.

_XGETTEXTDOMAIN
: Internal "macro". Generates **xgettext** domain name form source and target (default: `'${TARGET.filebase}'`).

XGETTEXTFLAGS
: Additional flags to **xgettext(1)**. See `xgettext` tool and `POTUpdate` builder.

XGETTEXTFROM

Name of file containing list of **xgettext(1)**'s source files. Autotools' users know this as `POTFILES.in` so they will in most cases set `XGETTEXTFROM="POTFILES.in"` here. The `$XGETTEXTFROM` files have same syntax and semantics as the well known GNU `POTFILES.in`. See `xgettext` tool and `POTUpdate` builder.

_XGETTEXTFROMFLAGS

Internal "macro". Genrates list of `-D<dir>` flags from the `$XGETTEXTPATH` list.

XGETTEXTFROMPREFIX

This flag is used to add single `$XGETTEXTFROM` file to **xgettext(1)**'s commandline (default: `'-f'`).

XGETTEXTFROMSUFFIX

(default: `''`)

XGETTEXTPATH

List of directories, there **xgettext(1)** will look for source files (default: `[]`).

Note

This variable works only together with `$XGETTEXTFROM`

See also `xgettext` tool and `POTUpdate` builder.

_XGETTEXTPATHFLAGS

Internal "macro". Generates list of `-f<file>` flags from `$XGETTEXTFROM`.

XGETTEXTPATHPREFIX

This flag is used to add single search path to **xgettext(1)**'s commandline (default: `'-D'`).

XGETTEXTPATHSUFFIX

(default: `''`)

YACC

The parser generator.

YACCCOM

The command line used to call the parser generator to generate a source file.

YACCCOMSTR

The string displayed when generating a source file using the parser generator. If this is not set, then `$YACCCOM` (the command line) is displayed.

```
env = Environment(YACCCOMSTR = "Yacc'ing $TARGET from $SOURCES")
```

YACCFLAGS

General options passed to the parser generator. If `$YACCFLAGS` contains a `-d` option, SCons assumes that the call will also create a .h file (if the yacc source file ends in a .y suffix) or a .hpp file (if the yacc source file ends in a .yy suffix)

YACCHFILESUFFIX

The suffix of the C header file generated by the parser generator when the `-d` option is used. Note that setting this variable does not cause the parser generator to generate a header file with the specified suffix, it exists to allow you to specify what suffix the parser generator will use of its own accord. The default value is `.h`.

YACCHXXFILESUFFIX

The suffix of the C++ header file generated by the parser generator when the `-d` option is used. Note that setting this variable does not cause the parser generator to generate a header file with the specified suffix, it exists to allow you to specify what suffix the parser generator will use of its own accord. The default value is `.hpp`, except on Mac OS X, where the default is `${TARGET.suffix}.h`. because the default bison parser generator just appends `.h` to the name of the generated C++ file.

`YACCVCGFILESUFFIX`

The suffix of the file containing the VCG grammar automaton definition when the `--graph=` option is used. Note that setting this variable does not cause the parser generator to generate a VCG file with the specified suffix, it exists to allow you to specify what suffix the parser generator will use of its own accord. The default value is `.vcg`.

`ZIP`

The zip compression and file packaging utility.

`ZIP_OVERRIDE_TIMESTAMP`

An optional timestamp which overrides the last modification time of the file when stored inside the Zip archive. This is a tuple of six values: Year (>= 1980) Month (one-based) Day of month (one-based) Hours (zero-based) Minutes (zero-based) Seconds (zero-based)

`ZIPCOM`

The command line used to call the zip utility, or the internal Python function used to create a zip archive.

`ZIPCOMPRESSION`

The `compression` flag from the Python `zipfile` module used by the internal Python function to control whether the zip archive is compressed or not. The default value is `zipfile.ZIP_DEFLATED`, which creates a compressed zip archive. This value has no effect if the `zipfile` module is unavailable.

`ZIPCOMSTR`

The string displayed when archiving files using the zip utility. If this is not set, then `$ZIPCOM` (the command line or internal Python function) is displayed.

```
env = Environment(ZIPCOMSTR = "Zipping $TARGET")
```

`ZIPFLAGS`

General options passed to the zip utility.

`ZIPROOT`

An optional zip root directory (default empty). The filenames stored in the zip file will be relative to this directory, if given. Otherwise the filenames are relative to the current directory of the command. For instance:

```
env = Environment()
env.Zip('foo.zip', 'subdir1/subdir2/file1', ZIPROOT='subdir1')
```

will produce a zip file `foo.zip` containing a file with the name `subdir2/file1` rather than `subdir1/subdir2/file1`.

`ZIPSUFFIX`

The suffix used for zip file names.

Configure Contexts

SCons supports a *configure context*, an integrated mechanism similar to the various `AC_CHECK` macros in GNU Autoconf for testing the existence of external items needed for the build, such as C header files, libraries, etc. The mechanism is portable across platforms.

scons does not maintain an explicit cache of the tested values (this is different than Autoconf), but uses its normal dependency tracking to keep the checked values up to date. However, users may override this behaviour with the `--config` command line option.

Configure(*env, [custom_tests, conf_dir, log_file, config_h, clean, help]*)
***env*.Configure(*[custom_tests, conf_dir, log_file, config_h, clean, help]*)**

Create a configure context, which tracks information discovered while running tests. The context includes a local construction environment (available as *context*.env) which is used when running the tests and which can be updated with the check results. Only one context may be active at a time (*since 4.0,* ***scons*** *will raise an exception on an attempt to create a new context when there is an active context*), but a new context can be created after the active one is completed. For the global function form, the required *env* describes the initial values for the context's local construction environment; for the construction environment method form the instance provides the values.

custom_tests specifies a dictionary containing custom tests (see the section on custom tests below). The default value is `None`, meaning no custom tests are added to the configure context.

conf_dir specifies a directory where the test cases are built. This directory is not used for building normal targets. The default value is “`#/.sconf_temp`”.

log_file specifies a file which collects the output from commands that are executed to check for the existence of header files, libraries, etc. The default is “`#/config.log`”. If you are using the `VariantDir` function, you may want to specify a subdirectory under your variant directory.

config_h specifies a C header file where the results of tests will be written. The results will consist of lines like `#define HAVE_STDIO_H`, `#define HAVE_LIBM`, etc. Customarily, the name chosen is “`config.h`”. The default is to not write a *config_h* file. You can specify the same *config_h* file in multiple calls to `Configure`, in which case SCons will concatenate all results in the specified file. Note that SCons uses its normal dependency checking to decide if it's necessary to rebuild the specified *config_h* file. This means that the file is not necessarily re-built each time scons is run, but is only rebuilt if its contents will have changed and some target that depends on the *config_h* file is being built.

The *clean* and *help* arguments can be used to suppress execution of the configuration tests when the `-c`/`--clean` or `-H`/`-h`/`--help` options are used, respectively. The default behavior is always to execute configure context tests, since the results of the tests may affect the list of targets to be cleaned or the help text. If the configure tests do not affect these, then you may add the `clean=False` or `help=False` arguments (or both) to avoid unnecessary test execution.

SConf.Finish(*context*)
***context*.Finish()**

This method must be called after configuration is done. Though required, this is not enforced except if `Configure` is called again while there is still an active context, in which case an exception is raised. `Finish` returns the environment as modified during the course of running the configuration checks. After this method is called, no further checks can be performed with this configuration context. However, you can create a new configure context to perform additional checks.

Example of a typical Configure usage:

```
env = Environment()
conf = Configure(env)
if not conf.CheckCHeader("math.h"):
    print("We really need math.h!")
    Exit(1)
if conf.CheckLibWithHeader("qt", "qapp.h", "c++", "QApplication qapp(0,0);"):
    # do stuff for qt - usage, e.g.
    conf.env.Append(CPPDEFINES="WITH_QT")
env = conf.Finish()
```

A configure context has the following predefined methods which can be used to perform checks. Where `language` is a required or optional parameter, the choice can currently be C or C++. The spellings accepted for C are “C” or “c”; for C++ the value can be “CXX”, “cxx”, “C++” or “c++”.

SConf.CheckHeader(*context, header, [include_quotes, language]*)
***context*.CheckHeader(*header, [include_quotes, language]*)**

Checks if *header* is usable in the specified language. *header* may be a list, in which case the last item in the list is the header file to be checked, and the previous list items are header files whose `#include` lines should precede the header line being checked for. The optional argument *include_quotes* must be a two character string, where the first character denotes the opening quote and the second character denotes the closing quote. By default, both characters are " (double quote). The optional argument *language* should be either **C** or **C++** and selects the compiler to be used for the check. Returns a boolean indicating success or failure.

SConf.CheckCHeader(*context, header, [include_quotes]*)
***context*.CheckCHeader(*header, [include_quotes]*)**

This is a wrapper around `SConf.CheckHeader` which checks if *header* is usable in the C language. *header* may be a list, in which case the last item in the list is the header file to be checked, and the previous list items are header files whose `#include` lines should precede the header line being checked for. The optional argument *include_quotes* must be a two character string, where the first character denotes the opening quote and the second character denotes the closing quote. By default, both characters are " (double quote). Returns a boolean indicating success or failure.

