SCONS
Build your software, better.

SCons 4.3.0

User Guide

The SCons Development Team

Version 4.3.0

Publication date Released: Mon, 31 Jul 2021 20:43:54 -0700

Table of Contents

List of Examples

Preface

Thank you for taking the time to read about SCons. SCons is a modern software construction too - a software utility for building software (or other files) and keeping built software up-to-date whenever the underlying input files change.

The most distinctive thing about SCons is that its configuration files are actually *scripts*, written in the Python programming language. This is in contrast to most alternative build tools, which typically invent a new language to configure the build. SCons still has a learning curve, of course, because you have to know what functions to call to set up your build properly, but the underlying syntax used should be familiar to anyone who has ever looked at a Python script.

Paradoxically, using Python as the configuration file format makes SCons *easier* for non-programmers to learn than the cryptic languages of other build tools, which are usually invented by programmers for other programmers. This is in no small part due to the consistency and readability that are hallmarks of Python. It just so happens that making a real, live scripting language the basis for the configuration files makes it a snap for more accomplished programmers to do more complicated things with builds, as necessary.

1. SCons Principles

There are a few overriding principles the SCons team tries to follow in the design and implementation.

Correctness
: First and foremost, by default, SCons guarantees a correct build even if it means sacrificing performance a little. We strive to guarantee the build is correct regardless of how the software being built is structured, how it may have been written, or how unusual the tools are that build it.

Performance
: Given that the build is correct, we try to make SCons build software as quickly as possible. In particular, wherever we may have needed to slow down the default SCons behavior to guarantee a correct build, we also try to make it easy to speed up SCons through optimization options that let you trade off guaranteed correctness in all end cases for a speedier build in the usual cases.

Convenience
: SCons tries to do as much for you out of the box as reasonable, including detecting the right tools on your system and using them correctly to build the software.

In a nutshell, we try hard to make SCons just "do the right thing" and build software correctly, with a minimum of hassles.

2. How to Use this Guide

This guide intends to coach you how to use SCons effectively and efficiently, by providing a range of examples and usage scenarios. As such it is not exactly a tutorial (as usually those build a single example topic from start to finish), but if you are just starting with SCons it *is* recommended you step through the first 10 chapters in sequence as this will give a solid grounding in the principles of working with SCons. If you follow that trail, you can feel free to initially skip sections on extending SCons, such as *Writing your own Decider Function*, and come back to those if the need arises.

The remaining chapters cover more advanced topics that not all build systems will need, and can be used in more of a single-topic way, to read if you find you need that particular information.

It is often useful to keep SCons man page open in a separate browser tab or window to refer to as a complement to this Guide, as the User Guide does not attempt to provide every detail. While this Guide's Appendices A-D do duplicate

information that appears in the man page (this is to allow intra-document links to definitions of construction variables, builders, tools and environment methods to work), the rest of the man page is unique content.

3. A Caveat About This Guide's Completeness

SCons is a volunteer-run open source project. As such, the SCons documentation isn't always completely up-to-date with all the available features - somehow it's almost harder to write high quality, easy to use documentation than it is to implement a feature in software. In other words, there may be a lot that SCons can do that isn't yet covered in this User's Guide.

Although this User's Guide may not be as complete as it could be, the development process does emphasize making sure that the SCons man page is kept up-to-date with new features. So if you're trying to figure out how to do something that SCons supports but can't find enough (or any) information here, it would be worth your while to look at the man page to see if the information is covered there. And if you do, maybe you'd even consider contributing a section to the User's Guide so the next person looking for that information won't have to go through the same thing...?

4. Acknowledgements

SCons would not exist without a lot of help from a lot of people, many of whom may not even be aware that they helped or served as inspiration. So in no particular order, and at the risk of leaving out someone:

First and foremost, SCons owes a tremendous debt to Bob Sidebotham, the original author of the classic Perl-based Cons tool which Bob first released to the world back around 1996. Bob's work on Cons classic provided the underlying architecture and model of specifying a build configuration using a real scripting language. My real-world experience working on Cons informed many of the design decisions in SCons, including the improved parallel build support, making Builder objects easily definable by users, and separating the build engine from the wrapping interface.

Greg Wilson was instrumental in getting SCons started as a real project when he initiated the Software Carpentry design competition in February 2000. Without that nudge, marrying the advantages of the Cons classic architecture with the readability of Python might have just stayed no more than a nice idea.

The entire SCons team have been absolutely wonderful to work with, and SCons would be nowhere near as useful a tool without the energy, enthusiasm and time people have contributed over the past few years. The "core team" of Chad Austin, Anthony Roach, Bill Deegan, Charles Crain, Steve Leblanc, Greg Noel, Gary Oberbrunner, Greg Spencer and Christoph Wiedemann have been great about reviewing my (and other) changes and catching problems before they get in the code base. Of particular technical note: Anthony's outstanding and innovative work on the tasking engine has given SCons a vastly superior parallel build model; Charles has been the master of the crucial Node infrastructure; Christoph's work on the Configure infrastructure has added crucial Autoconf-like functionality; and Greg has provided excellent support for Microsoft Visual Studio.

Special thanks to David Snopek for contributing his underlying "Autoscons" code that formed the basis of Christoph's work with the Configure functionality. David was extremely generous in making this code available to SCons, given that he initially released it under the GPL and SCons is released under a less-restrictive MIT-style license.

Thanks to Peter Miller for his splendid change management system, Aegis, which has provided the SCons project with a robust development methodology from day one, and which showed me how you could integrate incremental regression tests into a practical development cycle (years before eXtreme Programming arrived on the scene).

And last, thanks to Guido van Rossum for his elegant scripting language, which is the basis not only for the SCons implementation, but for the interface itself.

5. Contact

The best way to contact people involved with SCons, is through the SCons mailing lists.

If you want to ask general questions about how to use SCons send email to <scons-users@scons.org>.

If you want to contact the SCons development community directly, send email to <scons-dev@scons.org>.

For quicker, informal questions, discussion, etc. the project operated a Discord server at https://discord.gg/bXVpWAy and a Libera.chat IRC channel at https://web.libera.chat/#scons (the former channel at irc.freenode.net is now unused). Certain discussions may also be moved by administrators from mailing list or chat to GitHub Discussions [https://github.com/SCons/scons/discussions] for greater permanence and easier finding.

1 Building and Installing SCons

This chapter will take you through the basic steps of installing SCons on your system, and building SCons if you don't have a pre-built package available (or simply prefer the flexibility of building it yourself). Before that, however, this chapter will also describe the basic steps involved in installing Python on your system, in case that is necessary. Fortunately, both SCons and Python are very easy to install on almost any system, and Python already comes installed on many systems.

1.1. Installing Python

Because SCons is written in Python, you need to have Python installed on your system to use SCons. Before you try to install Python, you should check to see if Python is already available on your system by typing **`python -V`** (capital 'V') or **`python --version`** at your system's command-line prompt. For Linux/Unix/MacOS/BSD type systems this looks like:

```
$ python -V
Python 3.7.1
```

Note to Windows users: there are a number of different ways Python can be installed or invoked on Windows, it is beyond the scope of this guide to unravel all of them. Many will have an additional program called the *Python launcher* (described, somewhat technically, in PEP 397 [https://www.python.org/dev/peps/pep-0397/]): try using the command name **py** instead of **python**, if that is not available drop back to trying **python**.

```
C:\>py -V
Python 3.7.1
```

If Python is not installed on your system, or is not findable in the current search path, you will see an error message stating something like `"command not found"` (on UNIX or Linux) or `"'python' is not recognized as an internal or external command, operable progam or batch file"` (on Windows **cmd**). In that case, you need to either install Python or fix the search path before you can install SCons.

The canonical location for downloading Python from Python's own website is: https://www.python.org/download. There are useful system-specific entries on setup and usage to be found at: https://docs.python.org/3/using

For Linux systems, Python is almost certainly available as a supported package, possibly installed by default; this is often preferred over installing by other means, and is easier than installing from source code. Many such systems have separate packages for Python 2 and Python 3 - make sure the Python 3 package is installed, as the latest SCons requires it. Building from source may still be a useful option if you need a version that is not offered by the distribution you are using.

SCons will work with Python 3.5 or later. If you need to install Python and have a choice, we recommend using the most recent Python version available. Newer Pythons have significant improvements that help speed up the performance of SCons.

1.2. Installing SCons

The canonical way to install SCons is from the Python Package Index (PyPi):

```
% python -m pip install scons
```

If you prefer not to install to the Python system location, or do not have privileges to do so, you can add a flag to install to a location specific to your own account:

```
% python -m pip install --user scons
```

For those users using Anaconda or Miniconda, use the **conda** installer instead, so the **scons** install location will match the version of Python that system will be using. For example:

```
% conda install -c conda-forge scons
```

SCons comes pre-packaged for installation on many Linux systems. Check your package installation system to see if there is an SCons package available. Many people prefer to install distribution-native packages if available, as they provide a central point for management and updating. During the still-ongoing Python 2 to 3 transition, some distributions may still have two SCons packages available, one which uses Python 2 and one which uses Python 3. Since the latest **scons** only runs on Python 3, to get the current version you should choose the Python 3 package.

If you need a specific version of SCons that is different from the package available, `pip` has a version option or you can follow the instructions in the next section.

1.3. Building and Installing SCons on Any System

If a pre-built SCons package is not available for your system, and installing using `pip` is not suitable, then you can still easily build and install SCons using the native Python `setuptools` package.

The first step is to download either the `scons-4.3.0.tar.gz` or `scons-4.3.0.zip`, which are available from the SCons download page at https://scons.org/pages/download.html.

Unpack the archive you downloaded, using a utility like tar on Linux or UNIX, or WinZip on Windows. This will create a directory called `scons-4.3.0`, usually in your local directory. Then change your working directory to that directory and install SCons by executing the following commands:

```
# cd scons-4.3.0
# python setup.py install
```

This will build SCons, install the **scons** script in the python which is used to run the setup.py's scripts directory (`/usr/local/bin` or `C:\Python37\Scripts`), and will install the SCons build engine in the corresponding library directory for the python used (`/usr/local/lib/scons` or `C:\Python37\scons`). Because these are system directories, you may need root (on Linux or UNIX) or Administrator (on Windows) privileges to install SCons like this.

1.3.1. Building and Installing Multiple Versions of SCons Side-by-Side

The SCons `setup.py` script has some extensions that support easy installation of multiple versions of SCons in side-by-side locations. This makes it easier to download and experiment with different versions of SCons before moving your official build process to a new version, for example.

To install SCons in a version-specific location, add the `--version-lib` option when you call `setup.py`:

```
# python setup.py install --version-lib
```

This will install the SCons build engine in the `/usr/lib/scons-4.3.0` or `C:\Python27\scons-4.3.0` directory, for example.

If you use the `--version-lib` option the first time you install SCons, you do not need to specify it each time you install a new version. The SCons `setup.py` script will detect the version-specific directory name(s) and assume you want to install all versions in version-specific directories. You can override that assumption in the future by explicitly specifying the `--standalone-lib` option.

1.3.2. Installing SCons in Other Locations

You can install SCons in locations other than the default by specifying the `--prefix=` option:

```
# python setup.py install --prefix=/opt/scons
```

This would install the scons script in `/opt/scons/bin` and the build engine in `/opt/scons/lib/scons`,

Note that you can specify both the `--prefix=` and the `--version-lib` options at the same type, in which case `setup.py` will install the build engine in a version-specific directory relative to the specified prefix. Adding `--version-lib` to the above example would install the build engine in `/opt/scons/lib/scons-4.3.0`.

1.3.3. Building and Installing SCons Without Administrative Privileges

If you don't have the right privileges to install SCons in a system location, simply use the `--prefix=` option to install it in a location of your choosing. For example, to install SCons in appropriate locations relative to the user's `$HOME` directory, the **scons** script in `$HOME/bin` and the build engine in `$HOME/lib/scons`, simply type:

```
$ python setup.py install --prefix=$HOME
```

You may, of course, specify any other location you prefer, and may use the `--version-lib` option if you would like to install version-specific directories relative to the specified prefix.

This can also be used to experiment with a newer version of SCons than the one installed in your system locations. Of course, the location in which you install the newer version of the **scons** script (`$HOME/bin` in the above example) must be configured in your `PATH` variable before the directory containing the system-installed version of the **scons** script.

2 Simple Builds

In this chapter, you will see several examples of very simple build configurations using SCons, which will demonstrate how easy it is to use SCons to build programs from several different programming languages on different types of systems.

2.1. Building Simple C / C++ Programs

Here's the famous "Hello, World!" program in C:

```
int
main()
{
    printf("Hello, world!\n");
}
```

And here's how to build it using SCons. Save the code above into `hello.c`, and enter the following into a file named `SConstruct`:

```
Program('hello.c')
```

This minimal configuration file gives SCons two pieces of information: what you want to build (an executable program), and the input file from which you want it built (the `hello.c` file). `Program` is a *builder method*, a Python call that tells SCons that you want to build an executable program.

That's it. Now run the **scons** command to build the program. On a POSIX-compliant system like Linux or UNIX, you'll see something like:

```
% scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cc -o hello.o -c hello.c
cc -o hello hello.o
scons: done building targets.
```

On a Windows system with the Microsoft Visual C++ compiler, you'll see something like:

```
C:\>scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cl /Fohello.obj /c hello.c /nologo
link /nologo /OUT:hello.exe hello.obj
embedManifestExeCheck(target, source, env)
scons: done building targets.
```

First, notice that you only need to specify the name of the source file, and that SCons correctly deduces the names of the object and executable files to be built from the base of the source file name.