SConf.CheckCXXHeader(*context, header, [include_quotes]*)
***context*.CheckCXXHeader(*header, [include_quotes]*)**

This is a wrapper around `SConf.CheckHeader` which checks if *header* is usable in the C++ language. *header* may be a list, in which case the last item in the list is the header file to be checked, and the previous list items are header files whose `#include` lines should precede the header line being checked for. The optional argument *include_quotes* must be a two character string, where the first character denotes the opening quote and the second character denotes the closing quote. By default, both characters are " (double quote). Returns a boolean indicating success or failure.

SConf.CheckFunc(*context, function_name, [header, language]*)
***context*.CheckFunc(*function_name, [header, language]*)**

Checks if the specified C or C++ library function is available based on the context's local environment settings (that is, using the values of `CFLAGS`, `CPPFLAGS`, `LIBS` or other relevant construction variables).

function_name is the name of the function to check for. The optional *header* argument is a string that will be placed at the top of the test file that will be compiled to check if the function exists; the default is:

```

#ifdef __cplusplus
extern "C"
#endif
char function_name();
```

Returns an empty string on success, a string containing an error message on failure.

SConf.CheckLib(*context, [library, symbol, header, language, autoadd=True]*)
***context*.CheckLib(*[library, symbol, header, language, autoadd=True]*)**

Checks if *library* provides *symbol*. If *autoadd* is true (the default) and the library provides the specified *symbol*, appends the library to the `LIBS` construction variable *library* may also be `None` (the default), in which case *symbol* is checked with the current `LIBS` variable, or a list of library names, in which case each library in the list will be checked for *symbol*. If *symbol* is not set or is None, then SConf.CheckLib just checks if you can link against the specified *library*. Note though it is legal syntax, it would not be very useful to call this method with *library* and *symbol* both omitted or None. Returns a boolean indicating success or failure.

SConf.CheckLibWithHeader(*context, library, header, language, [call, autoadd=True]*)
***context*.CheckLibWithHeader(*library, header, language, [call, autoadd=True]*)**

Provides a more sophisticated way to check against libraries then the SConf.CheckLib call. *library* specifies the library or a list of libraries to check. *header* specifies a header to check for. *header* may be a list, in which case the last item in the list is the header file to be checked, and the previous list items are header files whose #include lines should precede the header line being checked for. *call* can be any valid expression (with a trailing ';'). If *call* is not set, the default simply checks that you can link against the specified *library*. *autoadd* (default true) specifies whether to add the library to the environment if the check succeeds. Returns a boolean indicating success or failure.

SConf.CheckType(*context, type_name, [includes, language]*)
***context*.CheckType(*type_name, [includes, language]*)**

Checks for the existence of a type defined by typedef. *type_name* specifies the typedef name to check for. *includes* is a string containing one or more #include lines that will be inserted into the program that will be run to test for the existence of the type. Example:

```
sconf.CheckType('foo_type', '#include "my_types.h"', 'C++')
```

Returns an empty string on success, a string containing an error message on failure.

SConf.CheckCC(*context*)
***context*.CheckCC()**

Checks whether the C compiler (as defined by the CC construction variable) works by trying to compile a small source file. Returns a boolean indicating success or failure.

By default, SCons only detects if there is a program with the correct name, not if it is a functioning compiler.

This uses the exact same command as the one used by the object builder for C source files, so it can be used to detect if a particular compiler flag works or not.

SConf.CheckCXX(*context*)
***context*.CheckCXX()**

Checks whether the C++ compiler (as defined by the CXX construction variable) works by trying to compile a small source file. By default, SCons only detects if there is a program with the correct name, not if it is a functioning compiler. Returns a boolean indicating success or failure.

This uses the exact same command as the one used by the object builder for C++ source files, so it can be used to detect if a particular compiler flag works or not.

SConf.CheckSHCC(*context*)
***context*.CheckSHCC()**

Checks whether the shared-object C compiler (as defined by the SHCC construction variable) works by trying to compile a small source file. By default, SCons only detects if there is a program with the correct name, not if it is a functioning compiler. Returns a boolean indicating success or failure.

This uses the exact same command as the one used by the object builder for C source file, so it can be used to detect if a particular compiler flag works or not. This does not check whether the object code can be used to build a shared library, only that the compilation (not link) succeeds.

SConf.CheckSHCXX(*context*)
***context*.CheckSHCXX()**

Checks whether the shared-object C++ compiler (as defined by the SHCXX construction variable) works by trying to compile a small source file. By default, SCons only detects if there is a program with the correct name, not if it is a functioning compiler. Returns a boolean indicating success or failure.

This uses the exact same command as the one used by the object builder for C++ source files, so it can be used to detect if a particular compiler flag works or not. This does not check whether the object code can be used to build a shared library, only that the compilation (not link) succeeds.

SConf.CheckTypeSize(*context, type_name, [header, language, expect]*)
***context*.CheckTypeSize(*type_name, [header, language, expect]*)**

Checks for the size of a type defined by `typedef`. *type_name* specifies the typedef name to check for. The optional *header* argument is a string that will be placed at the top of the test file that will be compiled to check if the type exists; the default is empty. If the optional *expect*, is supplied, it should be an integer size; `CheckTypeSize` will fail unless *type_name* is actually that size. Returns the size in bytes, or zero if the type was not found (or if the size did not match *expect*).

For example,

```
CheckTypeSize('short', expect=2)
```

will return the size 2 only if short is actually two bytes.

SConf.CheckDeclaration(*context, symbol, [includes, language]*)
***context*.CheckDeclaration(*symbol, [includes, language]*)**

Checks if the specified *symbol* is declared. *includes* is a string containing one or more `#include` lines that will be inserted into the program that will be run to test for the existence of the symbol. Returns a boolean indicating success or failure.

SConf.Define(*context, symbol, [value, comment]*)
***context*.Define(*symbol, [value, comment]*)**

This function does not check for anything, but defines a preprocessor symbol that will be added to the configuration header file. It is the equivalent of AC_DEFINE, and defines the symbol *name* with the optional *value* and the optional comment *comment*.

Define Examples:

```
env = Environment()
conf = Configure(env)

# Puts the following line in the config header file:
#    #define A_SYMBOL
conf.Define("A_SYMBOL")

# Puts the following line in the config header file:
#    #define A_SYMBOL 1
conf.Define("A_SYMBOL", 1)
```

Be careful about quoting string values, though:

```
env = Environment()
conf = Configure(env)
```

```
# Puts the following line in the config header file:
#    #define A_SYMBOL YA
conf.Define("A_SYMBOL", "YA")

# Puts the following line in the config header file:
#    #define A_SYMBOL "YA"
conf.Define("A_SYMBOL", '"YA"')
```

For comment:

```

env = Environment()
conf = Configure(env)

# Puts the following lines in the config header file:
#    /* Set to 1 if you have a symbol */
#    #define A_SYMBOL 1
conf.Define("A_SYMBOL", 1, "Set to 1 if you have a symbol")
```

You can define your own custom checks in addition to the predefined checks. You pass a dictionary of these to the `Configure` function as the *`custom_tests`* argument. This dictionary maps the names of the checks to the user defined Python callables (either Python functions or class instances implementing a `__call__` method). Each custom check will be called with a first argument of a *CheckContext*, instance followed by the arguments, which must be supplied by the user of the check. A CheckContext instance defines the following methods:

***`context`*.`Message(`*`text`*`)`**

Displays a message, as an indicator of progess. *`text`* will be displayed, e.g. `Checking for library X...`. Usually called before the check is started.

***`context`*.`Result(`*`res`*`)`**

Displays a “result” message, as an indicator of progress. *`res`* can be either an integer or a string. If an integer, displays `yes` (if *`res`* evaluates `True`) or `no` (if *`res`* evaluates `False`). If a string, it is displayed as-is. Usually called after the check has completed.

***`context`*.`TryCompile(`*`text, extension=''`*`)`**

Checks if a file with the specified *`extension`* (e.g. '.c') containing *`text`* can be compiled using the environment's `Object` builder. Returns a boolean indicating success or failure.

***`context`*.`TryLink(`*`text, extension=''`*`)`**

Checks, if a file with the specified *`extension`* (e.g. '.c') containing *`text`* can be compiled using the environment's `Program` builder. Returns a boolean indicating success or failure.

***`context`*.`TryRun(`*`text, extension=''`*`)`**

Checks if a file with the specified *`extension`* (e.g. '.c') containing *`text`* can be compiled using the environment's `Program` builder. On success, the program is run. If the program executes successfully (that is, its return status is 0), a tuple *(1, outputStr)* is returned, where `outputStr` is the standard output of the program. If the program fails execution (its return status is non-zero), then *(0, '')* is returned.

***`context`*.`TryAction(`*`action, [text, extension='']`*`)`**

Checks if the specified *`action`* with an optional source file (contents *`text`*, extension *`extension`*) can be executed. *`action`* may be anything which can be converted to a **scons** Action. On success, *(1, outputStr)* is returned, where `outputStr` is the content of the target file. On failure *(0, '')* is returned.

***`context`*.`TryBuild(`*`builder[, text, extension='']`*`)`**

Low level implementation for testing specific builds; the methods above are based on this method. Given the Builder instance *`builder`* and the optional *`text`* of a source file with optional *`extension`*, returns a boolean indicating success or failure. In addition, `context.lastTarget` is set to the build target node if the build was successful.

Example of implementing and using custom tests:

```
def CheckQt(context, qtdir):
    context.Message( 'Checking for qt ...' )
    lastLIBS = context.env['LIBS']
    lastLIBPATH = context.env['LIBPATH']
    lastCPPPATH= context.env['CPPPATH']
    context.env.Append(LIBS='qt', LIBPATH=qtdir + '/lib', CPPPATH=qtdir + '/include')
    ret = context.TryLink("""
#include <qapp.h>
int main(int argc, char **argv) {
  QApplication qapp(argc, argv);
  return 0;
}
""")
    if not ret:
        context.env.Replace(LIBS=lastLIBS, LIBPATH=lastLIBPATH, CPPPATH=lastCPPPATH)
    context.Result( ret )
    return ret

env = Environment()
conf = Configure(env, custom_tests={'CheckQt': CheckQt})
if not conf.CheckQt('/usr/lib/qt'):
    print('We really need qt!')
    Exit(1)
env = conf.Finish()
```

Command-Line Construction Variables

Often when building software, some variables need to be specified at build time. For example, libraries needed for the build may be in non-standard locations, or site-specific compiler options may need to be passed to the compiler. SCons provides a *`Variables`* object to support overriding construction variables with values obtained from various sources, often from the command line:

```
scons VARIABLE=foo
```

The variable values can also be specified in a configuration file or an SConscript file.

To obtain the object for manipulating values, call the `Variables` function:

`Variables`([*`files, [args]]`*)

If *`files`* is a file or list of files, they are executed as Python scripts, and the values of (global) Python variables set in those files are added as construction variables in the Default Environment. If no files are specified, or the *`files`* argument is `None`, then no files will be read (supplying `None` is necessary if there are no files but you want to specify *`args`* as a positional argument).

The following example file contents could be used to set an alternative C compiler:

```
CC = 'my_cc'
```

If *args* is specified, it is a dictionary of values that will override anything read from *files*. The primary use is to pass the ARGUMENTS dictionary that holds variables specified on the command line, allowing you to indicate that if a setting appears on both the command line and in the file(s), the command line setting takes precedence. However, any dictionary can be passed. Examples:

```
vars = Variables('custom.py')
vars = Variables('overrides.py', ARGUMENTS)
vars = Variables(None, {FOO:'expansion', BAR:7})
```

Calling `Variables` with no arguments is equivalent to:

```
vars = Variables(files=None, args=ARGUMENTS)
```

Note that since the variables are eventually added as construction variables, you should choose variable names which do not unintentionally change pre-defined construction variables that your project will make use of (see the section called “Construction Variables”).

Variables objects have the following methods:

***vars*.Add(*key, [help, default, validator, converter]*)**

Add a customizable construction variable to the Variables object. *key* is either the name of the variable, or a tuple (or list), in which case the first item in the tuple is taken as the variable name, and any remaining values are considered aliases for the variable. *help* is the help text for the variable (default empty string). *default* is the default value of the variable (default `None`). If *default* is `None` and a value is not specified, the construction variable will not be added to the construction environment.

As a special case, if *key* is a tuple (or list) and is the *only* argument, the tuple is unpacked into the five parameters listed above left to right, with any missing members filled with the respecitive default values. This form allows `Add` to consume a tuple emitted by the convenience functions `BoolVariable`, `EnumVariable`, `ListVariable`, `PackageVariable` and `PathVariable`.

If the optional *validator* is supplied, it is called to validate the value of the variable. A function supplied as a validator must accept three arguments: *key*, *value* and *env*, and should raise an exception with a helpful error message if *value* is invalid. No return value is expected from the validator.

If the optional *converter* is supplied, it is called to convert the value before putting it in the environment, and should take either a value or a value and environment as parameters. The converter function must return a value, which will be converted into a string and be passed to the *validator* (if any) and then added to the construction environment.

Examples:

```
vars.Add('CC', help='The C compiler')

def valid_color(key, val, env):
    if not val in ['red', 'blue', 'yellow']:
        raise Exception("Invalid color value '%s'" % val)

vars.Add('COLOR', validator=valid_color)
```

***vars*.AddVariables(*args*)**

A convenience method that adds one or more customizable construction variables to a Variables object in one call; equivalent to calling `Add` multiple times. The *args* are tuples (or lists) that contain the arguments for an individual call to the `Add` method. Since tuples are not Python mappings, the arguments cannot use the keyword form, but rather are positional arguments as documented for `Add`: a required name, the other four optional, but must be in the specified order if used.

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

***vars*.Update(*env, [args]*)**

Update a construction environment *`env`* with the customized construction variables. Any specified variables that are not configured for the Variables object will be saved and may be retrieved using the `UnknownVariables` method.

Normally this method is not called directly, but rather invoked indirectly by passing the Variables object to the `Environment` function:

```
env = Environment(variables=vars)
```

***vars*.UnknownVariables()**

Returns a dictionary containing any variables that were specified either in the files or the dictionary with which the Variables object was initialized, but for which the Variables object was not configured.

```
env = Environment(variables=vars)
for key, value in vars.UnknownVariables():
    print("unknown variable:  %s=%s" % (key, value))
```

***vars*.Save(*filename, env*)**

Save the currently set variables into a script file named by *`filename`*. Only variables that are set to non-default values are saved. You can load these saved settings on a subsequent run by passing *`filename`* to the `Variables` function, providing a way to cache particular settings for reuse.

```
env = Environment()
vars = Variables(['variables.cache', 'custom.py'])
vars.Add(...)
vars.Update(env)
vars.Save('variables.cache', env)
```

***vars*.GenerateHelpText(*env, [sort]*)**

Generate help text documenting the customizable construction variables, suitable for passing in to the `Help` function. *`env`* is the construction environment that will be used to get the actual values of the customizable variables. If the (optional) value of *`sort`* is callable, it is used as a comparison function to determine how to sort the added variables. This function must accept two arguments, compare them, and return a negative integer if the first is less-than the second, zero for equality, or a positive integer for greater-than. Optionally a Boolean value of `True` for *`sort`* will cause a standard alphabetical sort to be performed.

```
Help(vars.GenerateHelpText(env))

def cmp(a, b):
    return (a > b) - (a < b)
```

```
Help(vars.GenerateHelpText(env, sort=cmp))
```

***vars*.FormatVariableHelpText(*env, opt, help, default, actual*)**

Returns a formatted string containing the printable help text for one option. It is normally not called directly, but is called by the `GenerateHelpText` method to create the returned help text. It may be overridden with your own function that takes the arguments specified above and returns a string of help text formatted to your liking. Note that `GenerateHelpText` will not put any blank lines or extra characters in between the entries, so you must add those characters to the returned string if you want the entries separated.

```
def my_format(env, opt, help, default, actual):
    fmt = "\n%s: default=%s actual=%s (%s)\n"
    return fmt % (opt, default, actual, help)

vars.FormatVariableHelpText = my_format
```

To make it more convenient to work with customizable Variables, **scons** provides a number of functions that make it easy to set up various types of Variables. Each of these return a tuple ready to be passed to the `Add` or `AddVariables` method:

BoolVariable(*key, help, default*)

Returns a tuple of arguments to set up a Boolean option. The option will use the specified name *key*, have a default value of *default*, and *help* will form the descriptive part of the help text. The option will interpret the values **y**, **yes**, **t**, **true**, **1**, **on** and **all** as true, and the values **n**, **no**, **f**, **false**, **0**, **off** and **none** as false.

EnumVariable(*key, help, default, allowed_values, [map, ignorecase]*)

Returns a tuple of arguments to set up an option whose value may be one of a specified list of legal enumerated values. The option will use the specified name *key*, have a default value of *default*, and *help* will form the descriptive part of the help text. The option will only support those values in the *allowed_values* list. The optional *map* argument is a dictionary that can be used to convert input values into specific legal values in the *allowed_values* list. If the value of *ignore_case* is `0` (the default), then the values are case-sensitive. If the value of *ignore_case* is `1`, then values will be matched case-insensitively. If the value of *ignore_case* is `2`, then values will be matched case-insensitively, and all input values will be converted to lower case.

ListVariable(*key, help, default, names, [map]*)

Returns a tuple of arguments to set up an option whose value may be one or more of a specified list of legal enumerated values. The option will use the specified name *key*, have a default value of *default*, and *help* will form the descriptive part of the help text. The option will only accept the values “all”, “none”, or the values in the *names* list. More than one value may be specified, separated by commas. The default may be a string of comma-separated default values, or a list of the default values. The optional *map* argument is a dictionary that can be used to convert input values into specific legal values in the *names* list. (Note that the additional values accepted through the use of a *map* are not reflected in the generated help message).

PackageVariable(*key, help, default*)

Returns a tuple of arguments to set up an option whose value is a path name of a package that may be enabled, disabled or given an explicit path name. The option will use the specified name *key*, have a default value of *default*, and *help* will form the descriptive part of the help text. The option will support the values **yes**, **true**, **on**, **enable** or **search**, in which case the specified *default* will be used, or the option may be set to an arbitrary string (typically the path name to a package that is being enabled). The option will also support the values **no**, **false**, **off** or **disable** to disable use of the specified option.