Second, notice that the same input `SConstruct` file, without any changes, generates the correct output file names on both systems: `hello.o` and `hello` on POSIX systems, `hello.obj` and `hello.exe` on Windows systems. This is a simple example of how SCons makes it extremely easy to write portable software builds.

(Note that we won't provide duplicate side-by-side POSIX and Windows output for all of the examples in this guide; just keep in mind that, unless otherwise specified, any of the examples should work equally well on both types of systems.)

2.2. Building Object Files

The `Program` builder method is only one of many builder methods that SCons provides to build different types of files. Another is the `Object` builder method, which tells SCons to build an object file from the specified source file:

```
Object('hello.c')
```

Now when you run the **scons** command to build the program, it will build just the `hello.o` object file on a POSIX system:

```
% scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cc -o hello.o -c hello.c
scons: done building targets.
```

And just the `hello.obj` object file on a Windows system (with the Microsoft Visual C++ compiler):

```
C:\>scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cl /Fohello.obj /c hello.c /nologo
scons: done building targets.
```

2.3. Simple Java Builds

SCons also makes building with Java extremely easy. Unlike the `Program` and `Object` builder methods, however, the `Java` builder method requires that you specify the name of a destination directory in which you want the class files placed, followed by the source directory in which the `.java` files live:

```
```

```
Java('classes', 'src')
```

If the `src` directory contains a single `hello.java` file, then the output from running the **scons** command would look something like this (on a POSIX system):

```
% scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
javac -d classes -sourcepath src src/hello.java
scons: done building targets.
```

We'll cover Java builds in more detail, including building Java archive (`.jar`) and other types of file, in Chapter 24, *Java Builds*.

2.4. Cleaning Up After a Build

When using SCons, it is unnecessary to add special commands or target names to clean up after a build. Instead, you simply use the `-c` or `--clean` option when you invoke SCons, and SCons removes the appropriate built files. So if we build our example above and then invoke `scons -c` afterwards, the output on POSIX looks like:

```
% scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cc -o hello.o -c hello.c
cc -o hello hello.o
scons: done building targets.
% scons -c
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Cleaning targets ...
Removed hello.o
Removed hello
scons: done cleaning targets.
```

And the output on Windows looks like:

```
C:\>scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cl /Fohello.obj /c hello.c /nologo
link /nologo /OUT:hello.exe hello.obj
embedManifestExeCheck(target, source, env)
scons: done building targets.
C:\>scons -c
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Cleaning targets ...
Removed hello.obj
Removed hello.exe
scons: done cleaning targets.
```

Notice that SCons changes its output to tell you that it is `Cleaning targets ...` and `done cleaning targets.`

2.5. The SConstruct File

If you're used to build systems like Make you've already figured out that the `SConstruct` file is the SCons equivalent of a `Makefile`. That is, the `SConstruct` file is the input file that SCons reads to control the build.

2.5.1. SConstruct Files Are Python Scripts

There is, however, an important difference between an `SConstruct` file and a `Makefile`: the `SConstruct` file is actually a Python script. If you're not already familiar with Python, don't worry. This User's Guide will introduce you step-by-step to the relatively small amount of Python you'll need to know to be able to use SCons effectively. And Python is very easy to learn.

One aspect of using Python as the scripting language is that you can put comments in your `SConstruct` file using Python's commenting convention; that is, everything between a '#' and the end of the line will be ignored:

```
# Arrange to build the "hello" program.
Program('hello.c')    # "hello.c" is the source file.
```

You'll see throughout the remainder of this Guide that being able to use the power of a real scripting language can greatly simplify the solutions to complex requirements of real-world builds.

2.5.2. SCons Functions Are Order-Independent

One important way in which the `SConstruct` file is not exactly like a normal Python script, and is more like a `Makefile`, is that the order in which the SCons functions are called in the `SConstruct` file does *not* affect the order in which SCons actually builds the programs and object files you want it to build [1]. In other words, when you call the `Program` builder (or any other builder method), you're not telling SCons to build the program at that moment. Instead, you're telling SCons what you want accomplished, and it's up to SCons to figure out how to do that, and to take those steps if/when it's necessary. We'll learn more about how SCons decides when building or rebuilding a target is necessary in Chapter 6, *Dependencies*, below.

SCons reflects this distinction between *calling a builder method like* `Program` and *actually building the program* by printing the status messages that indicate when it's "just reading" the `SConstruct` file, and when it's actually building the target files. This is to make it clear when SCons is executing the Python statements that make up the `SConstruct` file, and when SCons is actually executing the commands or other actions to build the necessary files.

Let's clarify this with an example. Python has a `print` function that prints a string of characters to the screen. If we put `print` calls around our calls to the `Program` builder method:

```
print("Calling Program('hello.c')")
Program('hello.c')
print("Calling Program('goodbye.c')")
Program('goodbye.c')
print("Finished calling Program()")
```

[1] In programming parlance, the `SConstruct` file is *declarative*, meaning you tell SCons what you want done and let it figure out the order in which to do it, rather than strictly *imperative*, where you specify explicitly the order in which to do things.

Then when we execute SCons, we see the output from calling the `print` function in between the messages about reading the `SConscript` files, indicating that is when the Python statements are being executed:

```
% scons
scons: Reading SConscript files ...
Calling Program('hello.c')
Calling Program('goodbye.c')
Finished calling Program()
scons: done reading SConscript files.
scons: Building targets ...
cc -o goodbye.o -c goodbye.c
cc -o goodbye goodbye.o
cc -o hello.o -c hello.c
cc -o hello hello.o
scons: done building targets.
```

Notice that SCons built the `goodbye` program first, even though the "reading `SConscript`" output shows that we called `Program('hello.c')` first in the `SConstruct` file.

Notice also that SCons was able to infer a lot of information from the two `Program` calls. Because `hello.c` and `goodbye.c` were recognized as C-language source files, it knew to build the intermediate target files `hello.o` and `goodbye.o` and the final files `hello` and `goodbye` It was not necessary to program **scons** beyond just calling `Program`.

2.6. Making the SCons Output Less Verbose

You've already seen how SCons prints some messages about what it's doing, surrounding the actual commands used to build the software:

```
C:\>scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cl /Fohello.obj /c hello.c /nologo
link /nologo /OUT:hello.exe hello.obj
embedManifestExeCheck(target, source, env)
scons: done building targets.
```

These messages emphasize the order in which SCons does its work: all of the configuration files (generically referred to as `SConscript` files) are read and executed first, and only then are the target files built. Among other benefits, these messages help to distinguish between errors that occur while the configuration files are read, and errors that occur while targets are being built.

One drawback, of course, is that these messages clutter the output. Fortunately, they're easily disabled by using the `-Q` option when invoking SCons:

```
C:\>scons -Q
cl /Fohello.obj /c hello.c /nologo
link /nologo /OUT:hello.exe hello.obj
embedManifestExeCheck(target, source, env)
```

Because we want this User's Guide to focus on what SCons is actually doing, we're going to use the `-Q` option to remove these messages from the output of all the remaining examples in this Guide.

3 Less Simple Things to Do With Builds

In this chapter, you will see several examples of very simple build configurations using SCons, which will demonstrate how easy it is to use SCons to build programs from several different programming languages on different types of systems.

3.1. Specifying the Name of the Target (Output) File

You've seen that when you call the `Program` builder method, it builds the resulting program with the same base name as the source file. That is, the following call to build an executable program from the `hello.c` source file will build an executable program named `hello` on POSIX systems, and an executable program named `hello.exe` on Windows systems:

```
Program('hello.c')
```

If you want to build a program with a different name than the base of the source file name, you simply put the target file name to the left of the source file name:

```
Program('new_hello', 'hello.c')
```

(SCons requires the target file name first, followed by the source file name, so that the order mimics that of an assignment statement in most programming languages, including Python: `"target = source files"`. For an alternative way to supply this information, see Section 3.6, “Keyword Arguments”).

Now SCons will build an executable program named `new_hello` when run on a POSIX system:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o new_hello hello.o
```

And SCons will build an executable program named `new_hello.exe` when run on a Windows system:

```
C:\>scons -Q
```

```
cl /Fohello.obj /c hello.c /nologo
link /nologo /OUT:new_hello.exe hello.obj
embedManifestExeCheck(target, source, env)
```

3.2. Compiling Multiple Source Files

You've just seen how to configure SCons to compile a program from a single source file. It's more common, of course, that you'll need to build a program from many input source files, not just one. To do this, you need to put the source files in a Python list (enclosed in square brackets), like so:

```
Program(['prog.c', 'file1.c', 'file2.c'])
```

A build of the above example would look like:

```
% scons -Q
cc -o file1.o -c file1.c
cc -o file2.o -c file2.c
cc -o prog.o -c prog.c
cc -o prog prog.o file1.o file2.o
```

Notice that SCons deduces the output program name from the first source file specified in the list--that is, because the first source file was `prog.c`, SCons will name the resulting program `prog` (or `prog.exe` on a Windows system). If you want to specify a different program name, then (as we've seen in the previous section) you slide the list of source files over to the right to make room for the output program file name. (SCons puts the output file name to the left of the source file names so that the order mimics that of an assignment statement: `program = source files`.) This makes our example:

```
Program('program', ['prog.c', 'file1.c', 'file2.c'])
```

On Linux, a build of this example would look like:

```
% scons -Q
cc -o file1.o -c file1.c
cc -o file2.o -c file2.c
cc -o prog.o -c prog.c
cc -o program prog.o file1.o file2.o
```

Or on Windows:

```
C:\>scons -Q
cl /Fofile1.obj /c file1.c /nologo
cl /Fofile2.obj /c file2.c /nologo
cl /Foprog.obj /c prog.c /nologo
link /nologo /OUT:program.exe prog.obj file1.obj file2.obj
embedManifestExeCheck(target, source, env)
```

3.3. Making a list of files with `Glob`

You can also use the `Glob` function to find all files matching a certain template, using the standard shell pattern matching characters `*`, `?` and `[abc]` to match any of `a`, `b` or `c`. `[!abc]` is also supported, to match any character *except* `a`, `b` or `c`. This makes many multi-source-file builds quite easy:

```
Program('program', Glob('*.c'))
```

The SCons man page has more details on using `Glob` with variant directories (see ???, below) and repositories (see Chapter 16, *Building From Code Repositories*, below), excluding some files and returning strings rather than Nodes.

3.4. Specifying Single Files Vs. Lists of Files

We've now shown you two ways to specify the source for a program, one with a list of files:

```
Program('hello', ['file1.c', 'file2.c'])
```

And one with a single file:

```
Program('hello', 'hello.c')
```

You could actually put a single file name in a list, too, which you might prefer just for the sake of consistency:

```
Program('hello', ['hello.c'])
```

SCons functions will accept a single file name in either form. In fact, internally, SCons treats all input as lists of files, but allows you to omit the square brackets to cut down a little on the typing when there's only a single file name.

Important

Although SCons functions are forgiving about whether or not you use a string vs. a list for a single file name, Python itself is more strict about treating lists and strings differently. So where SCons allows either a string or list:

```
# The following two calls both work correctly:
Program('program1', 'program1.c')
Program('program2', ['program2.c'])
```

Trying to do "Python things" that mix strings and lists will cause errors or lead to incorrect results:

```
common_sources = ['file1.c', 'file2.c']

# THE FOLLOWING IS INCORRECT AND GENERATES A PYTHON ERROR
# BECAUSE IT TRIES TO ADD A STRING TO A LIST:
Program('program1', common_sources + 'program1.c')

# The following works correctly, because it's adding two
# lists together to make another list.
Program('program2', common_sources + ['program2.c'])
```

3.5. Making Lists of Files Easier to Read

One drawback to the use of a Python list for source files is that each file name must be enclosed in quotes (either single quotes or double quotes). This can get cumbersome and difficult to read when the list of file names is long. Fortunately, SCons and Python provide a number of ways to make sure that the `SConstruct` file stays easy to read.

To make long lists of file names easier to deal with, SCons provides a `Split` function that takes a quoted list of file names, with the names separated by spaces or other white-space characters, and turns it into a list of separate file names. Using the `Split` function turns the previous example into:

```
Program('program', Split('main.c file1.c file2.c'))
```

(If you're already familiar with Python, you'll have realized that this is similar to the `split()` method in the Python standard `string` module. Unlike the `split()` member function of strings, however, the `Split` function does not require a string as input and will wrap up a single non-string object in a list, or return its argument untouched if it's already a list. This comes in handy as a way to make sure arbitrary values can be passed to SCons functions without having to check the type of the variable by hand.)

Putting the call to the `Split` function inside the `Program` call can also be a little unwieldy. A more readable alternative is to assign the output from the `Split` call to a variable name, and then use the variable when calling the `Program` function:

```
src_files = Split('main.c file1.c file2.c')
Program('program', src_files)
```

Lastly, the `Split` function doesn't care how much white space separates the file names in the quoted string. This allows you to create lists of file names that span multiple lines, which often makes for easier editing:

```
src_files = Split("""main.c
                     file1.c
                     file2.c""")
Program('program', src_files)
```

(Note in this example that we used the Python "triple-quote" syntax, which allows a string to contain multiple lines. The three quotes can be either single or double quotes.)

3.6. Keyword Arguments

SCons also allows you to identify the output file and input source files using Python keyword arguments *`target`* and *`source`*. The Python syntax for this is:

```
src_files = Split('main.c file1.c file2.c')
Program(target='program', source=src_files)
```

Because the keywords explicitly identify what each argument is, the order does not matter and you can reverse it if you prefer:

```
src_files = Split('main.c file1.c file2.c')
Program(source=src_files, target='program')
```

Whether or not you choose to use keyword arguments to identify the target and source files, and the order in which you specify them when using keywords, are purely personal choices; SCons functions the same regardless.

3.7. Compiling Multiple Programs

In order to compile multiple programs within the same `SConstruct` file, simply call the `Program` method multiple times, once for each program you need to build:

```
Program('foo.c')
Program('bar', ['bar1.c', 'bar2.c'])
```

SCons would then build the programs as follows:

```
% scons -Q
cc -o bar1.o -c bar1.c
cc -o bar2.o -c bar2.c
cc -o bar bar1.o bar2.o
cc -o foo.o -c foo.c
cc -o foo foo.o
```

Notice that SCons does not necessarily build the programs in the same order in which you specify them in the `SConstruct` file. SCons does, however, recognize that the individual object files must be built before the resulting program can be built. We'll discuss this in greater detail in the "Dependencies" section, below.

3.8. Sharing Source Files Between Multiple Programs

It's common to re-use code by sharing source files between multiple programs. One way to do this is to create a library from the common source files, which can then be linked into resulting programs. (Creating libraries is discussed in Chapter 4, *Building and Linking with Libraries*, below.)

A more straightforward, but perhaps less convenient, way to share source files between multiple programs is simply to include the common files in the lists of source files for each program:

```
Program(Split('foo.c common1.c common2.c'))
Program('bar', Split('bar1.c bar2.c common1.c common2.c'))
```

SCons recognizes that the object files for the `common1.c` and `common2.c` source files each need to be built only once, even though the resulting object files are each linked in to both of the resulting executable programs:

```
% scons -Q
cc -o bar1.o -c bar1.c
cc -o bar2.o -c bar2.c
cc -o common1.o -c common1.c
```

```
cc -o common2.o -c common2.c
cc -o bar bar1.o bar2.o common1.o common2.o
cc -o foo.o -c foo.c
cc -o foo foo.o common1.o common2.o
```

If two or more programs share a lot of common source files, repeating the common files in the list for each program can be a maintenance problem when you need to change the list of common files. You can simplify this by creating a separate Python list to hold the common file names, and concatenating it with other lists using the Python + operator:

```
common = ['common1.c', 'common2.c']
foo_files = ['foo.c'] + common
bar_files = ['bar1.c', 'bar2.c'] + common
Program('foo', foo_files)
Program('bar', bar_files)
```

This is functionally equivalent to the previous example.

4 Building and Linking with Libraries

It's often useful to organize large software projects by collecting parts of the software into one or more libraries. SCons makes it easy to create libraries and to use them in the programs.

4.1. Building Libraries

You build your own libraries by specifying `Library` instead of `Program`:

```
Library('foo', ['f1.c', 'f2.c', 'f3.c'])
```

SCons uses the appropriate library prefix and suffix for your system. So on POSIX or Linux systems, the above example would build as follows (although ranlib may not be called on all systems):

```
% scons -Q
cc -o f1.o -c f1.c
cc -o f2.o -c f2.c
cc -o f3.o -c f3.c
ar rc libfoo.a f1.o f2.o f3.o
ranlib libfoo.a
```

On a Windows system, a build of the above example would look like:

```
C:\>scons -Q
cl /Fof1.obj /c f1.c /nologo
cl /Fof2.obj /c f2.c /nologo
cl /Fof3.obj /c f3.c /nologo
lib /nologo /OUT:foo.lib f1.obj f2.obj f3.obj
```

The rules for the target name of the library are similar to those for programs: if you don't explicitly specify a target library name, SCons will deduce one from the name of the first source file specified, and SCons will add an appropriate file prefix and suffix if you leave them off.

4.1.1. Building Libraries From Source Code or Object Files

The previous example shows building a library from a list of source files. You can, however, also give the `Library` call object files, and it will correctly realize they are object files. In fact, you can arbitrarily mix source code files and object files in the source list:

```
Library('foo', ['f1.c', 'f2.o', 'f3.c', 'f4.o'])
```

And SCons realizes that only the source code files must be compiled into object files before creating the final library:

```
% scons -Q
cc -o f1.o -c f1.c
cc -o f3.o -c f3.c
ar rc libfoo.a f1.o f2.o f3.o f4.o
ranlib libfoo.a
```

Of course, in this example, the object files must already exist for the build to succeed. See Chapter 5, *Node Objects*, below, for information about how you can build object files explicitly and include the built files in a library.

4.1.2. Building Static Libraries Explicitly: the `StaticLibrary` Builder

The `Library` function builds a traditional static library. If you want to be explicit about the type of library being built, you can use the synonym `StaticLibrary` function instead of `Library`:

```
StaticLibrary('foo', ['f1.c', 'f2.c', 'f3.c'])
```

There is no functional difference between the `StaticLibrary` and `Library` functions.

4.1.3. Building Shared (DLL) Libraries: the `SharedLibrary` Builder

If you want to build a shared library (on POSIX systems) or a DLL file (on Windows systems), you use the `SharedLibrary` function:

```
SharedLibrary('foo', ['f1.c', 'f2.c', 'f3.c'])
```

The output on POSIX:

```
% scons -Q
cc -o f1.os -c f1.c
cc -o f2.os -c f2.c
cc -o f3.os -c f3.c
cc -o libfoo.so -shared f1.os f2.os f3.os
```

And the output on Windows:

```
C:\>scons -Q
cl /Fof1.obj /c f1.c /nologo
cl /Fof2.obj /c f2.c /nologo
cl /Fof3.obj /c f3.c /nologo
link /nologo /dll /out:foo.dll /implib:foo.lib f1.obj f2.obj f3.obj
RegServerFunc(target, source, env)
embedManifestDllCheck(target, source, env)
```

Notice again that SCons takes care of building the output file correctly, adding the `-shared` option for a POSIX compilation, and the `/dll` option on Windows.

4.2. Linking with Libraries

Usually, you build a library because you want to link it with one or more programs. You link libraries with a program by specifying the libraries in the `$LIBS` construction variable, and by specifying the directory in which the library will be found in the `$LIBPATH` construction variable:

```
Library('foo', ['f1.c', 'f2.c', 'f3.c'])
Program('prog.c', LIBS=['foo', 'bar'], LIBPATH='.')
```

Notice, of course, that you don't need to specify a library prefix (like `lib`) or suffix (like `.a` or `.lib`). SCons uses the correct prefix or suffix for the current system.

On a POSIX or Linux system, a build of the above example would look like:

```
% scons -Q
cc -o f1.o -c f1.c
cc -o f2.o -c f2.c
cc -o f3.o -c f3.c
ar rc libfoo.a f1.o f2.o f3.o
ranlib libfoo.a
cc -o prog.o -c prog.c
cc -o prog prog.o -L. -lfoo -lbar
```

On a Windows system, a build of the above example would look like:

```
C:\>scons -Q
cl /Fof1.obj /c f1.c /nologo
cl /Fof2.obj /c f2.c /nologo
cl /Fof3.obj /c f3.c /nologo
lib /nologo /OUT:foo.lib f1.obj f2.obj f3.obj
cl /Foprog.obj /c prog.c /nologo
link /nologo /OUT:prog.exe /LIBPATH:. foo.lib bar.lib prog.obj
embedManifestExeCheck(target, source, env)
```

As usual, notice that SCons has taken care of constructing the correct command lines to link with the specified library on each system.

Note also that, if you only have a single library to link with, you can specify the library name in single string, instead of a Python list, so that:

```
Program('prog.c', LIBS='foo', LIBPATH='.')
```

```
```

is equivalent to:

```
Program('prog.c', LIBS=['foo'], LIBPATH='.')
```

This is similar to the way that SCons handles either a string or a list to specify a single source file.

4.3. Finding Libraries: the $LIBPATH Construction Variable

By default, the linker will only look in certain system-defined directories for libraries. SCons knows how to look for libraries in directories that you specify with the $LIBPATH construction variable. $LIBPATH consists of a list of directory names, like so:

```
Program('prog.c', LIBS = 'm',
                  LIBPATH = ['/usr/lib', '/usr/local/lib'])
```

Using a Python list is preferred because it's portable across systems. Alternatively, you could put all of the directory names in a single string, separated by the system-specific path separator character: a colon on POSIX systems:

```
LIBPATH = '/usr/lib:/usr/local/lib'
```

or a semi-colon on Windows systems:

```
LIBPATH = 'C:\\lib;D:\\lib'
```

(Note that Python requires that the backslash separators in a Windows path name be escaped within strings.)

When the linker is executed, SCons will create appropriate flags so that the linker will look for libraries in the same directories as SCons. So on a POSIX or Linux system, a build of the above example would look like:

```
% scons -Q
cc -o prog.o -c prog.c
cc -o prog prog.o -L/usr/lib -L/usr/local/lib -lm
```

On a Windows system, a build of the above example would look like:

```
C:\>scons -Q
cl /Foprog.obj /c prog.c /nologo
link /nologo /OUT:prog.exe /LIBPATH:\usr\lib /LIBPATH:\usr\local\lib m.lib prog.obj
embedManifestExeCheck(target, source, env)
```

Note again that SCons has taken care of the system-specific details of creating the right command-line options.

5 Node Objects

Internally, SCons represents all of the files and directories it knows about as `Nodes`. These internal objects (not object *files*) can be used in a variety of ways to make your `SConscript` files portable and easy to read.

5.1. Builder Methods Return Lists of Target Nodes

All builder methods return a list of `Node` objects that identify the target file or files that will be built. These returned `Nodes` can be passed as arguments to other builder methods.

For example, suppose that we want to build the two object files that make up a program with different options. This would mean calling the `Object` builder once for each object file, specifying the desired options:

```
Object('hello.c', CCFLAGS='-DHELLO')
Object('goodbye.c', CCFLAGS='-DGOODBYE')
```

One way to combine these object files into the resulting program would be to call the `Program` builder with the names of the object files listed as sources:

```
Object('hello.c', CCFLAGS='-DHELLO')
Object('goodbye.c', CCFLAGS='-DGOODBYE')
Program(['hello.o', 'goodbye.o'])
```

The problem with specifying the names as strings is that our `SConstruct` file is no longer portable across operating systems. It won't, for example, work on Windows because the object files there would be named `hello.obj` and `goodbye.obj`, not `hello.o` and `goodbye.o`.

A better solution is to assign the lists of targets returned by the calls to the `Object` builder to variables, which we can then concatenate in our call to the `Program` builder:

```
hello_list = Object('hello.c', CCFLAGS='-DHELLO')
goodbye_list = Object('goodbye.c', CCFLAGS='-DGOODBYE')
Program(hello_list + goodbye_list)
```

This makes our `SConstruct` file portable again, the build output on Linux looking like:

```
% scons -Q
cc -o goodbye.o -c -DGOODBYE goodbye.c
cc -o hello.o -c -DHELLO hello.c
cc -o hello hello.o goodbye.o
```

And on Windows:

```
C:\>scons -Q
cl /Fogoodbye.obj /c goodbye.c -DGOODBYE
cl /Fohello.obj /c hello.c -DHELLO
link /nologo /OUT:hello.exe hello.obj goodbye.obj
embedManifestExeCheck(target, source, env)
```

We'll see examples of using the list of nodes returned by builder methods throughout the rest of this guide.

5.2. Explicitly Creating File and Directory Nodes

It's worth mentioning here that SCons maintains a clear distinction between Nodes that represent files and Nodes that represent directories. SCons supports `File` and `Dir` functions that, respectively, return a file or directory Node:

```
hello_c = File('hello.c')
Program(hello_c)

classes = Dir('classes')
Java(classes, 'src')
```

Normally, you don't need to call `File` or `Dir` directly, because calling a builder method automatically treats strings as the names of files or directories, and translates them into the Node objects for you. The `File` and `Dir` functions can come in handy in situations where you need to explicitly instruct SCons about the type of Node being passed to a builder or other function, or unambiguously refer to a specific file in a directory tree.

There are also times when you may need to refer to an entry in a file system without knowing in advance whether it's a file or a directory. For those situations, SCons also supports an `Entry` function, which returns a Node that can represent either a file or a directory.

```
xyzzy = Entry('xyzzy')
```

The returned `xyzzy` Node will be turned into a file or directory Node the first time it is used by a builder method or other function that requires one vs. the other.

5.3. Printing Node File Names

One of the most common things you can do with a Node is use it to print the file name that the node represents. Keep in mind, though, that because the object returned by a builder call is a *list* of Nodes, you must use Python subscripts to fetch individual Nodes from the list. For example, the following `SConstruct` file:

```
```

```
object_list = Object('hello.c')
program_list = Program(object_list)
print("The object file is: %s"%object_list[0])
print("The program file is: %s"%program_list[0])
```

Would print the following file names on a POSIX system:

```
% scons -Q
The object file is: hello.o
The program file is: hello
cc -o hello.o -c hello.c
cc -o hello hello.o
```

And the following file names on a Windows system:

```
C:\>scons -Q
The object file is: hello.obj
The program file is: hello.exe
cl /Fohello.obj /c hello.c /nologo
link /nologo /OUT:hello.exe hello.obj
embedManifestExeCheck(target, source, env)
```

Note that in the above example, the `object_list[0]` extracts an actual Node *object* from the list, and the Python `print` function converts the object to a string for printing.

5.4. Using a Node's File Name as a String

Printing a `Node`'s name as described in the previous section works because the string representation of a `Node` object is the name of the file. If you want to do something other than print the name of the file, you can fetch it by using the builtin Python `str` function. For example, if you want to use the Python `os.path.exists` to figure out whether a file exists while the `SConstruct` file is being read and executed, you can fetch the string as follows:

```
import os.path
program_list = Program('hello.c')
program_name = str(program_list[0])
if not os.path.exists(program_name):
    print("%s does not exist!"%program_name)
```

Which executes as follows on a POSIX system:

```
% scons -Q
hello does not exist!
cc -o hello.o -c hello.c
cc -o hello hello.o
```

5.5. `GetBuildPath`: Getting the Path From a Node or String

`env.GetBuildPath(file_or_list)` returns the path of a `Node` or a string representing a path. It can also take a list of `Node`s and/or strings, and returns the list of paths. If passed a single `Node`, the result is the same as calling

str(node) (see above). The string(s) can have embedded construction variables, which are expanded as usual, using the calling environment's set of variables. The paths can be files or directories, and do not have to exist.