PathVariable(*key, help, default, [validator]*)

Returns a tuple of arguments to set up an option whose value is expected to be a path name. The option will use the specified name *key*, have a default value of *default*, and *help* will form the descriptive part of the help text. An additional `validator` may be specified that will be called to verify that the specified path is acceptable. SCons supplies the following ready-made validators:

`PathVariable.PathExists`
: Verify that the specified path exists (this the default behavior if no `validator` is supplied).

`PathVariable.PathIsFile`
: Verify that the specified path exists and is a regular file.

`PathVariable.PathIsDir`
: Verify that the specified path exists and is a directory.

`PathVariable.PathIsDirCreate`
: Verify that the specified path exists and is a directory; if it does not exist, create the directory.

`PathVariable.PathAccept`
: Accept the specific path name argument without validation, suitable for when you want your users to be able to specify a directory path that will be created as part of the build process, for example.

You may supply your own *validator* function, which must accept three arguments (`key`, the name of the variable to be set; `val`, the specified value being checked; and `env`, the construction environment) and should raise an exception if the specified value is not acceptable.

These functions make it convenient to create a number of variables with consistent behavior in a single call to the `AddVariables` method:

```
vars.AddVariables(
    BoolVariable(
        "warnings",
        help="compilation with -Wall and similar",
        default=1,
    ),
    EnumVariable(
        "debug",
        help="debug output and symbols",
        default="no",
        allowed_values=("yes", "no", "full"),
        map={},
        ignorecase=0,  # case sensitive
    ),
    ListVariable(
        "shared",
        help="libraries to build as shared libraries",
        default="all",
        names=list_of_libs,
    ),
    PackageVariable(
        "x11",
        help="use X11 installed here (yes = search some places)",
        default="yes",
    ),
    PathVariable(
        "qtdir",
        help="where the root of Qt is installed",
        default=qtdir),
    PathVariable(
        "foopath",
        help="where the foo library is installed",
        default=foopath,
        validator=PathVariable.PathIsDir,
    ),
)
```

File and Directory Nodes

The `File` and `Dir` functions/methods return File and Directory Nodes, respectively. Such nodes are Python objects with several user-visible attributes and methods that are often useful to access in SConscript files:

***n*.`path`**
: The build path of the given file or directory. This path is relative to the top-level directory (where the `SConstruct` file is found). The build path is the same as the source path if *variant_dir* is not being used.

***n*.`abspath`**
: The absolute build path of the given file or directory.

***n*.`relpath`**
: The build path of the given file or directory relative to the root SConstruct file's directory.

***n*.`srcnode`()**
: The `srcnode` method returns another File or Directory Node representing the source path of the given File or Directory Node.

For example:

```

# Get the current build dir's path, relative to top.
Dir('.').path
# Current dir's absolute path
Dir('.').abspath
# Current dir's path relative to the root SConstruct file's directory
Dir('.').relpath
# Next line is always '.', because it is the top dir's path relative to itself.
Dir('#.').path
File('foo.c').srcnode().path   # source path of the given source file.

# Builders also return File objects:
foo = env.Program('foo.c')
print("foo will be built in", foo.path)
```

File and Directory Node objects have methods to create File and Directory Nodes relative to the original Node.

If the object is a Directory Node, these methods will place the the new Node within the directory the Node represents:

***d*.`Dir`(*name*)**
: Returns a directory Node for a subdirectory of *d* named *name*.

***d*.`File`(*name*)**
: Returns a file Node for a file within *d* named *name*.

***d*.`Entry`(*name*)**
: Returns an unresolved Node within *d* named *name*.

If the object is a File Node, these methods will place the the new Node in the same directory as the one the Node represents:

***f*.Dir(*name*)**
Returns a directory named *`name`* within the parent directory of *`f`*.

***f*.File(*name*)**
Returns a file named *`name`* within the parent directory of *`f`*.

***f*.Entry(*name*)**
Returns an unresolved Node named *`name`* within the parent directory of *`f`*.

For example:

```
# Get a Node for a file within a directory
incl = Dir('include')
f = incl.File('header.h')

# Get a Node for a subdirectory within a directory
dist = Dir('project-3.2.1')
src = dist.Dir('src')

# Get a Node for a file in the same directory
cfile = File('sample.c')
hfile = cfile.File('sample.h')

# Combined example
docs = Dir('docs')
html = docs.Dir('html')
index = html.File('index.html')
css = index.File('app.css')
```

EXTENDING SCONS

Builder Objects

scons can be extended to build different types of targets by adding new Builder objects to a construction environment. *In general*, you should only need to add a new Builder object when you want to build a new type of file or other external target. For output file types **scons** already knows about, you can usually modify the behavior of premade Builders such as `Program`, `Object` or `Library` by changing the construction variables they use (`$CC`, `$LINK`, etc.). In this manner you can, for example, change the compiler to use, which is simpler and less error-prone than writing a new builder. The documentation for each Builder lists which construction variables it uses.

Builder objects are created using the `Builder` factory function. Once created, a builder is added to an environment by entering it in the `$BUILDERS` dictionary in that environment (some of the examples in this section illustrate that).

The `Builder` function accepts the following keyword arguments:

action
The command used to build the target from the source. *`action`* may be a string representing a template command line to execute, a list of strings representing the command to execute with its arguments (suitable for enclosing white space in an argument), a dictionary mapping source file name suffixes to any combination of command line strings (if the builder should accept multiple source file extensions), a Python function, an Action object (see the section called “Action Objects”) or a list of any of the above.

An action function must accept three arguments: *`source`*, *`target`* and *`env`*. *`source`* is a list of source nodes; *`target`* is a list of target nodes; *`env`* is the construction environment to use for context.

The *`action`* and *`generator`* arguments must not both be used for the same Builder.

`prefix`

The prefix to prepend to the target file name. *`prefix`* may be a string, a function (or other callable) that takes two arguments (a construction environment and a list of sources) and returns a prefix string, or a dictionary specifying a mapping from a specific source suffix (of the first source specified) to a corresponding target prefix string. For the dictionary form, both the source suffix (key) and target prefix (value) specifications may use environment variable substitution, and the target prefix may also be a callable object. The default target prefix may be indicated by a dictionary entry with a key of `None`.

```
b = Builder("build_it < $SOURCE > $TARGET",
            prefix="file-")

def gen_prefix(env, sources):
    return "file-" + env['PLATFORM'] + '-'

b = Builder("build_it < $SOURCE > $TARGET",
            prefix=gen_prefix)

b = Builder("build_it < $SOURCE > $TARGET",
            suffix={None: "file-", "$SRC_SFX_A": gen_prefix})
```

`suffix`

The suffix to append to the target file name. Specified in the same manner as for *`prefix`* above. If the suffix is a string, then **scons** prepends a `'.'` to the suffix if it's not already there. The string returned by the callable object or obtained from the dictionary is untouched and you need to manually prepend a `'.'` if one is required.

```
b = Builder("build_it < $SOURCE > $TARGET"
            suffix="-file")

def gen_suffix(env, sources):
    return "." + env['PLATFORM'] + "-file"

b = Builder("build_it < $SOURCE > $TARGET",
            suffix=gen_suffix)

b = Builder("build_it < $SOURCE > $TARGET",
            suffix={None: ".sfx1", "$SRC_SFX_A": gen_suffix})
```

`ensure_suffix`

If set to a true value, ensures that targets will end in *`suffix`*. Thus, the suffix will also be added to any target strings that have a suffix that is not already *`suffix`*. The default behavior (also indicated by a false value) is to leave unchanged any target string that looks like it already has a suffix.

```
b1 = Builder("build_it < $SOURCE > $TARGET"
             suffix = ".out")
b2 = Builder("build_it < $SOURCE > $TARGET"
             suffix = ".out",
             ensure_suffix=True)
```

```
env = Environment()
env['BUILDERS']['B1'] = b1
env['BUILDERS']['B2'] = b2

# Builds "foo.txt" because ensure_suffix is not set.
env.B1('foo.txt', 'foo.in')

# Builds "bar.txt.out" because ensure_suffix is set.
env.B2('bar.txt', 'bar.in')
```

src_suffix

The expected source file name suffix. *src_suffix* may be a string or a list of strings.

target_scanner

A Scanner object that will be invoked to find implicit dependencies for this target file. This keyword argument should be used for Scanner objects that find implicit dependencies based only on the target file and the construction environment, *not* for implicit dependencies based on source files. See the section called “Scanner Objects” for information about creating Scanner objects.

source_scanner

A Scanner object that will be invoked to find implicit dependencies in any source files used to build this target file. This is where you would specify a scanner to find things like `#include` lines in source files. The pre-built `DirScanner` Scanner object may be used to indicate that this Builder should scan directory trees for on-disk changes to files that **scons** does not know about from other Builder or function calls. See the section called “Scanner Objects” for information about creating your own Scanner objects.

target_factory

A factory function that the Builder will use to turn any targets specified as strings into SCons Nodes. By default, SCons assumes that all targets are files. Other useful target_factory values include **Dir**, for when a Builder creates a directory target, and **Entry**, for when a Builder can create either a file or directory target.

Example:

```
MakeDirectoryBuilder = Builder(action=my_mkdir, target_factory=Dir)
env = Environment()
env.Append(BUILDERS={'MakeDirectory': MakeDirectoryBuilder})
env.MakeDirectory('new_directory', [])
```

Note that the call to this `MakeDirectory` Builder needs to specify an empty source list to make the string represent the builder's target; without that, it would assume the argument is the source, and would try to deduce the target name from it, which in the absence of an automatically-added prefix or suffix would lead to a matching target and source name and a circular dependency.

source_factory

A factory function that the Builder will use to turn any sources specified as strings into SCons Nodes. By default, SCons assumes that all source are files. Other useful source_factory values include **Dir**, for when a Builder uses a directory as a source, and **Entry**, for when a Builder can use files or directories (or both) as sources.

Example:

```
CollectBuilder = Builder(action=my_mkdir, source_factory=Entry)
env = Environment()
env.Append(BUILDERS={'Collect': CollectBuilder})
env.Collect('archive', ['directory_name', 'file_name'])
```

emitter

A function or list of functions to manipulate the target and source lists before dependencies are established and the target(s) are actually built. *emitter* can also be a string containing a construction variable to expand to an emitter function or list of functions, or a dictionary mapping source file suffixes to emitter functions. (Only the suffix of the first source file is used to select the actual emitter function from an emitter dictionary.)

A function passed as *emitter* must accept three arguments: *source*, *target* and *env*. *source* is a list of source nodes, *target* is a list of target nodes, *env* is the construction environment to use for context.

An emitter must return a tuple containing two lists, the list of targets to be built by this builder, and the list of sources for this builder.

Example:

```
def e(target, source, env):
    return (target + ['foo.foo'], source + ['foo.src'])

# Simple association of an emitter function with a Builder.
b = Builder("my_build < $TARGET > $SOURCE",
            emitter = e)

def e2(target, source, env):
    return (target + ['bar.foo'], source + ['bar.src'])

# Simple association of a list of emitter functions with a Builder.
b = Builder("my_build < $TARGET > $SOURCE",
            emitter = [e, e2])

# Calling an emitter function through a construction variable.
env = Environment(MY_EMITTER=e)
b = Builder("my_build < $TARGET > $SOURCE",
            emitter='$MY_EMITTER')

# Calling a list of emitter functions through a construction variable.
env = Environment(EMITTER_LIST=[e, e2])
b = Builder("my_build < $TARGET > $SOURCE",
            emitter='$EMITTER_LIST')

# Associating multiple emitters with different file
# suffixes using a dictionary.
def e_suf1(target, source, env):
    return (target + ['another_target_file'], source)
def e_suf2(target, source, env):
    return (target, source + ['another_source_file'])
b = Builder("my_build < $TARGET > $SOURCE",
            emitter={'.suf1' : e_suf1,
                     '.suf2' : e_suf2})
```

multi

Specifies whether this builder is allowed to be called multiple times for the same target file(s). The default is `False`, which means the builder can not be called multiple times for the same target file(s). Calling a builder multiple times for the same target simply adds additional source files to the target; it is not allowed to change the environment associated with the target, specify additional environment overrides, or associate a different builder with the target.

env

A construction environment that can be used to fetch source code using this Builder. (Note that this environment is *not* used for normal builds of normal target files, which use the environment that was used to call the Builder for the target file.)

generator

A function that returns a list of actions that will be executed to build the target(s) from the source(s). The returned action(s) may be an Action object, or anything that can be converted into an Action object (see the next section).

A function passed as *generator* must accept four arguments: *source*, *target*, *env* and *for_signature*. *source* is a list of source nodes, *target* is a list of target nodes, *env* is the construction environment to use for context, *for_signature* is a Boolean value that specifies whether the generator is being called for generating a build signature (as opposed to actually executing the command).

Example:

```
def g(source, target, env, for_signature):
    return [["gcc", "-c", "-o"] + target + source]

b = Builder(generator=g)
```

The *generator* and *action* arguments must not both be used for the same Builder.

src_builder

Specifies a builder to use when a source file name suffix does not match any of the suffixes of the builder. Using this argument produces a multi-stage builder.

single_source

Specifies that this builder expects exactly one source file per call. Giving more than one source file without target files results in implicitly calling the builder multiple times (once for each source given). Giving multiple source files together with target files results in a `UserError` exception.

source_ext_match

When the specified *action* argument is a dictionary, the default behavior when a builder is passed multiple source files is to make sure that the extensions of all the source files match. If it is legal for this builder to be called with a list of source files with different extensions, this check can be suppressed by setting *source_ext_match* to `False` or some other non-true value. In this case, **scons** will use the suffix of the first specified source file to select the appropriate action from the *action* dictionary.

In the following example, the setting of *source_ext_match* prevents **scons** from exiting with an error due to the mismatched suffixes of `foo.in` and `foo.extra`.

```
b = Builder(action={'.in' : 'build $SOURCES > $TARGET'},
            source_ext_match=False)

env = Environment(BUILDERS={'MyBuild':b})
env.MyBuild('foo.out', ['foo.in', 'foo.extra'])
```

env

A construction environment that can be used to fetch source code using this Builder. (Note that this environment is *not* used for normal builds of normal target files, which use the environment that was used to call the Builder for the target file.)

```
b = Builder(action="build < $SOURCE > $TARGET")
env = Environment(BUILDERS={'MyBuild' : b})
env.MyBuild('foo.out', 'foo.in', my_arg='xyzzy')
```

chdir

A directory from which scons will execute the action(s) specified for this Builder. If the *chdir* argument is a string or a directory Node, scons will change to the specified directory. If the *chdir* is not a string or Node and is non-zero, then scons will change to the target file's directory.

Note that scons will *not* automatically modify its expansion of construction variables like `$TARGET` and `$SOURCE` when using the *chdir* keyword argument--that is, the expanded file names will still be relative to the top-level directory containing the `SConstruct` file, and consequently incorrect relative to the chdir directory. Builders created using *chdir* keyword argument, will need to use construction variable expansions like `${TARGET.file}` and `${SOURCE.file}` to use just the filename portion of the targets and source.

```

b = Builder(action="build < ${SOURCE.file} > ${TARGET.file}",
            chdir=1)
env = Environment(BUILDERS={'MyBuild' : b})
env.MyBuild('sub/dir/foo.out', 'sub/dir/foo.in')
```

Warning

Python only keeps one current directory location even if there are multiple threads. This means that use of the *chdir* argument will *not* work with the SCons `-j` option, because individual worker threads spawned by SCons interfere with each other when they start changing directory.

Any additional keyword arguments supplied when a Builder object is created (that is, when the `Builder` function is called) will be set in the executing construction environment when the Builder object is called. The canonical example here would be to set a construction variable to the repository of a source code system.

Any additional keyword arguments supplied when a Builder object is called will only be associated with the target created by that particular `Builder` call (and any other files built as a result of the call).

These extra keyword arguments are passed to the following functions: command generator functions, function Actions, and emitter functions.

Action Objects

The `Builder` factory function will turn its *action* keyword argument into an appropriate internal Action object, as will the `Command` function. You can also explicitly create Action objects for passing to `Builder`, or other functions that take actions as arguments, by calling the `Action` factory function. This may more efficient when multiple Builder objects need to do the same thing rather than letting each of those Builder objects create a separate Action object. It also allows more flexible configuration of an Action object. For example, to control the message printed when the action is taken you need to create the action object using `Action`.

The `Action` factory function returns an appropriate object for the action represented by the type of the *action* argument (the first positional parmeter):

- If *action* is already an Action object, the object is simply returned.
- If *action* is a string, a command-line Action is returned. If such a string begins with `@`, the command line is not printed. If the string begins with hyphen (`-`), the exit status from the specified command is ignored, allowing execution to continue even if the command reports failure:

```
Action('$CC -c -o $TARGET $SOURCES')

# Doesn't print the line being executed.
Action('@build $TARGET $SOURCES')

# Ignores return value
Action('-build $TARGET $SOURCES')
```

- If *action* is a list, then a list of Action objects is returned. An Action object is created as necessary for each element in the list. If an element within the list is itself a list, the embedded list is taken as the command and arguments to be executed via the command line. This allows white space to be enclosed in an argument rather than taken as a separator by defining a command in a list within a list:

```
Action([['cc', '-c', '-DWHITE SPACE', '-o', '$TARGET', '$SOURCES']])
```

- If *action* is a callable object, a Function Action is returned. The callable must accept three keyword arguments: *target*, *source* and *env*. *target* is a Node object representing the target file, *source* is a Node object representing the source file and *env* is the construction environment used for building the target file.

  The *target* and *source* arguments may be lists of Node objects if there is more than one target file or source file. The actual target and source file name(s) may be retrieved from their Node objects via the built-in Python `str` function:

```
target_file_name = str(target)
source_file_names = [str(x) for x in source]
```

  The function should return `0` or `None` to indicate a successful build of the target file(s). The function may raise an exception or return a non-zero exit status to indicate an unsuccessful build.

```
def build_it(target=None, source=None, env=None):
    # build the target from the source
    return 0

a = Action(build_it)
```

- If *action* is not one of the above types, no action object is generated and `Action` returns `None`.

The environment method form `env.Action` will expand construction variables in any argument strings, including *action*, at the time it is called, using the construction variables in the construction environment through which it was called. The global function form `Action` delays variable expansion until the Action object is actually used.