```
env=Environment(VAR="value")
n=File("foo.c")
print(env.GetBuildPath([n, "sub/dir/$VAR"]))
```

Would print the following file names:

```
% scons -Q
['foo.c', 'sub/dir/value']
scons: `.' is up to date.
```

There is also a function version of GetBuildPath which can be called without an Environment; that uses the default SCons Environment to do substitution on any string arguments.

6 Dependencies

So far we've seen how SCons handles one-time builds. But one of the main functions of a build tool like SCons is to rebuild only what is necessary when source files change--or, put another way, SCons should *not* waste time rebuilding things that don't need to be rebuilt. You can see this at work simply by re-invoking SCons after building our simple `hello` example:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q
scons: `.' is up to date.
```

The second time it is executed, SCons realizes that the `hello` program is up-to-date with respect to the current `hello.c` source file, and avoids rebuilding it. You can see this more clearly by naming the `hello` program explicitly on the command line:

```
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q hello
scons: `hello' is up to date.
```

Note that SCons reports `"...is up to date"` only for target files named explicitly on the command line, to avoid cluttering the output.

6.1. Deciding When an Input File Has Changed: the `Decider` Function

Another aspect of avoiding unnecessary rebuilds is the fundamental build tool behavior of *rebuilding* things when an input file changes, so that the built software is up to date. By default, SCons keeps track of this through an MD5 `signature`, or checksum, of the contents of each file, although you can easily configure SCons to use the modification times (or time stamps) instead. You can even specify your own Python function for deciding if an input file has changed.

6.1.1. Using MD5 Signatures to Decide if a File Has Changed

By default, SCons keeps track of whether a file has changed based on an MD5 checksum of the file's contents, not the file's modification time. This means that you may be surprised by the default SCons behavior if you are used to the Make convention of forcing a rebuild by updating the file's modification time (using the touch command, for example):

```
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
% touch hello.c
% scons -Q hello
scons: `hello' is up to date.
```

Even though the file's modification time has changed, SCons realizes that the contents of the `hello.c` file have *not* changed, and therefore that the `hello` program need not be rebuilt. This avoids unnecessary rebuilds when, for example, someone rewrites the contents of a file without making a change. But if the contents of the file really do change, then SCons detects the change and rebuilds the program as required:

```
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
%     [CHANGE THE CONTENTS OF hello.c]
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
```

Note that you can, if you wish, specify this default behavior (MD5 signatures) explicitly using the `Decider` function as follows:

```
Program('hello.c')
Decider('MD5')
```

You can also use the string `'content'` as a synonym for `'MD5'` when calling the `Decider` function.

6.1.1.1. Ramifications of Using MD5 Signatures

Using MD5 signatures to decide if an input file has changed has one surprising benefit: if a source file has been changed in such a way that the contents of the rebuilt target file(s) will be exactly the same as the last time the file was built, then any "downstream" target files that depend on the rebuilt-but-not-changed target file actually need not be rebuilt.

So if, for example, a user were to only change a comment in a `hello.c` file, then the rebuilt `hello.o` file would be exactly the same as the one previously built (assuming the compiler doesn't put any build-specific information in the object file). SCons would then realize that it would not need to rebuild the `hello` program as follows:

```
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
%   [CHANGE A COMMENT IN hello.c]
% scons -Q hello
cc -o hello.o -c hello.c
scons: `hello' is up to date.
```

In essence, SCons "short-circuits" any dependent builds when it realizes that a target file has been rebuilt to exactly the same file as the last build. This does take some extra processing time to read the contents of the target (`hello.o`) file, but often saves time when the rebuild that was avoided would have been time-consuming and expensive.

6.1.2. Using Time Stamps to Decide If a File Has Changed

If you prefer, you can configure SCons to use the modification time of a file, not the file contents, when deciding if a target needs to be rebuilt. SCons gives you two ways to use time stamps to decide if an input file has changed since the last time a target has been built.

The most familiar way to use time stamps is the way Make does: that is, have SCons decide that a target must be rebuilt if a source file's modification time is *newer* than the target file. To do this, call the `Decider` function as follows:

```
Object('hello.c')
Decider('timestamp-newer')
```

This makes SCons act like Make when a file's modification time is updated (using the touch command, for example):

```
% scons -Q hello.o
cc -o hello.o -c hello.c
% touch hello.c
% scons -Q hello.o
cc -o hello.o -c hello.c
```

And, in fact, because this behavior is the same as the behavior of Make, you can also use the string `'make'` as a synonym for `'timestamp-newer'` when calling the `Decider` function:

```
Object('hello.c')
Decider('make')
```

One drawback to using times stamps exactly like Make is that if an input file's modification time suddenly becomes *older* than a target file, the target file will not be rebuilt. This can happen if an old copy of a source file is restored from a backup archive, for example. The contents of the restored file will likely be different than they were the last time a dependent target was built, but the target won't be rebuilt because the modification time of the source file is not newer than the target.

Because SCons actually stores information about the source files' time stamps whenever a target is built, it can handle this situation by checking for an exact match of the source file time stamp, instead of just whether or not the source file is newer than the target file. To do this, specify the argument `'timestamp-match'` when calling the `Decider` function:

```
Object('hello.c')
Decider('timestamp-match')
```

When configured this way, SCons will rebuild a target whenever a source file's modification time has changed. So if we use the `touch -t` option to change the modification time of `hello.c` to an old date (January 1, 1989), SCons will still rebuild the target file:

```
% scons -Q hello.o
```

```
cc -o hello.o -c hello.c
% touch -t 198901010000 hello.c
% scons -Q hello.o
cc -o hello.o -c hello.c
```

In general, the only reason to prefer `timestamp-newer` instead of `timestamp-match`, would be if you have some specific reason to require this Make-like behavior of not rebuilding a target when an otherwise-modified source file is older.

6.1.3. Deciding If a File Has Changed Using Both MD Signatures and Time Stamps

As a performance enhancement, SCons provides a way to use MD5 checksums of file contents but to read those contents only when the file's timestamp has changed. To do this, call the `Decider` function with `'MD5-timestamp'` argument as follows:

```
Program('hello.c')
Decider('MD5-timestamp')
```

So configured, SCons will still behave like it does when using `Decider('MD5')`:

```
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
% touch hello.c
% scons -Q hello
scons: `hello' is up to date.
% edit hello.c
    [CHANGE THE CONTENTS OF hello.c]
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
```

However, the second call to SCons in the above output, when the build is up-to-date, will have been performed by simply looking at the modification time of the `hello.c` file, not by opening it and performing an MD5 checksum calcuation on its contents. This can significantly speed up many up-to-date builds.

The only drawback to using `Decider('MD5-timestamp')` is that SCons will *not* rebuild a target file if a source file was modified within one second of the last time SCons built the file. While most developers are programming, this isn't a problem in practice, since it's unlikely that someone will have built and then thought quickly enough to make a substantive change to a source file within one second. Certain build scripts or continuous integration tools may, however, rely on the ability to apply changes to files automatically and then rebuild as quickly as possible, in which case use of `Decider('MD5-timestamp')` may not be appropriate.

6.1.4. Extending SCons: Writing Your Own Custom `Decider` Function

The different string values that we've passed to the `Decider` function are essentially used by SCons to pick one of several specific internal functions that implement various ways of deciding if a dependency (usually a source file)

has changed since a target file has been built. As it turns out, you can also supply your own function to decide if a dependency has changed.

For example, suppose we have an input file that contains a lot of data, in some specific regular format, that is used to rebuild a lot of different target files, but each target file really only depends on one particular section of the input file. We'd like to have each target file depend on only its section of the input file. However, since the input file may contain a lot of data, we want to open the input file only if its timestamp has changed. This could be done with a custom Decider function that might look something like this:

```
Program('hello.c')
def decide_if_changed(dependency, target, prev_ni, repo_node=None):
    if dependency.get_timestamp() != prev_ni.timestamp:
        dep = str(dependency)
        tgt = str(target)
        if specific_part_of_file_has_changed(dep, tgt):
            return True
    return False
Decider(decide_if_changed)
```

Note that in the function definition, the dependency (input file) is the first argument, and then the target. Both of these are passed to the functions as SCons Node objects, which we convert to strings using the Python str().

The third argument, prev_ni, is an object that holds the signature or timestamp information that was recorded about the dependency the last time the target was built. A prev_ni object can hold different information, depending on the type of thing that the dependency argument represents. For normal files, the prev_ni object has the following attributes:

.csig
The *content signature*, or MD5 checksum, of the contents of the dependency file the last time the target was built.

.size
The size in bytes of the dependency file the last time the target was built.

.timestamp
The modification time of the dependency file the last time the target was built.

These attributes may not be present at the time of the first run. Without any prior build, no targets have been created and no .sconsign DB file exists yet. So you should always check whether the prev_ni attribute in question is available (use the Python hasattr method or a try-except block).

The fourth argument repo_node is the Node to use if it is not None when comparing BuildInfo. This is typically only set when the target node only exists in a Repository

Note that ignoring some of the arguments in your custom Decider function is a perfectly normal thing to do, if they don't impact the way you want to decide if the dependency file has changed.

We finally present a small example for a csig-based decider function. Note how the signature information for the dependency file has to get initialized via get_csig during each function call (this is mandatory!).

```
env = Environment()
```

```
def config_file_decider(dependency, target, prev_ni, repo_node=None):
    import os.path

    # We always have to init the .csig value...
    dep_csig = dependency.get_csig()
    # .csig may not exist, because no target was built yet...
    if not prev_ni.hasattr("csig"):
        return True
    # Target file may not exist yet
    if not os.path.exists(str(target.abspath)):
        return True
    if dep_csig != prev_ni.csig:
        # Some change on source file => update installed one
        return True
    return False


def update_file():
    with open("test.txt", "a") as f:
        f.write("some line\n")


update_file()

# Activate our own decider function
env.Decider(config_file_decider)

env.Install("install", "test.txt")
```

6.1.5. Mixing Different Ways of Deciding If a File Has Changed

The previous examples have all demonstrated calling the global `Decider` function to configure all dependency decisions that SCons makes. Sometimes, however, you want to be able to configure different decision-making for different targets. When that's necessary, you can use the `env.Decider` method to affect only the configuration decisions for targets built with a specific construction environment.

For example, if we arbitrarily want to build one program using MD5 checkums and another using file modification times from the same source we might configure it this way:

```
env1 = Environment(CPPPATH = ['.'])
env2 = env1.Clone()
env2.Decider('timestamp-match')
env1.Program('prog-MD5', 'program1.c')
env2.Program('prog-timestamp', 'program2.c')
```

If both of the programs include the same `inc.h` file, then updating the modification time of `inc.h` (using the touch command) will cause only `prog-timestamp` to be rebuilt:

```
% scons -Q
cc -o program1.o -c -I. program1.c
```

```
cc -o prog-MD5 program1.o
cc -o program2.o -c -I. program2.c
cc -o prog-timestamp program2.o
% touch inc.h
% scons -Q
cc -o program2.o -c -I. program2.c
cc -o prog-timestamp program2.o
```

6.2. Implicit Dependencies: The $CPPPATH Construction Variable

Now suppose that our "Hello, World!" program actually has an `#include` line to include the `hello.h` file in the compilation:

```
#include <hello.h>
int
main()
{
    printf("Hello, %s!\n", string);
}
```

And, for completeness, the `hello.h` file looks like this:

```
#define string    "world"
```

In this case, we want SCons to recognize that, if the contents of the `hello.h` file change, the `hello` program must be recompiled. To do this, we need to modify the `SConstruct` file like so:

```
Program('hello.c', CPPPATH = '.')
```

The `$CPPPATH` value tells SCons to look in the current directory (`'.'`) for any files included by C source files (`.c` or `.h` files). With this assignment in the `SConstruct` file:

```
% scons -Q hello
cc -o hello.o -c -I. hello.c
cc -o hello hello.o
% scons -Q hello
scons: `hello' is up to date.
%     [CHANGE THE CONTENTS OF hello.h]
% scons -Q hello
cc -o hello.o -c -I. hello.c
cc -o hello hello.o
```

First, notice that SCons added the `-I.` argument from the `$CPPPATH` variable so that the compilation would find the `hello.h` file in the local directory.

Second, realize that SCons knows that the `hello` program must be rebuilt because it scans the contents of the `hello.c` file for the `#include` lines that indicate another file is being included in the compilation. SCons records

these as *implicit dependencies* of the target file, Consequently, when the `hello.h` file changes, SCons realizes that the `hello.c` file includes it, and rebuilds the resulting `hello` program that depends on both the `hello.c` and `hello.h` files.

Like the `$LIBPATH` variable, the `$CPPPATH` variable may be a list of directories, or a string separated by the system-specific path separation character (':' on POSIX/Linux, ';' on Windows). Either way, SCons creates the right command-line options so that the following example:

```
Program('hello.c', CPPPATH = ['include', '/home/project/inc'])
```

Will look like this on POSIX or Linux:

```
% scons -Q hello
cc -o hello.o -c -Iinclude -I/home/project/inc hello.c
cc -o hello hello.o
```

And like this on Windows:

```
C:\>scons -Q hello.exe
cl /Fohello.obj /c hello.c /nologo /Iinclude /I\home\project\inc
link /nologo /OUT:hello.exe hello.obj
embedManifestExeCheck(target, source, env)
```

6.3. Caching Implicit Dependencies

Scanning each file for `#include` lines does take some extra processing time. When you're doing a full build of a large system, the scanning time is usually a very small percentage of the overall time spent on the build. You're most likely to notice the scanning time, however, when you *rebuild* all or part of a large system: SCons will likely take some extra time to "think about" what must be built before it issues the first build command (or decides that everything is up to date and nothing must be rebuilt).