The optional second argument to `Action` is used to control the output which is printed when the Action is actually performed. If this parameter is omitted, or if the value is an empty string, a default output depending on the type of the action is used. For example, a command-line action will print the executed command. The following argument types are accepted:

- If *output* is a string, substitution is performed on the string before it is printed. The string typically contains variables, notably `$TARGET(S)` and `$SOURCE(S)`, or consists of just a single variable, which is optionally defined somewhere else. SCons itself heavily uses the latter variant.

- If *output* is a function, the function will be called to obtain a string describing the action being executed. The function must accept three keyword arguments: *target*, *source* and *env*, with the same interpretation as for a callable *action* argument above.

- If *`output`*is `None`, output is suppressed entirely.

Instead of using a positional argument, the *`cmdstr`* keyword argument may be used to specify the output string, or the *`strfunction`* keyword argument may be used to specify a function to return the output string. `cmdstr=None` suppresses output entirely.

Examples:

```
def build_it(target, source, env):
    # build the target from the source
    return 0

def string_it(target, source, env):
    return "building '%s' from '%s'" % (target[0], source[0])

# Use a positional argument.
f = Action(build_it, string_it)
s = Action(build_it, "building '$TARGET' from '$SOURCE'")

# Alternatively, use a keyword argument.
f = Action(build_it, strfunction=string_it)
s = Action(build_it, cmdstr="building '$TARGET' from '$SOURCE'")

# You can provide a configurable variable.
l = Action(build_it, '$STRINGIT')
```

Any additional positional arguments, if present, may either be construction variables or lists of construction variables whose values will be included in the signature of the Action when deciding whether a target should be rebuilt because the action changed. Such variables may also be specified using the *`varlist`* keyword parameter; both positional and keyword forms may be present, and will be combined. This is necessary whenever you want a target to be rebuilt when a specific construction variable changes. This is not often needed for a string action, as the expanded variables will normally be part of the command line, but may be needed if a Python function action uses the value of a construction variable when generating the command line.

```
def build_it(target, source, env):
    # build the target from the 'XXX' construction variable
    with open(target[0], 'w') as f:
        f.write(env['XXX'])
    return 0

# Use positional arguments.
a = Action(build_it, '$STRINGIT', ['XXX'])

# Alternatively, use a keyword argument.
a = Action(build_it, varlist=['XXX'])
```

The `Action` factory function can be passed the following optional keyword arguments to modify the Action object's behavior:

`chdir`

If *`chdir`* is true (the default is `False`), SCons will change directories before executing the action. If the value of *`chdir`* is a string or a directory Node, SCons will change to the specified directory. Otherwise, if *`chdir`* evaluates true, SCons will change to the target file's directory.

Note that SCons will *not* automatically modify its expansion of construction variables like `$TARGET` and `$SOURCE` when using the *`chdir`* parameter - that is, the expanded file names will still be relative to the top-level directory containing the `SConstruct` file, and consequently incorrect relative to the chdir directory. Builders created using *`chdir`* keyword argument, will need to use construction variable expansions like `${TARGET.file}` and `${SOURCE.file}` to use just the filename portion of the targets and source. Example:

```
a = Action("build < ${SOURCE.file} > ${TARGET.file}", chdir=True)
```

`exitstatfunc`

If provided, must be a callable which accepts a single parameter, the exit status (or return value) from the specified action, and which returns an arbitrary or modified value. This can be used, for example, to specify that an Action object's return value should be ignored under special conditions and SCons should, therefore, consider that the action always succeeds. Example:

```
def always_succeed(s):
    # Always return 0, which indicates success.
    return 0

a = Action("build < ${SOURCE.file} > ${TARGET.file}",
           exitstatfunc=always_succeed)
```

`batch_key`

If provided, indicates that the Action can create multiple target files by processing multiple independent source files simultaneously. (The canonical example is "batch compilation" of multiple object files by passing multiple source files to a single invocation of a compiler such as Microsoft's Visual C / C++ compiler.) If the *`batch_key`* argument evaluates True and is not a callable object, the configured Action object will cause **scons** to collect all targets built with the Action object and configured with the same construction environment into single invocations of the Action object's command line or function. Command lines will typically want to use the `$CHANGED_SOURCES` construction variable (and possibly `$CHANGED_TARGETS` as well) to only pass to the command line those sources that have actually changed since their targets were built. Example:

```
a = Action('build $CHANGED_SOURCES', batch_key=True)
```

The *`batch_key`* argument may also be a callable function that returns a key that will be used to identify different "batches" of target files to be collected for batch building. A *`batch_key`* function must accept four parameters: *`action`*, *`env`*, *`target`* and *`source`*. The first parameter, *`action`*, is the active action object. The second parameter, *`env`*, is the construction environment configured for the target. The *`target`* and *`source`* parameters are the lists of targets and sources for the configured action.

The returned key should typically be a tuple of values derived from the arguments, using any appropriate logic to decide how multiple invocations should be batched. For example, a *`batch_key`* function may decide to return the value of a specific construction variable from *`env`* which will cause **scons** to batch-build targets with matching values of that construction variable, or perhaps return the Python `id`() of the entire construction environment, in which case **scons** will batch-build all targets configured with the same construction environment. Returning `None` indicates that the particular target should *not* be part of any batched build, but instead will be built by a separate invocation of action's command or function. Example:

```
def batch_key(action, env, target, source):
    tdir = target[0].dir
    if tdir.name == 'special':
        # Don't batch-build any target
        # in the special/ subdirectory.
        return None
    return (id(action), id(env), tdir)
a = Action('build $CHANGED_SOURCES', batch_key=batch_key)
```

Miscellaneous Action Functions

SCons supplies Action functions that arrange for various common file and directory manipulations to be performed. These are similar in concept to "tasks" in the Ant build tool, although the implementation is slightly different. These functions do not actually perform the specified action at the time the function is called, but rather are factory functions which return an Action object that can be executed at the appropriate time.

There are two natural ways that these Action Functions are intended to be used.

First, if you need to perform the action at the time the SConscript file is being read, you can use the `Execute` global function:

```
Execute(Touch('file'))
```

Second, you can use these functions to supply Actions in a list for use by the `env.Command` method. This can allow you to perform more complicated sequences of file manipulation without relying on platform-specific external commands:

```
env = Environment(TMPBUILD='/tmp/builddir')
env.Command(
    target='foo.out',
    source='foo.in',
    action=[
        Mkdir('$TMPBUILD'),
        Copy('$TMPBUILD', '${SOURCE.dir}'),
        "cd $TMPBUILD && make",
        Delete('$TMPBUILD'),
    ],
)
```

Chmod(*dest, mode*)

Returns an Action object that changes the permissions on the specified *dest* file or directory to the specified *mode* which can be octal or string, similar to the bash command. Examples:

```
Execute(Chmod('file', 0o755))

env.Command('foo.out', 'foo.in',
            [Copy('$TARGET', '$SOURCE'),
             Chmod('$TARGET', 0o755)])

Execute(Chmod('file', "ugo+w"))

env.Command('foo.out', 'foo.in',
            [Copy('$TARGET', '$SOURCE'),
             Chmod('$TARGET', "ugo+w")])
```

The behavior of `Chmod` is limited on Windows, see the notes in the Python documentation for `os.chmod`, which is the underlying function.

Copy(*dest, src*)

Returns an Action object that will copy the *src* source file or directory to the *dest* destination file or directory. Examples:

```
Execute(Copy('foo.output', 'foo.input'))

env.Command('bar.out', 'bar.in', Copy('$TARGET', '$SOURCE'))
```

Delete(*entry, [must_exist]*)

Returns an Action that deletes the specified *entry*, which may be a file or a directory tree. If a directory is specified, the entire directory tree will be removed. If the *must_exist* flag is set to a true value, then a Python error will be raised if the specified entry does not exist; the default is false, that is, the Action will silently do nothing if the entry does not exist. Examples:

```
Execute(Delete('/tmp/buildroot'))

env.Command(
    'foo.out',
    'foo.in',
    action=[
        Delete('${TARGET.dir}'),
        MyBuildAction,
    ],
)

Execute(Delete('file_that_must_exist', must_exist=True))
```

Mkdir(*name*)

Returns an Action that creates the directory *name* and all needed intermediate directories. *name* may also be a list of directories to create. Examples:

```
Execute(Mkdir('/tmp/outputdir'))

env.Command(
    'foo.out',
    'foo.in',
    action=[
        Mkdir('/tmp/builddir'),
        Copy('/tmp/builddir/foo.in', '$SOURCE'),
        "cd /tmp/builddir && make",
        Copy('$TARGET', '/tmp/builddir/foo.out'),
    ],
)
```

Move(*dest, src*)

Returns an Action that moves the specified *src* file or directory to the specified *dest* file or directory. Examples:

```
Execute(Move('file.destination', 'file.source'))

env.Command(
    'output_file',
    'input_file',
    action=[MyBuildAction, Move('$TARGET', 'file_created_by_MyBuildAction')],
)
```

`Touch(`*`file`*`)`

Returns an Action that updates the modification time on the specified *`file`*. Examples:

```

Execute(Touch('file_to_be_touched'))

env.Command('marker', 'input_file', action=[MyBuildAction, Touch('$TARGET')])
```

Variable Substitution

Before executing a command, **scons** performs variable substitution on the string that makes up the action part of the builder. Variables to be interpolated are indicated in the string with a leading `$`, to distinguish them from plain text which is not to be substituted. The name may be surrounded by curly braces (`${}`) to separate the name from surrounding characters if necessary. Curly braces are required when you use Python list subscripting/slicing notation on a variable to select one or more items from a list, or access a variable's special attributes, or use Python expression substitution.

Besides regular construction variables, scons provides the following special variables for use in expanding commands:

`$CHANGED_SOURCES`

The file names of all sources of the build command that have changed since the target was last built.

`$CHANGED_TARGETS`

The file names of all targets that would be built from sources that have changed since the target was last built.

`$SOURCE`

The file name of the source of the build command, or the file name of the first source if multiple sources are being built.

`$SOURCES`

The file names of the sources of the build command.

`$TARGET`

The file name of the target being built, or the file name of the first target if multiple targets are being built.

`$TARGETS`

The file names of all targets being built.

`$UNCHANGED_SOURCES`

The file names of all sources of the build command that have *not* changed since the target was last built.

`$UNCHANGED_TARGETS`

The file names of all targets that would be built from sources that have *not* changed since the target was last built.

These names are reserved and may not be assigned to or used as construction variables.

For example, the following builder call:

```

env = Environment(CC='cc')
env.Command(
    target=['foo'],
    source=['foo.c', 'bar.c'],
    action='@echo $CC -c -o $TARGET $SOURCES'
)
```

would produce the following output:

```
cc -c -o foo foo.c bar.c
```

In the previous example, a string `${SOURCES[1]}` would expand to: `bar.c`.

A variable name may have the following modifiers appended within the enclosing curly braces to access properties of the interpolated string. These are known as *special attributes*.

`base` - The base path of the file name, including the directory path but excluding any suffix.
`dir` - The name of the directory in which the file exists.
`file` - The file name, minus any directory portion.
`filebase` - Like *`file`* but minus its suffix.
`suffix` - Just the file suffix.
`abspath` - The absolute path name of the file.
`relpath` - The path name of the file relative to the root SConstruct file's directory.
`posix` - The path with directories separated by forward slashes (/). Sometimes necessary on Windows systems when a path references a file on other (POSIX) systems.
`windows` - The path with directories separated by backslashes (`\\`). Sometimes necessary on POSIX-style systems when a path references a file on other (Windows) systems. *`win32`* is a (deprecated) synonym for *`windows`*.
`srcpath` - The directory and file name to the source file linked to this file through `VariantDir`(). If this file isn't linked, it just returns the directory and filename unchanged.
`srcdir` - The directory containing the source file linked to this file through `VariantDir`(). If this file isn't linked, it just returns the directory part of the filename.
`rsrcpath` - The directory and file name to the source file linked to this file through `VariantDir`(). If the file does not exist locally but exists in a Repository, the path in the Repository is returned. If this file isn't linked, it just returns the directory and filename unchanged.
`rsrcdir` - The Repository directory containing the source file linked to this file through `VariantDir`(). If this file isn't linked, it just returns the directory part of the filename.

For example, the specified target will expand as follows for the corresponding modifiers:

```
$TARGET              => sub/dir/file.x
${TARGET.base}       => sub/dir/file
${TARGET.dir}        => sub/dir
${TARGET.file}       => file.x
${TARGET.filebase}   => file
${TARGET.suffix}     => .x
${TARGET.abspath}    => /top/dir/sub/dir/file.x
${TARGET.relpath}    => sub/dir/file.x

$TARGET              => ../dir2/file.x
${TARGET.abspath}    => /top/dir2/file.x
${TARGET.relpath}    => ../dir2/file.x

SConscript('src/SConscript', variant_dir='sub/dir')
$SOURCE              => sub/dir/file.x
${SOURCE.srcpath}    => src/file.x
${SOURCE.srcdir}     => src

Repository('/usr/repository')
```

```
$SOURCE              => sub/dir/file.x
${SOURCE.rsrcpath}   => /usr/repository/src/file.x
${SOURCE.rsrcdir}    => /usr/repository/src
```

Some modifiers can be combined, like `${TARGET.srcpath.base)`, `${TARGET.file.suffix}`, etc.

The curly brace notation may also be used to enclose a Python expression to be evaluated. See the section called “Python Code Substitution” below for a description.

A variable name may also be a Python function associated with a construction variable in the environment. The function should accept four arguments:

`target` - a list of target nodes
`source` - a list of source nodes
`env` - the construction environment
`for_signature` - a Boolean value that specifies whether the function is being called for generating a build signature.

SCons will insert whatever the called function returns into the expanded string:

```
def foo(target, source, env, for_signature):
    return "bar"

# Will expand $BAR to "bar baz"
env=Environment(FOO=foo, BAR="$FOO baz")
```

As a reminder, this evaluation happens when `$BAR` is actually used in a builder action. The value of `env['BAR']` will be exactly as it was set: `"$FOO baz"`.

You can use this feature to pass arguments to a Python function by creating a callable class that stores one or more arguments in an object, and then uses them when the `__call__()` method is called. Note that in this case, the entire variable expansion must be enclosed by curly braces so that the arguments will be associated with the instantiation of the class:

```
class foo:
    def __init__(self, arg):
        self.arg = arg

    def __call__(self, target, source, env, for_signature):
        return self.arg + " bar"

# Will expand $BAR to "my argument bar baz"
env=Environment(FOO=foo, BAR="${FOO('my argument')} baz")
```

The special pseudo-variables **$(** and **$)** may be used to surround parts of a command line that may change *without* causing a rebuild--that is, which are not included in the signature of target files built with this command. All text between **$(** and **$)** will be removed from the command line before it is added to file signatures, and the **$(** and **$)** will be removed before the command is executed. For example, the command line:

```
echo Last build occurred $( $TODAY $). > $TARGET
```

would execute the command:

```
echo Last build occurred $TODAY. > $TARGET
```

but the command signature added to any target files would be:

```
echo Last build occurred  . > $TARGET
```

Python Code Substitution

If a substitutable expression using the notation `${something}` does not appear to match one of the other substitution patterns, it is evaluated as a Python expression. This uses Python's `eval` function, with the *`globals`* parameter set to the current environment's set of construction variables, and the result substituted in. So in the following case:

```
env.Command(
    'foo.out', 'foo.in', "echo ${COND==1 and 'FOO' or 'BAR'} > $TARGET"
)
```

the command executed will be either

```
echo FOO > foo.out
```

or

```
echo BAR > foo.out
```

according to the current value of `env['COND']` when the command is executed. The evaluation takes place when the target is being built, not when the SConscript is being read. So if `env['COND']` is changed later in the SConscript, the final value will be used.

Here's a more complete example. Note that all of `COND`, `FOO`, and `BAR` are construction variables, and their values are substituted into the final command. `FOO` is a list, so its elements are interpolated separated by spaces.

```
env=Environment()
env['COND'] = 1
env['FOO'] = ['foo1', 'foo2']
env['BAR'] = 'barbar'
env.Command(
    'foo.out', 'foo.in', "echo ${COND==1 and FOO or BAR} > $TARGET"
)
```

will execute:

```
echo foo1 foo2 > foo.out
```

In point of fact, Python expression evaluation is how the special attributes are substituted: they are simply attributes of the Python objects that represent `$TARGET`, `$SOURCES`, etc., which SCons passes to `eval` which returns the value.

SCons uses the following rules when converting construction variables into command lines:

string
: When the value is a string it is interpreted as a space delimited list of command line arguments.

list
: When the value is a list it is interpreted as a list of command line arguments. Each element of the list is converted to a string.

other

Anything that is not a list or string is converted to a string and interpreted as a single command line argument.

newline

Newline characters (`\n`) delimit lines. The newline parsing is done after all other parsing, so it is not possible for arguments (e.g. file names) to contain embedded newline characters.

Note

Use of the Python `eval` function is considered to have security implications, since, depending on input sources, arbitrary unchecked strings of code can be executed by the Python interpreter. Although SCons makes use of it in a somewhat restricted context, you should be aware of this issue when using the `${python-expression-for-subst}` form.

Scanner Objects

Scanner objects are used to scan specific file types for implicit dependencies. SCons has a number of pre-built Scanner objects, so it is usually only necessary to set up Scanners for new file types. You do this by calling the `Scanner` function. The `Scanner` function accepts the following arguments, only *`function`* is required, the rest are optional:

`function`

A Python function that will process a given Node (usually a file) and return a list of Nodes representing the implicit dependencies (file names) found in the contents. The function must accept three required arguments, *`node`*, *`env`* and *`path`*, and an optional fourth, *`arg`*. *`node`* is the internal SCons node representing the file to scan, *`env`* is the construction environment to use during the scan, and *`path`* is a tuple of directories that can be searched for files, as generated by the optional *`path_function`* (see below). If *`argument`* was supplied when the Scanner object was created, it is given as *`arg`* when the function is called; since *`argument`* is optional, the default is no *`arg`*.

The function can use use `str`(*`node`*) to fetch the name of the file, and *`node`*.`get_contents`() to fetch the contents of the file as bytes or *`node`*.`get_text_contents`() to fetch the file's contents as text. Note that the file is *not* guaranteed to exist at the time the scanner is called (it could be a generated file, not generated yet), so the scanner function must be tolerant of that.