In practice, having SCons scan files saves time relative to the amount of potential time lost to tracking down subtle problems introduced by incorrect dependencies. Nevertheless, the "waiting time" while SCons scans files can annoy individual developers waiting for their builds to finish. Consequently, SCons lets you cache the implicit dependencies that its scanners find, for use by later builds. You can do this by specifying the `--implicit-cache` option on the command line:

```
% scons -Q --implicit-cache hello
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q hello
scons: `hello' is up to date.
```

If you don't want to specify `--implicit-cache` on the command line each time, you can make it the default behavior for your build by setting the `implicit_cache` option in an `SConscript` file:

```
SetOption('implicit_cache', 1)
```

SCons does not cache implicit dependencies like this by default because the `--implicit-cache` causes SCons to simply use the implicit dependencies stored during the last run, without any checking for whether or not those dependencies are still correct. Specifically, this means `--implicit-cache` instructs SCons to *not* rebuild "correctly" in the following cases:

- When --implicit-cache is used, SCons will ignore any changes that may have been made to search paths (like $CPPPATH or $LIBPATH,). This can lead to SCons not rebuilding a file if a change to $CPPPATH would normally cause a different, same-named file from a different directory to be used.
- When --implicit-cache is used, SCons will not detect if a same-named file has been added to a directory that is earlier in the search path than the directory in which the file was found last time.

6.3.1. The --implicit-deps-changed Option

When using cached implicit dependencies, sometimes you want to "start fresh" and have SCons re-scan the files for which it previously cached the dependencies. For example, if you have recently installed a new version of external code that you use for compilation, the external header files will have changed and the previously-cached implicit dependencies will be out of date. You can update them by running SCons with the --implicit-deps-changed option:

```
% scons -Q --implicit-deps-changed hello
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q hello
scons: `hello' is up to date.
```

In this case, SCons will re-scan all of the implicit dependencies and cache updated copies of the information.

6.3.2. The --implicit-deps-unchanged Option

By default when caching dependencies, SCons notices when a file has been modified and re-scans the file for any updated implicit dependency information. Sometimes, however, you may want to force SCons to use the cached implicit dependencies, even if the source files changed. This can speed up a build for example, when you have changed your source files but know that you haven't changed any #include lines. In this case, you can use the --implicit-deps-unchanged option:

```
% scons -Q --implicit-deps-unchanged hello
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q hello
scons: `hello' is up to date.
```

In this case, SCons will assume that the cached implicit dependencies are correct and will not bother to re-scan changed files. For typical builds after small, incremental changes to source files, the savings may not be very big, but sometimes every bit of improved performance counts.

6.4. Explicit Dependencies: the Depends Function

Sometimes a file depends on another file that is not detected by an SCons scanner. For this situation, SCons allows you to specific explicitly that one file depends on another file, and must be rebuilt whenever that file changes. This is specified using the Depends method:

```
hello = Program('hello.c')
Depends(hello, 'other_file')
```

```
% scons -Q hello
cc -c hello.c -o hello.o
cc -o hello hello.o
% scons -Q hello
scons: `hello' is up to date.
% edit other_file
    [CHANGE THE CONTENTS OF other_file]
% scons -Q hello
cc -c hello.c -o hello.o
cc -o hello hello.o
```

Note that the dependency (the second argument to Depends) may also be a list of Node objects (for example, as returned by a call to a Builder):

```
hello = Program('hello.c')
goodbye = Program('goodbye.c')
Depends(hello, goodbye)
```

in which case the dependency or dependencies will be built before the target(s):

```
% scons -Q hello
cc -c goodbye.c -o goodbye.o
cc -o goodbye goodbye.o
cc -c hello.c -o hello.o
cc -o hello hello.o
```

6.5. Dependencies From External Files: the ParseDepends Function

SCons has built-in scanners for a number of languages. Sometimes these scanners fail to extract certain implicit dependencies due to limitations of the scanner implementation.

The following example illustrates a case where the built-in C scanner is unable to extract the implicit dependency on a header file.

```
#define FOO_HEADER <foo.h>
#include FOO_HEADER

int main() {
    return FOO;
}
```

```
% scons -Q
cc -o hello.o -c -I. hello.c
cc -o hello hello.o
%    [CHANGE CONTENTS OF foo.h]
% scons -Q
```

```
scons: `.' is up to date.
```

Apparently, the scanner does not know about the header dependency. Being not a full-fledged C preprocessor, the scanner does not expand the macro.

In these cases, you may also use the compiler to extract the implicit dependencies. ParseDepends can parse the contents of the compiler output in the style of Make, and explicitly establish all of the listed dependencies.

The following example uses ParseDepends to process a compiler generated dependency file which is generated as a side effect during compilation of the object file:

```
obj = Object('hello.c', CCFLAGS='-MD -MF hello.d', CPPPATH='.')
SideEffect('hello.d', obj)
ParseDepends('hello.d')
Program('hello', obj)
```

```
% scons -Q
cc -o hello.o -c -MD -MF hello.d -I. hello.c
cc -o hello hello.o
%    [CHANGE CONTENTS OF foo.h]
% scons -Q
cc -o hello.o -c -MD -MF hello.d -I. hello.c
```

Parsing dependencies from a compiler-generated .d file has a chicken-and-egg problem, that causes unnecessary rebuilds:

```
% scons -Q
cc -o hello.o -c -MD -MF hello.d -I. hello.c
cc -o hello hello.o
% scons -Q --debug=explain
scons: rebuilding `hello.o' because `foo.h' is a new dependency
cc -o hello.o -c -MD -MF hello.d -I. hello.c
% scons -Q
scons: `.' is up to date.
```

In the first pass, the dependency file is generated while the object file is compiled. At that time, SCons does not know about the dependency on foo.h. In the second pass, the object file is regenerated because foo.h is detected as a new dependency.

ParseDepends immediately reads the specified file at invocation time and just returns if the file does not exist. A dependency file generated during the build process is not automatically parsed again. Hence, the compiler-extracted dependencies are not stored in the signature database during the same build pass. This limitation of ParseDepends leads to unnecessary recompilations. Therefore, ParseDepends should only be used if scanners are not available for the employed language or not powerful enough for the specific task.

6.6. Ignoring Dependencies: the Ignore Function

Sometimes it makes sense to not rebuild a program, even if a dependency file changes. In this case, you would tell SCons specifically to ignore a dependency as follows:

```
hello_obj=Object('hello.c')
hello = Program(hello_obj)
Ignore(hello_obj, 'hello.h')
```

```
% scons -Q hello
cc -c -o hello.o hello.c
cc -o hello hello.o
% scons -Q hello
scons: `hello' is up to date.
% edit hello.h
  [CHANGE THE CONTENTS OF hello.h]
% scons -Q hello
scons: `hello' is up to date.
```

Now, the above example is a little contrived, because it's hard to imagine a real-world situation where you wouldn't want to rebuild `hello` if the `hello.h` file changed. A more realistic example might be if the `hello` program is being built in a directory that is shared between multiple systems that have different copies of the `stdio.h` include file. In that case, SCons would notice the differences between the different systems' copies of `stdio.h` and would rebuild `hello` each time you change systems. You could avoid these rebuilds as follows:

```
hello = Program('hello.c', CPPPATH=['/usr/include'])
Ignore(hello, '/usr/include/stdio.h')
```

`Ignore` can also be used to prevent a generated file from being built by default. This is due to the fact that directories depend on their contents. So to ignore a generated file from the default build, you specify that the directory should ignore the generated file. Note that the file will still be built if the user specifically requests the target on scons command line, or if the file is a dependency of another file which is requested and/or is built by default.

```
hello_obj=Object('hello.c')
hello = Program(hello_obj)
Ignore('.',[hello,hello_obj])
```

```
% scons -Q
scons: `.' is up to date.
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q hello
scons: `hello' is up to date.
```

6.7. Order-Only Dependencies: the `Requires` Function

Occasionally, it may be useful to specify that a certain file or directory must, if necessary, be built or created before some other target is built, but that changes to that file or directory do *not* require that the target itself be rebuilt. Such

a relationship is called an *order-only dependency* because it only affects the order in which things must be built--the dependency before the target--but it is not a strict dependency relationship because the target should not change in response to changes in the dependent file.

For example, suppose that you want to create a file every time you run a build that identifies the time the build was performed, the version number, etc., and which is included in every program that you build. The version file's contents will change every build. If you specify a normal dependency relationship, then every program that depends on that file would be rebuilt every time you ran SCons. For example, we could use some Python code in a SConstruct file to create a new version.c file with a string containing the current date every time we run SCons, and then link a program with the resulting object file by listing version.c in the sources:

```
import time

version_c_text = """
char *date = "%s";
""" % time.ctime(time.time())
open('version.c', 'w').write(version_c_text)

hello = Program(['hello.c', 'version.c'])
```

If we list version.c as an actual source file, though, then the version.o file will get rebuilt every time we run SCons (because the SConstruct file itself changes the contents of version.c) and the hello executable will get re-linked every time (because the version.o file changes):

```
% scons -Q hello
cc -o hello.o -c hello.c
cc -o version.o -c version.c
cc -o hello hello.o version.o
% sleep 1
% scons -Q hello
cc -o version.o -c version.c
cc -o hello hello.o version.o
% sleep 1
% scons -Q hello
cc -o version.o -c version.c
cc -o hello hello.o version.o
```

(Note that for the above example to work, we sleep for one second in between each run, so that the SConstruct file will create a version.c file with a time string that's one second later than the previous run.)

One solution is to use the Requires function to specify that the version.o must be rebuilt before it is used by the link step, but that changes to version.o should not actually cause the hello executable to be re-linked:

```
import time

version_c_text = """
char *date = "%s";
""" % time.ctime(time.time())
open('version.c', 'w').write(version_c_text)

version_obj = Object('version.c')
```

```
hello = Program('hello.c',
                LINKFLAGS = str(version_obj[0]))

Requires(hello, version_obj)
```

Notice that because we can no longer list `version.c` as one of the sources for the `hello` program, we have to find some other way to get it into the link command line. For this example, we're cheating a bit and stuffing the object file name (extracted from `version_obj` list returned by the `Object` call) into the `$LINKFLAGS` variable, because `$LINKFLAGS` is already included in the `$LINKCOM` command line.

With these changes, we get the desired behavior of only re-linking the `hello` executable when the `hello.c` has changed, even though the `version.o` is rebuilt (because the `SConstruct` file still changes the `version.c` contents directly each run):

```
% scons -Q hello
cc -o version.o -c version.c
cc -o hello.o -c hello.c
cc -o hello version.o hello.o
% sleep 1
% scons -Q hello
cc -o version.o -c version.c
scons: `hello' is up to date.
% sleep 1
%     [CHANGE THE CONTENTS OF hello.c]
% scons -Q hello
cc -o version.o -c version.c
cc -o hello.o -c hello.c
cc -o hello version.o hello.o
% sleep 1
% scons -Q hello
cc -o version.o -c version.c
scons: `hello' is up to date.
```

6.8. The `AlwaysBuild` Function

How SCons handles dependencies can also be affected by the `AlwaysBuild` method. When a file is passed to the `AlwaysBuild` method, like so:

```
hello = Program('hello.c')
AlwaysBuild(hello)
```

Then the specified target file (`hello` in our example) will always be considered out-of-date and rebuilt whenever that target file is evaluated while walking the dependency graph:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q
cc -o hello hello.o
```

The `AlwaysBuild` function has a somewhat misleading name, because it does not actually mean the target file will be rebuilt every single time SCons is invoked. Instead, it means that the target will, in fact, be rebuilt whenever the

target file is encountered while evaluating the targets specified on the command line (and their dependencies). So specifying some other target on the command line, a target that does *not* itself depend on the AlwaysBuild target, will still be rebuilt only if it's out-of-date with respect to its dependencies:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q hello.o
scons: `hello.o' is up to date.
```

7 Environments

An *environment* is a collection of values that can affect how a program executes. SCons distinguishes between three different types of environments that can affect the behavior of SCons itself (subject to the configuration in the `SConscript` files), as well as the compilers and other tools it executes:

External Environment
: The *External Environment* is the set of variables in the user's environment at the time the user runs SCons. These variables are not automatically part of an SCons build but are available to be examined if needed. See Section 7.1, “Using Values From the External Environment”, below.

Construction Environment
: A *Construction Environment* is a distinct object created within a `SConscript` file and which contains values that affect how SCons decides what action to use to build a target, and even to define which targets should be built from which sources. One of the most powerful features of SCons is the ability to create multiple construction environments, including the ability to clone a new, customized construction environment from an existing construction environment. See Section 7.2, “Construction Environments”, below.

Execution Environment
: An *Execution Environment* is the values that SCons sets when executing an external command (such as a compiler or linker) to build one or more targets. Note that this is not the same as the external environment (see above). See Section 7.3, “Controlling the Execution Environment for Issued Commands”, below.

Unlike Make, SCons does not automatically copy or import values between different environments (with the exception of explicit clones of construction environments, which inherit the values from their parent). This is a deliberate design choice to make sure that builds are, by default, repeatable regardless of the values in the user's external environment. This avoids a whole class of problems with builds where a developer's local build works because a custom variable setting causes a different compiler or build option to be used, but the checked-in change breaks the official build because it uses different environment variable settings.