`name`

The name to use for the Scanner. This is mainly used to identify the Scanner internally. The default value is `"NONE"`.

`argument`

If specified, will be passed to the scanner function *`function`* and the path function *`path_function`* when called, as the additional argument each of those functions takes.

`skeys`

Scanner key(s) indicating the file types this scanner is associated with. Used internally to select an appropriate scanner. In the usual case of scanning for file names, this argument will be a list of suffixes for the different file types that this Scanner knows how to scan. If *`skeys`* is a string, it will be expanded into a list by the current environment.

`path_function`

A Python function that takes four or five arguments: a construction environment, a Node for the directory containing the SConscript file in which the first target was defined, a list of target nodes, a list of source nodes, and the value of *`argument`* if it was supplied when the scanner was created. Must return a tuple of directories that can be searched for files to be returned by this Scanner object. (Note that the `FindPathDirs` function can be used to return a ready-made *`path_function`* for a given construction variable name, instead of having to write your own function from scratch.)

`node_class`
: The class of Node that should be returned by this Scanner object. Any strings or other objects returned by the scanner function that are not of this class will be run through the function supplied by the *`node_factory`* argument. A value of `None` can be supplied to indicate no conversion; the default is to return File nodes.

`node_factory`
: A Python function that will take a string or other object and turn it into the appropriate class of Node to be returned by this Scanner object, as indicated by *`node_class`*.

`scan_check`
: A Python function that takes two arguments, a Node (file) and a construction environment, and returns whether the Node should, in fact, be scanned for dependencies. This check can be used to eliminate unnecessary calls to the scanner function when, for example, the underlying file represented by a Node does not yet exist.

`recursive`
: Specifies whether this scanner should be re-invoked on the dependency files returned by the scanner. If omitted, the Node subsystem will only invoke the scanner on the file being scanned and not recurse. Recursion is needed when the files returned by the scanner may themselves contain further file dependencies, as in the case of preprocessor `#include` lines. A value that evaluates true enables recursion; *recursive* may be a callable function, in which case it will be called with a list of Nodes found and should return a list of Nodes that should be scanned recursively; this can be used to select a specific subset of Nodes for additional scanning.

Note that **scons** has a global `SourceFileScanner` object that is used by the `Object`, `SharedObject` and `StaticObject` builders to decide which scanner should be used for different file extensions. You can use the `SourceFileScanner.add_scanner()` method to add your own Scanner object to the SCons infrastructure that builds target programs or libraries from a list of source files of different types:

```
def xyz_scan(node, env, path):
    contents = node.get_text_contents()
    # Scan the contents and return the included files.

XYZScanner = Scanner(xyz_scan)

SourceFileScanner.add_scanner('.xyz', XYZScanner)

env.Program('my_prog', ['file1.c', 'file2.f', 'file3.xyz'])
```

SYSTEM-SPECIFIC BEHAVIOR

scons and its configuration files are very portable, due largely to its implementation in Python. There are, however, a few portability issues waiting to trap the unwary.

.C file suffix

scons handles the upper-case `.C` file suffix differently, depending on the capabilities of the underlying system. On a case-sensitive system such as Linux or UNIX, **scons** treats a file with a `.C` suffix as a C++ source file. On a case-insensitive system such as Windows, **scons** treats a file with a `.C` suffix as a C source file.

Fortran file suffixes

scons handles upper-case Fortran file suffixes differently depending on the capabilities of the underlying system. On a case-sensitive system such as Linux or UNIX, **scons** treats a file with a `.F` as a Fortran source file that is to be first run through the standard C preprocessor, while the lower-case version is not. This matches the convention of

gfortran, which may also be followed by other Fortran compilers. This also applies to other naming variants, `.FOR`, `.FTN`, `.F90`, `.F95`, `.F03` and `.F08`; files suffixed with `.FPP` and `.fpp` are both run through the preprocessor, as indicated by the `pp` part of the name. On a case-insensitive system such as Windows, **scons** treats a file with a `.F` suffix as a Fortran source file that should *not* be run through the C preprocessor.

Run through the C preprocessor here means that a different set of construction variables will be applied in constructed commands, for example `$FORTRANPPCOM` and `$FORTRANPPCOMSTR` instead of `$FORTRANCOM` and `$FORTRANCOMSTR`. See the Fortran-related construction variables for more details.

Windows: Cygwin Tools and Cygwin Python vs. Windows Pythons

Cygwin supplies a set of tools and utilities that let users work on a Windows system using a more POSIX-like environment. The Cygwin tools, including Cygwin Python, do this, in part, by sharing an ability to interpret UNIX-like path names. For example, the Cygwin tools will internally translate a Cygwin path name like `/cygdrive/c/mydir` to an equivalent Windows pathname of `C:/mydir` (equivalent to `C:\mydir`).

Versions of Python that are built for native Windows execution, such as the python.org and ActiveState versions, do not have the Cygwin path name semantics. This means that using a native Windows version of Python to build compiled programs using Cygwin tools (such as gcc, bison and flex) may yield unpredictable results. "Mixing and matching" in this way can be made to work, but it requires careful attention to the use of path names in your SConscript files.

In practice, users can sidestep the issue by adopting the following rules: When using Cygwin's gcc for compiling, use the Cygwin-supplied Python interpreter to run **scons**; when using Microsoft Visual C/C++ (or some other Windows compiler) use the python.org or Microsoft Store or ActiveState version of Python to run **scons**.

Windows: `scons.bat` file

On Windows, if **scons** is executed via a wrapper `scons.bat` batch file, there are (at least) two ramifications. Note this is no longer the default - **scons** installed via Python's **pip** installer will have an **scons.exe** which does not have these limitations:

First, Windows command-line users that want to use variable assignment on the command line may have to put double quotes around the assignments, otherwise the Windows command shell will consume those as arguments to itself, not to **scons**:

```
scons "FOO=BAR" "BAZ=BLEH"
```

Second, the Cygwin shell does not reognize typing **`scons`** at the command line prompt as referring to this weapper. You can work around this either by executing **`scons.bat`** (including the extension) from the Cygwin command line, or by creating a wrapper shell script named `scons` which invokes `scons.bat`.

MinGW

The MinGW `bin` directory must be in your `PATH` environment variable or the `['ENV']['PATH']` construction variable for **scons** to detect and use the MinGW tools. When running under the native Windows Python interpreter, **scons** will prefer the MinGW tools over the Cygwin tools, if they are both installed, regardless of the order of the bin directories in the `PATH` variable. If you have both MSVC and MinGW installed and you want to use MinGW instead of MSVC, then you must explicitly tell **scons** to use MinGW by passing `tools=['mingw']` to the `Environment` function, because **scons** will prefer the MSVC tools over the MinGW tools.

ENVIRONMENT

In general, **scons** is not controlled by environment variables set in the shell used to invoke it, leaving it up to the SConscript file author to import those if desired. However the following variables are imported by **scons** itself if set:

SCONS_LIB_DIR

Specifies the directory that contains the **scons** Python module directory. Normally **scons** can deduce this, but in some circumstances, such as working with a source release, it may be necessary to specify (for example, `/home/aroach/scons-src-0.01/src/engine`).

SCONSFLAGS

A string containing options that will be used by **scons** in addition to those passed on the command line. Can be used to reduce frequent retyping of common options. The contents of `SCONSFLAGS` are considered before any passed command line options, so the command line can be used to override `SCONSFLAGS` options if necessary.

SCONS_CACHE_MSVC_CONFIG

(Windows only). If set, save the shell environment variables generated when setting up the Microsoft Visual C++ compiler (and/or Build Tools) to a cache file, to give these settings, which are relatively expensive to generate, persistence across **scons** invocations. Use of this option is primarily intended to aid performance in tightly controlled Continuous Integration setups.

If set to a True-like value (`"1"`, `"true"` or `"True"`) will cache to a file named `.scons_msvc_cache` in the user's home directory. If set to a pathname, will use that pathname for the cache.

Note: use this cache with caution as it might be somewhat fragile: while each major toolset version (e.g. Visual Studio 2017 vs 2019) and architecture pair will get separate cache entries, if toolset updates cause a change to settings within a given release series, **scons** will not detect the change and will reuse old settings. Remove the cache file in case of problems with this. **scons** will ignore failures reading or writing the file and will silently revert to non-cached behavior in such cases.

*Available since **scons** 3.1 (experimental).*

QTDIR

If using the `qt` tool, this is the path to the Qt installation to build against. SCons respects this setting because it is a long-standing convention in the Qt world, where multiple Qt installations are possible.

SEE ALSO

The SCons User Guide at https://scons.org/doc/production/HTML/scons-user.html
The SCons Design Document (old)
The SCons Cookbook at https://scons-cookbook.readthedocs.io for examples of how to solve various problems with SCons.
SCons source code on GitHub [https://github.com/SCons/scons]
The SCons API Reference https://scons.org/doc/production/HTML/scons-api/index.html (for internal details)

AUTHORS

Originally: Steven Knight `<knight@baldmt.com>` and Anthony Roach `<aroach@electriceyeball.com>`.

Since 2010: The SCons Development Team `<scons-dev@scons.org>`.