Note that the `SConscript` writer can easily arrange for variables to be copied or imported between environments, and this is often very useful (or even downright necessary) to make it easy for developers to customize the build in appropriate ways. The point is *not* that copying variables between different environments is evil and must always be avoided. Instead, it should be up to the implementer of the build system to make conscious choices about how and when to import a variable from one environment to another, making informed decisions about striking the right balance between making the build repeatable on the one hand and convenient to use on the other.

Sidebar: Python Dictionaries

If you're not familiar with the Python programming language, we need to talk a little bit about the Python dictionary data type. A dictionary (also known by terms such as mapping, associative array and key-value store) associates keys with values, such that asking the dict about a key gives you back the associated value and assigning to a key creates the association - either a new setting if the key was unknown, or replacing the previous association if the key was already in the dictionary. Values can be retrieved using *item access* (the key name in square brackets (`[]`)), and dictionaries also provide a method named `get` which responds with a default value, either `None` or a value you supply as the second argument, if the key is not in the dictionary, which avoids failing in that case. The syntax for initializing a dictionary uses curly braces (`{}`). Here are some simple examples (inspired by those in the official Python tutorial) using syntax that indicates interacting with the Python interpreter (`>>>` is the interpreter prompt) - you can try these out:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> print(tel)
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> 'guido' in tel
True
>>> print(tel['jack'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'jack'
>>> print(tel.get('jack'))
None
```

Construction environments are written to behave like a Python dictionary, and the `$ENV` construction variable in a construction environment *is* a Python dictionary. The `os.environ` value that Python uses to make available the external environment is also a dictionary. We will need these concepts in this chapter and throughout the rest of this guide.

7.1. Using Values From the External Environment

The external environment variable settings that the user has in force when executing SCons are available in the Python `os.environ` dictionary. That syntax means the `environ` attribute of the `os` module. In Python, to access the contents of a module you must first `import` it - so you would include the `import os` statement to any `SConscript` file in which you want to use values from the user's external environment.

```
import os

print("Shell is", os.environ['SHELL'])
```

More usefully, you can use the `os.environ` dictionary in your `SConscript` files to initialize construction environments with values from the user's external environment. Read on to the next section for information on how to do this.

7.2. Construction Environments

It is rare that all of the software in a large, complicated system needs to be built exactly the same way. For example, different source files may need different options enabled on the command line, or different executable programs need to be linked with different libraries. SCons accommodates these different build requirements by allowing you to create and configure multiple construction environments that control how the software is built. A construction environment is an object that has a number of associated construction variables, each with a name and a value, just like a dictionary. (A construction environment also has an attached set of `Builder` methods, about which we'll learn more later.)

7.2.1. Creating a Construction Environment: the `Environment` Function

A construction environment is created by the `Environment` method:

```
env = Environment()
```

By default, SCons initializes every new construction environment with a set of construction variables based on the tools that it finds on your system, plus the default set of builder methods necessary for using those tools. The construction variables are initialized with values describing the C compiler, the Fortran compiler, the linker, etc., as well as the command lines to invoke them.

When you initialize a construction environment you can set the values of the environment's construction variables to control how a program is built. For example:

```
env = Environment(CC='gcc', CCFLAGS='-O2')
env.Program('foo.c')
```

The construction environment in this example is still initialized with the same default construction variable values, except that the user has explicitly specified use of the GNU C compiler gcc, and that the `-O2` (optimization level two) flag should be used when compiling the object file. In other words, the explicit initializations of `$CC` and `$CCFLAGS` override the default values in the newly-created construction environment. So a run from this example would look like:

```
% scons -Q
gcc -o foo.o -c -O2 foo.c
gcc -o foo foo.o
```

7.2.2. Fetching Values From a Construction Environment

You can fetch individual values, known as *Construction Variables*, using the same syntax used for accessing individual named items in a Python dictionary:

```
env = Environment()
print("CC is: %s" % env['CC'])
```

```
print("LATEX is: %s" % env.get('LATEX', None))
```

This example `SConstruct` file doesn't contain instructions for building any targets, but because it's still a valid `SConstruct` it will be evaluated and the Python `print` calls will output the values of `$CC` and `$LATEX` for us (remember using the `.get()` method for fetching means we get a default value back, rather than a failure, if the variable is not set):

```
% scons -Q
CC is: cc
LATEX is: None
scons: `.' is up to date.
```

A construction environment is actually an object with associated methods and attributes. If you want to have direct access to only the dictionary of construction variables you can fetch this using the `env.Dictionary` method (although it's rarely necessary to use this method):

```
env = Environment(FOO='foo', BAR='bar')
cvars = env.Dictionary()
for key in ['OBJSUFFIX', 'LIBSUFFIX', 'PROGSUFFIX']:
    print("key = %s, value = %s" % (key, cvars[key]))
```

This `SConstruct` file will print the specified dictionary items for us on POSIX systems as follows:

```
% scons -Q
key = OBJSUFFIX, value = .o
key = LIBSUFFIX, value = .a
key = PROGSUFFIX, value =
scons: `.' is up to date.
```

And on Windows:

```
C:\>scons -Q
key = OBJSUFFIX, value = .obj
key = LIBSUFFIX, value = .lib
key = PROGSUFFIX, value = .exe
scons: `.' is up to date.
```

If you want to loop and print the values of all of the construction variables in a construction environment, the Python code to do that in sorted order might look something like:

```
env = Environment()
for item in sorted(env.Dictionary().items()):
    print("construction variable = '%s', value = '%s'" % item)
```

It should be noted that for the previous example, there is actually a construction environment method that does the same thing more simply, and tries to format the output nicely as well:

```
env = Environment()
print(env.Dump())
```

7.2.3. Expanding Values From a Construction Environment: the subst Method

Another way to get information from a construction environment is to use the `subst` method on a string containing `$` expansions of construction variable names. As a simple example, the example from the previous section that used `env['CC']` to fetch the value of `$CC` could also be written as:

```
env = Environment()
print("CC is: %s" % env.subst('$CC'))
```

One advantage of using `subst` to expand strings is that construction variables in the result get re-expanded until there are no expansions left in the string. So a simple fetch of a value like `$CCCOM`:

```
env = Environment(CCFLAGS='-DFOO')
print("CCCOM is: %s" % env['CCCOM'])
```

Will print the unexpanded value of `$CCCOM`, showing us the construction variables that still need to be expanded:

```
% scons -Q
CCCOM is: $CC $CCFLAGS $CPPFLAGS $_CPPDEFFLAGS $_CPPINCFLAGS -c -o $TARGET $SOURCES
scons: `.' is up to date.
```

Calling the `subst` method on `$CCOM`, however:

```
env = Environment(CCFLAGS='-DFOO')
print("CCCOM is: %s" % env.subst('$CCCOM'))
```

Will recursively expand all of the construction variables prefixed with $ (dollar signs), showing us the final output:

```
% scons -Q
CCCOM is: gcc -DFOO -c -o
scons: `.' is up to date.
```

Note that because we're not expanding this in the context of building something there are no target or source files for `$TARGET` and `$SOURCES` to expand.

7.2.4. Handling Problems With Value Expansion

If a problem occurs when expanding a construction variable, by default it is expanded to `''` (an empty string), and will not cause **scons** to fail.

```
env = Environment()
print("value is: %s"%env.subst( '->$MISSING<-' ))
```

```
% scons -Q
value is: -><-
scons: `.' is up to date.
```

This default behaviour can be changed using the `AllowSubstExceptions` function. When a problem occurs with a variable expansion it generates an exception, and the `AllowSubstExceptions` function controls which of these exceptions are actually fatal and which are allowed to occur safely. By default, `NameError` and `IndexError` are the two exceptions that are allowed to occur: so instead of causing **scons** to fail, these are caught, the variable expanded to `''` and **scons** execution continues. To require that all construction variable names exist, and that indexes out of range are not allowed, call `AllowSubstExceptions` with no extra arguments.

```
AllowSubstExceptions()
env = Environment()
print("value is: %s"%env.subst( '->$MISSING<-' ))
```

```
% scons -Q

scons: *** NameError `name 'MISSING' is not defined' trying to evaluate `$MISSING'
File "/home/my/project/SConstruct", line 3, in <module>
```

This can also be used to allow other exceptions that might occur, most usefully with the `${...}` construction variable syntax. For example, this would allow zero-division to occur in a variable expansion in addition to the default exceptions allowed

```
AllowSubstExceptions(IndexError, NameError, ZeroDivisionError)
env = Environment()
print("value is: %s"%env.subst( '->${1 / 0}<-' ))
```

```
% scons -Q
value is: -><-
scons: `.' is up to date.
```

If `AllowSubstExceptions` is called multiple times, each call completely overwrites the previous list of allowed exceptions.

7.2.5. Controlling the Default Construction Environment: the `DefaultEnvironment` Function

All of the `Builder` functions that we've introduced so far, like `Program` and `Library`, use a construction environment that contains settings for the various compilers and other tools that SCons configures by default, or otherwise knows about and has discovered on your system. If not invoked as methods of a specific construction environment, they use the default construction environment The goal of the default construction environment is to make many configurations "just work" to build software using readily available tools with a minimum of configuration changes.

If needed, you can control the default construction environment by using the `DefaultEnvironment` function to initialize various settings by passing them as keyword arguments:

```
DefaultEnvironment(CC='/usr/local/bin/gcc')
```

When configured as above, all calls to the `Program` or `Object` Builder will build object files with the `/usr/local/bin/gcc` compiler.

The `DefaultEnvironment` function returns the initialized default construction environment object, which can then be manipulated like any other construction environment (note that the default environment works like a singleton - it can have only one instance - so the keyword arguments are processed only on the first call. On any subsequent call the existing object is returned). So the following would be equivalent to the previous example, setting the `$CC` variable to `/usr/local/bin/gcc` but as a separate step after the default construction environment has been initialized:

```
def_env = DefaultEnvironment()
def_env['CC'] = '/usr/local/bin/gcc'
```

One very common use of the `DefaultEnvironment` function is to speed up SCons initialization. As part of trying to make most default configurations "just work," SCons will actually search the local system for installed compilers and other utilities. This search can take time, especially on systems with slow or networked file systems. If you know which compiler(s) and/or other utilities you want to configure, you can control the search that SCons performs by specifying some specific tool modules with which to initialize the default construction environment:

```
def_env = DefaultEnvironment(tools=['gcc', 'gnulink'], CC='/usr/local/bin/gcc')
```

So the above example would tell SCons to explicitly configure the default environment to use its normal GNU Compiler and GNU Linker settings (without having to search for them, or any other utilities for that matter), and specifically to use the compiler found at `/usr/local/bin/gcc`.

7.2.6. Multiple Construction Environments

The real advantage of construction environments is that you can create as many different ones as you need, each tailored to a different way to build some piece of software or other file. If, for example, we need to build one program with the `-O2` flag and another with the `-g` (debug) flag, we would do this like so:

```
opt = Environment(CCFLAGS='-O2')
dbg = Environment(CCFLAGS='-g')

opt.Program('foo', 'foo.c')

dbg.Program('bar', 'bar.c')
```

```
% scons -Q
cc -o bar.o -c -g bar.c
cc -o bar bar.o
cc -o foo.o -c -O2 foo.c
cc -o foo foo.o
```

We can even use multiple construction environments to build multiple versions of a single program. If you do this by simply trying to use the `Program` builder with both environments, though, like this:

```
opt = Environment(CCFLAGS='-O2')
dbg = Environment(CCFLAGS='-g')
```

```
opt.Program('foo', 'foo.c')

dbg.Program('foo', 'foo.c')
```

Then SCons generates the following error:

```
% scons -Q

scons: *** Two environments with different actions were specified for the same target: foo
File "/home/my/project/SConstruct", line 6, in <module>
```

This is because the two `Program` calls have each implicitly told SCons to generate an object file named `foo.o`, one with a `$CCFLAGS` value of `-O2` and one with a `$CCFLAGS` value of `-g`. SCons can't just decide that one of them should take precedence over the other, so it generates the error. To avoid this problem, we must explicitly specify that each environment compile `foo.c` to a separately-named object file using the `Object` builder, like so:

```
opt = Environment(CCFLAGS='-O2')
dbg = Environment(CCFLAGS='-g')

o = opt.Object('foo-opt', 'foo.c')
opt.Program(o)

d = dbg.Object('foo-dbg', 'foo.c')
dbg.Program(d)
```

Notice that each call to the `Object` builder returns a value, an internal SCons object that represents the object file that will be built. We then use that object as input to the `Program` builder. This avoids having to specify explicitly the object file name in multiple places, and makes for a compact, readable `SConstruct` file. Our SCons output then looks like:

```
% scons -Q
cc -o foo-dbg.o -c -g foo.c
cc -o foo-dbg foo-dbg.o
cc -o foo-opt.o -c -O2 foo.c
cc -o foo-opt foo-opt.o
```

7.2.7. Making Copies of Construction Environments: the `Clone` Method

Sometimes you want more than one construction environment to share the same values for one or more variables. Rather than always having to repeat all of the common variables when you create each construction environment, you can use the `env.Clone` method to create a copy of a construction environment.

Like the `Environment` call that creates a construction environment, the `Clone` method takes construction variable assignments, which will override the values in the copied construction environment. For example, suppose we want to use gcc to create three versions of a program, one optimized, one debug, and one with neither. We could do this by creating a "base" construction environment that sets `$CC` to gcc, and then creating two copies, one which sets `$CCFLAGS` for optimization and the other which sets `$CCFLAGS` for debugging:

```
env = Environment(CC='gcc')
```

```
opt = env.Clone(CCFLAGS='-O2')
dbg = env.Clone(CCFLAGS='-g')

env.Program('foo', 'foo.c')

o = opt.Object('foo-opt', 'foo.c')
opt.Program(o)

d = dbg.Object('foo-dbg', 'foo.c')
dbg.Program(d)
```

Then our output would look like:

```
% scons -Q
gcc -o foo.o -c foo.c
gcc -o foo foo.o
gcc -o foo-dbg.o -c -g foo.c
gcc -o foo-dbg foo-dbg.o
gcc -o foo-opt.o -c -O2 foo.c
gcc -o foo-opt foo-opt.o
```

7.2.8. Replacing Values: the Replace Method

You can replace existing construction variable values using the `env.Replace` method:

```
env = Environment(CCFLAGS='-DDEFINE1')
env.Replace(CCFLAGS='-DDEFINE2')
env.Program('foo.c')
```

The replacing value (`-DDEFINE2` in the above example) completely replaces the value in the construction environment:

```
% scons -Q
cc -o foo.o -c -DDEFINE2 foo.c
cc -o foo foo.o
```

You can safely call `Replace` for construction variables that don't exist in the construction environment:

```
env = Environment()
env.Replace(NEW_VARIABLE='xyzzy')
print("NEW_VARIABLE = %s" % env['NEW_VARIABLE'])
```

In this case, the construction variable simply gets added to the construction environment:

```
% scons -Q
NEW_VARIABLE = xyzzy
scons: `.' is up to date.
```

Because the variables aren't expanded until the construction environment is actually used to build the targets, and because SCons function and method calls are order-independent, the last replacement "wins" and is used to build all targets, regardless of the order in which the calls to Replace() are interspersed with calls to builder methods:

```
env = Environment(CCFLAGS='-DDEFINE1')
print("CCFLAGS = %s" % env['CCFLAGS'])
env.Program('foo.c')

env.Replace(CCFLAGS='-DDEFINE2')
print("CCFLAGS = %s" % env['CCFLAGS'])
env.Program('bar.c')
```

The timing of when the replacement actually occurs relative to when the targets get built becomes apparent if we run **scons** without the `-Q` option:

```
% scons
scons: Reading SConscript files ...
CCFLAGS = -DDEFINE1
CCFLAGS = -DDEFINE2
scons: done reading SConscript files.
scons: Building targets ...
cc -o bar.o -c -DDEFINE2 bar.c
cc -o bar bar.o
cc -o foo.o -c -DDEFINE2 foo.c
cc -o foo foo.o
scons: done building targets.
```

Because the replacement occurs while the `SConscript` files are being read, the `$CCFLAGS` variable has already been set to `-DDEFINE2` by the time the `foo.o` target is built, even though the call to the `Replace` method does not occur until later in the `SConscript` file.

7.2.9. Setting Values Only If They're Not Already Defined: the `SetDefault` Method

Sometimes it's useful to be able to specify that a construction variable should be set to a value only if the construction environment does not already have that variable defined You can do this with the `env.SetDefault` method, which behaves similarly to the `setdefault` method of Python dictionary objects:

```
env.SetDefault(SPECIAL_FLAG='-extra-option')
```

This is especially useful when writing your own `Tool` modules to apply variables to construction environments.

7.2.10. Appending to the End of Values: the Append Method

You can append a value to an existing construction variable using the `env.Append` method:

```
env = Environment(CPPDEFINES=['MY_VALUE'])
env.Append(CPPDEFINES=['LAST'])
env.Program('foo.c')
```

Note $CPPDEFINES is the preferred way to set preprocessor defines, as SCons will generate the command line arguments using the correct prefix/suffix for the platform, leaving the usage portable. If you use $CCFLAGS and $SHCCFLAGS, you need to include them in their final form, which is less portable.

```
% scons -Q
cc -o foo.o -c -DMY_VALUE -DLAST foo.c
cc -o foo foo.o
```

If the construction variable doesn't already exist, the Append method will create it:

```
env = Environment()
env.Append(NEW_VARIABLE = 'added')
print("NEW_VARIABLE = %s"%env['NEW_VARIABLE'])
```

Which yields:

```
% scons -Q
NEW_VARIABLE = added
scons: `.' is up to date.
```

Note that the Append function tries to be "smart" about how the new value is appended to the old value. If both are strings, the previous and new strings are simply concatenated. Similarly, if both are lists, the lists are concatenated. If, however, one is a string and the other is a list, the string is added as a new element to the list.

7.2.11. Appending Unique Values: the AppendUnique Method

Sometimes it's useful to add a new value only if the existing construction variable doesn't already contain the value. This can be done using the env.AppendUnique method:

```
env.AppendUnique(CCFLAGS=['-g'])
```

In the above example, the -g would be added only if the $CCFLAGS variable does not already contain a -g value.

7.2.12. Prepending to the Beginning of Values: the Prepend Method

You can prepend a value to the beginning of an existing construction variable using the env.Prepend method:

```
env = Environment(CPPDEFINES=['MY_VALUE'])
env.Prepend(CPPDEFINES=['FIRST'])
env.Program('foo.c')
```

SCons then generates the preprocessor define arguments from CPPDEFINES values with the correct prefix/suffix. For example on Linux or POSIX, the following arguments would be generated: -DFIRST and -DMY_VALUE

```
% scons -Q
```

```
cc -o foo.o -c -DFIRST -DMY_VALUE foo.c
cc -o foo foo.o
```

If the construction variable doesn't already exist, the Prepend method will create it:

```
env = Environment()
env.Prepend(NEW_VARIABLE='added')
print("NEW_VARIABLE = %s" % env['NEW_VARIABLE'])
```

Which yields:

```
% scons -Q
NEW_VARIABLE = added
scons: `.' is up to date.
```

Like the Append function, the Prepend function tries to be "smart" about how the new value is appended to the old value. If both are strings, the previous and new strings are simply concatenated. Similarly, if both are lists, the lists are concatenated. If, however, one is a string and the other is a list, the string is added as a new element to the list.

7.2.13. Prepending Unique Values: the PrependUnique Method

Some times it's useful to add a new value to the beginning of a construction variable only if the existing value doesn't already contain the to-be-added value. This can be done using the env.PrependUnique method:

```
env.PrependUnique(CCFLAGS=['-g'])
```

In the above example, the -g would be added only if the $CCFLAGS variable does not already contain a -g value.

7.2.14. Overriding Construction Variable Settings

Rather than creating a cloned environmant for specific tasks, you can *override* or add construction variables when calling a builder method by passing them as keyword arguments. The values of these overridden or added variables will only be in effect when building that target, and will not affect other parts of the build. For example, if you want to add additional libraries for 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'],
)
```

When overriding this way, the Python keyword arguments in the builder call mean "set to this value". If you want your override to augment an existing value, you have to take some extra steps. Inside the builder call, it is possible to substitute in the existing value by using a string containing the variable name prefaced by a dollar sign ($).

```
env = Environment(CPPDEFINES="FOO")
env.Object(target="foo1.o", source="foo.c")
env.Object(target="foo2.o", source="foo.c", CPPDEFINES="BAR")
env.Object(target="foo3.o", source="foo.c", CPPDEFINES=["BAR", "$CPPDEFINES"])
```

Which yields:

```
% scons -Q
cc -o foo1.o -c -DFOO foo.c
cc -o foo2.o -c -DBAR foo.c
cc -o foo3.o -c -DBAR -DFOO foo.c
```

It is also possible to use the *parse_flags* keyword argument in an override to merge command-line style arguments into the appropriate construction variables. This works like the `env.MergeFlags` method, which will be fully described in the next chapter.

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

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

So when executed:

```
% scons -Q
cc -o hello.o -c -DEBUG -Iinclude hello.c
cc -o hello hello.o -lm
```

Using temporary overrides this way is lighter weight than making a full construction environment, so it can help performance in large projects which have lots of special case values to set. However, keep in mind that this only works well when the targets are unique. Using builder overrides to try to build the same target with different sets of flags or other construction variables will lead to the `scons: *** Two environments with different actions...` error described in Section 7.2.6, “Multiple Construction Environments” above. In this case you will actually want to create separate environments.

7.3. Controlling the Execution Environment for Issued Commands

When SCons builds a target file, it does not execute the commands with the external environment that you used to execute SCons. Instead, it builds an execution environment from the values stored in the `$ENV` construction variable and uses that for executing commands.

The most important ramification of this behavior is that the `PATH` environment variable, which controls where the operating system will look for commands and utilities, will almost certainly not be the same as in the external environment from which you called SCons. This means that SCons might not necessarily find all of the tools that you can successfully execute from the command line.

The default value of the PATH environment variable on a POSIX system is `/usr/local/bin:/opt/bin:/bin:/usr/bin:/snap/bin`. The default value of the PATH environment variable on a Windows system comes from the Windows registry value for the command interpreter. If you want to execute any commands--compilers, linkers, etc.--that are not in these default locations, you need to set the PATH value in the $ENV dictionary in your construction environment.

The simplest way to do this is to initialize explicitly the value when you create the construction environment; this is one way to do that:

```
path = ['/usr/local/bin', '/bin', '/usr/bin']
env = Environment(ENV={'PATH': path})
```

Assigning a dictionary to the $ENV construction variable in this way completely resets the execution environment, so that the only variable that will be set when external commands are executed will be the PATH value. If you want to use the rest of the values in $ENV and only set the value of PATH, you can assign a value only to that variable:

```
env['ENV']['PATH'] = ['/usr/local/bin', '/bin', '/usr/bin']
```

Note that SCons does allow you to define the directories in the PATH in a string with paths separated by the pathname-separator character for your system (`':'` on POSIX systems, `';'` on Windows).

```
env['ENV']['PATH'] = '/usr/local/bin:/bin:/usr/bin'
```

But doing so makes your `SConscript` file less portable, since it will be correct only for the system type that matches the separator. You can use the Python `os.pathsep` for for greater portability - don't worry too much if this Python syntax doesn't make sense since there are other ways available:

```
import os
env['ENV']['PATH'] = os.pathsep.join(['/usr/local/bin', '/bin', '/usr/bin'])
```

7.3.1. Propagating PATH From the External Environment

You may want to propagate the external environment PATH to the execution environment for commands. You do this by initializing the PATH variable with the PATH value from the `os.environ` dictionary, which is Python's way of letting you get at the external environment:

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

Alternatively, you may find it easier to just propagate the entire external environment to the execution environment for commands. This is simpler to code than explicity selecting the PATH value:

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

Either of these will guarantee that SCons will be able to execute any command that you can execute from the command line. The drawback is that the build can behave differently if it's run by people with different PATH values in their environment--for example, if both the /bin and /usr/local/bin directories have different cc commands, then which one will be used to compile programs will depend on which directory is listed first in the user's PATH variable.

7.3.2. Adding to PATH Values in the Execution Environment

One of the most common requirements for manipulating a variable in the execution environment is to add one or more custom directories to a path search variable like PATH on Linux or POSIX systems, or %PATH% on Windows, so that a locally-installed compiler or other utility can be found when SCons tries to execute it to update a target. SCons provides env.PrependENVPath and env.AppendENVPath functions to make adding things to execution variables convenient. You call these functions by specifying the variable to which you want the value added, and then value itself. So to add some /usr/local directories to the $PATH and $LIB variables, you might:

```
env = Environment(ENV=os.environ.copy())
env.PrependENVPath('PATH', '/usr/local/bin')
env.AppendENVPath('LIB', '/usr/local/lib')
```

Note that the added values are strings, and if you want to add multiple directories to a variable like $PATH, you must include the path separator character in the string (: on Linux or POSIX, ; on Windows, or use os.pathsep for portability).

7.4. Using the toolpath for external Tools

7.4.1. The default tool search path

Normally when using a tool from the construction environment, several different search locations are checked by default. This includes the SCons/Tools/ directory that is part of the **scons** distribution and the directory site_scons/site_tools relative to the root SConstruct file.

```
# Builtin tool or tool located within site_tools
env = Environment(tools=['SomeTool'])
env.SomeTool(targets, sources)

# The search locations would include by default
SCons/Tool/SomeTool.py
SCons/Tool/SomeTool/__init__.py
./site_scons/site_tools/SomeTool.py
./site_scons/site_tools/SomeTool/__init__.py
```

7.4.2. Providing an external directory to toolpath

In some cases you may want to specify a different location to search for tools. The Environment function contains an option for this called *toolpath* This can be used to add additional search directories.

```
# Tool located within the toolpath directory option
```

```
env = Environment(
    tools=['SomeTool'],
    toolpath=['/opt/SomeToolPath', '/opt/SomeToolPath2']
)
env.SomeTool(targets, sources)

# The search locations in this example would include:
/opt/SomeToolPath/SomeTool.py
/opt/SomeToolPath/SomeTool/__init__.py
/opt/SomeToolPath2/SomeTool.py
/opt/SomeToolPath2/SomeTool/__init__.py
SCons/Tool/SomeTool.py
SCons/Tool/SomeTool/__init__.py
./site_scons/site_tools/SomeTool.py
./site_scons/site_tools/SomeTool/__init__.py
```

7.4.3. Nested Tools within a toolpath

Since SCons 3.0, a Builder may be located within a sub-directory / sub-package of the toolpath. This is similar to namespacing within Python. With nested or namespaced tools we can use the dot notation to specify a sub-directory that the tool is located under.

```
# namespaced target
env = Environment(
    tools=['SubDir1.SubDir2.SomeTool'],
    toolpath=['/opt/SomeToolPath']
)
env.SomeTool(targets, sources)

# With this example the search locations would include
/opt/SomeToolPath/SubDir1/SubDir2/SomeTool.py
/opt/SomeToolPath/SubDir1/SubDir2/SomeTool/__init__.py
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
```

7.4.4. Using sys.path within the toolpath

If we want to access tools external to **scons** which are findable via `sys.path` (for example, tools installed via Python's **pip** package manager), it is possible to use `sys.path` with the toolpath. One thing to watch out for with this approach is that `sys.path` can sometimes contains paths to `.egg` files instead of directories. So we need to filter those out with this approach.

```
# namespaced target using sys.path within toolpath

searchpaths = []
for item in sys.path:
    if os.path.isdir(item):
        searchpaths.append(item)
```

```
env = Environment(
    tools=['someinstalledpackage.SomeTool'],
    toolpath=searchpaths
)
env.SomeTool(targets, sources)
```

By using sys.path with the toolpath argument and by using the nested syntax we can have **scons** search packages installed via **pip** for Tools.

```
# For Windows based on the python version and install directory, this may be something lik
C:\Python35\Lib\site-packages\someinstalledpackage\SomeTool.py
C:\Python35\Lib\site-packages\someinstalledpackage\SomeTool\__init__.py

# For Linux this could be something like:
/usr/lib/python3/dist-packages/someinstalledpackage/SomeTool.py
/usr/lib/python3/dist-packages/someinstalledpackage/SomeTool/__init__.py
```

7.4.5. Using the PyPackageDir function to add to the toolpath

In some cases you may want to use a tool located within a installed external pip package. This is possible by the use of sys.path with the toolpath. However in that situation you need to provide a prefix to the toolname to indicate where it is located within sys.path.

```
searchpaths = []
for item in sys.path:
    if os.path.isdir(item):
        searchpaths.append(item)
env = Environment(
    tools=['tools_example.subdir1.subdir2.SomeTool'],
    toolpath=searchpaths
)
env.SomeTool(targets, sources)
```

To avoid the use of a prefix within the name of the tool or filtering sys.path for directories, we can use PyPackageDir function to locate the directory of the python package. PyPackageDir returns a Dir object which represents the path of the directory for the python package / module specified as a parameter.

```
# namespaced target using sys.path
env = Environment(
    tools=['SomeTool'],
    toolpath=[PyPackageDir('tools_example.subdir1.subdir2')]
)
env.SomeTool(targets, sources)
```

8 Automatically Putting Command-line Options into their Construction Variables

This chapter describes the `MergeFlags`, `ParseFlags`, and `ParseConfig` methods of a construction environment, as well as the *`parse_flags`* keyword argument to methods that construct environments.

8.1. Merging Options into the Environment: the `MergeFlags` Function

SCons construction environments have an `env.MergeFlags` method that merges values from a passed-in argument into the construction environment If the argument is a dictionary, `MergeFlags` treats each value in the dictionary as a list of options such as one might pass to a command (such as a compiler or linker). `MergeFlags` will not duplicate an option if it already exists in the construction environment variable. If the argument is a string, `MergeFlags` calls the `env.ParseFlags` method to burst it out into a dictionary first, then acts on the result.

`MergeFlags` tries to be intelligent about merging options, knowing that different construction variables may have different needs. When merging options to any variable whose name ends in `PATH`, `MergeFlags` keeps the leftmost occurrence of the option, because in typical lists of directory paths, the first occurrence "wins." When merging options to any other variable name, `MergeFlags` keeps the rightmost occurrence of the option, because in a list of typical command-line options, the last occurrence "wins."

```
env = Environment()
env.Append(CCFLAGS='-option -O3 -O1')
flags = {'CCFLAGS': '-whatever -O3'}
env.MergeFlags(flags)
print(env['CCFLAGS'])
```

```
% scons -Q
['-option', '-O1', '-whatever', '-O3']
scons: `.' is up to date.
```

Note that the default value for `$CCFLAGS` is an internal SCons object which automatically converts the options we specified as a string into a list.

```
env = Environment()
env.Append(CPPPATH=['/include', '/usr/local/include', '/usr/include'])
flags = {'CPPPATH': ['/usr/opt/include', '/usr/local/include']}
env.MergeFlags(flags)
print(env['CPPPATH'])
```

```
% scons -Q
['/include', '/usr/local/include', '/usr/include', '/usr/opt/include']
scons: `.' is up to date.
```

Note that the default value for $CPPPATH is a normal Python list, so we must specify its values as a list in the dictionary we pass to the MergeFlags function.

If MergeFlags is passed anything other than a dictionary, it calls the ParseFlags method to convert it into a dictionary.

```
env = Environment()
env.Append(CCFLAGS='-option -O3 -O1')
env.Append(CPPPATH=['/include', '/usr/local/include', '/usr/include'])
env.MergeFlags('-whatever -I/usr/opt/include -O3 -I/usr/local/include')
print(env['CCFLAGS'])
print(env['CPPPATH'])
```

```
% scons -Q
['-option', '-O1', '-whatever', '-O3']
['/include', '/usr/local/include', '/usr/include', '/usr/opt/include']
scons: `.' is up to date.
```

In the combined example above, ParseFlags has sorted the options into their corresponding variables and returned a dictionary for MergeFlags to apply to the construction variables in the specified construction environment.

8.2. Merging Options While Creating Environment: the *parse_flags* Parameter

It is also possible to merge construction variable values from arguments given to the Environment call itself. If the *parse_flags* keyword argument is given, its value is distributed to construction variables in the new environment in the same way as described for the MergeFlags method. This also works when calling env.Clone, as well as in overrides to builder methods (see Section 7.2.14, “Overriding Construction Variable Settings”).

```
env = Environment(parse_flags="-I/opt/include -L/opt/lib -lfoo")
for k in ('CPPPATH', 'LIBPATH', 'LIBS'):
    print("%s:" % k, env.get(k))
env.Program("f1.c")
```

```
% scons -Q
CPPPATH: ['/opt/include']
LIBPATH: ['/opt/lib']
LIBS: ['foo']
cc -o f1.o -c -I/opt/include f1.c
```

```
cc -o f1 f1.o -L/opt/lib -lfoo
```

8.3. Separating Compile Arguments into their Variables: the ParseFlags Function

SCons has a bewildering array of construction variables for different types of options when building programs. Sometimes you may not know exactly which variable should be used for a particular option.

SCons construction environments have a `env.ParseFlags` method that takes a set of typical command-line options and distributes them into the appropriate construction variables. Historically, it was created to support the `env.ParseConfig` method, so it focuses on options used by the GNU Compiler Collection (GCC) for the C and C++ toolchains.

`ParseFlags` returns a dictionary containing the options distributed into their respective construction variables. Normally, this dictionary would then be passed to `MergeFlags` to merge the options into a construction environment, but the dictionary can be edited if desired to provide additional functionality. (Note that if the flags are not going to be edited, calling `MergeFlags` with the options directly will avoid an additional step.)

```
env = Environment()
d = env.ParseFlags("-I/opt/include -L/opt/lib -lfoo")
for k, v in sorted(d.items()):
    if v:
        print(k, v)
env.MergeFlags(d)
env.Program("f1.c")
```

```
% scons -Q
CPPPATH ['/opt/include']
LIBPATH ['/opt/lib']
LIBS ['foo']
cc -o f1.o -c -I/opt/include f1.c
cc -o f1 f1.o -L/opt/lib -lfoo
```

Note that if the options are limited to generic types like those above, they will be correctly translated for other platform types:

```
C:\>scons -Q
CPPPATH ['/opt/include']
LIBPATH ['/opt/lib']
LIBS ['foo']
cl /Fof1.obj /c f1.c /nologo /I\opt\include
link /nologo /OUT:f1.exe /LIBPATH:\opt\lib foo.lib f1.obj
embedManifestExeCheck(target, source, env)
```

Since the assumption is that the flags are used for the GCC toolchain, unrecognized flags are placed in `$CCFLAGS` so they will be used for both C and C++ compiles:

```
env = Environment()
d = env.ParseFlags("-whatever")
for k, v in sorted(d.items()):
    if v:
        print(k, v)
```

```
env.MergeFlags(d)
env.Program("f1.c")
```

```
% scons -Q
CCFLAGS -whatever
cc -o f1.o -c -whatever f1.c
cc -o f1 f1.o
```

`ParseFlags` will also accept a (recursive) list of strings as input; the list is flattened before the strings are processed:

```
env = Environment()
d = env.ParseFlags(["-I/opt/include", ["-L/opt/lib", "-lfoo"]])
for k, v in sorted(d.items()):
    if v:
        print(k, v)
env.MergeFlags(d)
env.Program("f1.c")
```

```
% scons -Q
CPPPATH ['/opt/include']
LIBPATH ['/opt/lib']
LIBS ['foo']
cc -o f1.o -c -I/opt/include f1.c
cc -o f1 f1.o -L/opt/lib -lfoo
```

If a string begins with a an exclamation mark (`!`, sometimes also called a bang), the string is passed to the shell for execution. The output of the command is then parsed:

```
env = Environment()
d = env.ParseFlags(["!echo -I/opt/include", "!echo -L/opt/lib", "-lfoo"])
for k, v in sorted(d.items()):
    if v:
        print(k, v)
env.MergeFlags(d)
env.Program("f1.c")
```

```
% scons -Q
CPPPATH ['/opt/include']
LIBPATH ['/opt/lib']
LIBS ['foo']
cc -o f1.o -c -I/opt/include f1.c
cc -o f1 f1.o -L/opt/lib -lfoo
```

`ParseFlags` is regularly updated for new options; consult the man page for details about those currently recognized.

8.4. Finding Installed Library Information: the `ParseConfig` Function

Configuring the right options to build programs to work with libraries--especially shared libraries--that are available on POSIX systems can be very complicated. To help this situation, various utilies with names that end in `config` return

the command-line options for the GNU Compiler Collection (GCC) that are needed to use these libraries; for example, the command-line options to use a library named `lib` would be found by calling a utility named `lib-config`.

A more recent convention is that these options are available from the generic `pkg-config` program, which has common framework, error handling, and the like, so that all the package creator has to do is provide the set of strings for his particular package.

SCons construction environments have a `ParseConfig` method that executes a `*config` utility (either `pkg-config` or a more specific utility) and configures the appropriate construction variables in the environment based on the command-line options returned by the specified command.

```
env = Environment()
env['CPPPATH'] = ['/lib/compat']
env.ParseConfig("pkg-config x11 --cflags --libs")
print(env['CPPPATH'])
```

SCons will execute the specified command string, parse the resultant flags, and add the flags to the appropriate environment variables.

```
% scons -Q
['/lib/compat', '/usr/X11/include']
scons: `.' is up to date.
```

In the example above, SCons has added the include directory to `CPPPATH`. (Depending upon what other flags are emitted by the `pkg-config` command, other variables may have been extended as well.)

Note that the options are merged with existing options using the `MergeFlags` method, so that each option only occurs once in the construction variable:

```
env = Environment()
env.ParseConfig("pkg-config x11 --cflags --libs")
env.ParseConfig("pkg-config x11 --cflags --libs")
print(env['CPPPATH'])
```

```
% scons -Q
['/usr/X11/include']
scons: `.' is up to date.
```

9 Controlling Build Output

A key aspect of creating a usable build configuration is providing useful output from the build so its users can readily understand what the build is doing and get information about how to control the build. SCons provides several ways of controlling output from the build configuration to help make the build more useful and understandable.

9.1. Providing Build Help: the `Help` Function

It's often very useful to be able to give users some help that describes the specific targets, build options, etc., that can be used for your build. SCons provides the `Help` function to allow you to specify this help text:

```
Help("""
Type: 'scons program' to build the production program,
      'scons debug' to build the debug version.
""")
```

Optionally, one can specify the append flag:

```
Help("""
Type: 'scons program' to build the production program,
      'scons debug' to build the debug version.
""", append=True)
```

(Note the above use of the Python triple-quote syntax, which comes in very handy for specifying multi-line strings like help text.)

When the `SConstruct` or `SConscript` files contain such a call to the `Help` function, the specified help text will be displayed in response to the SCons `-h` option:

```
% scons -h
scons: Reading SConscript files ...
scons: done reading SConscript files.

Type: 'scons program' to build the production program,
      'scons debug' to build the debug version.

Use scons -H for help about command-line options.
```

The SConscript files may contain multiple calls to the Help function, in which case the specified text(s) will be concatenated when displayed. This allows you to split up the help text across multiple SConscript files. In this situation, the order in which the SConscript files are called will determine the order in which the Help functions are called, which will determine the order in which the various bits of text will get concatenated.

When used with AddOption Help("text", append=False) will clobber any help output associated with AddOption(). To preserve the help output from AddOption(), set append=True.

Another use would be to make the help text conditional on some variable. For example, suppose you only want to display a line about building a Windows-only version of a program when actually run on Windows. The following SConstruct file:

```
env = Environment()

Help("\nType: 'scons program' to build the production program.\n")

if env['PLATFORM'] == 'win32':
    Help("\nType: 'scons windebug' to build the Windows debug version.\n")
```

Will display the complete help text on Windows:

```
C:\>scons -h
scons: Reading SConscript files ...
scons: done reading SConscript files.

Type: 'scons program' to build the production program.

Type: 'scons windebug' to build the Windows debug version.

Use scons -H for help about command-line options.
```

But only show the relevant option on a Linux or UNIX system:

```
% scons -h
scons: Reading SConscript files ...
scons: done reading SConscript files.

Type: 'scons program' to build the production program.

Use scons -H for help about command-line options.
```

If there is no Help text in the SConstruct or SConscript files, SCons will revert to displaying its standard list that describes the SCons command-line options. This list is also always displayed whenever the -H option is used.

9.2. Controlling How SCons Prints Build Commands: the $*COMSTR Variables

Sometimes the commands executed to compile object files or link programs (or build other targets) can get very long, long enough to make it difficult for users to distinguish error messages or other important build output from the commands themselves. All of the default $*COM variables that specify the command lines used to build various types of target files have a corresponding $*COMSTR variable that can be set to an alternative string that will be displayed when the target is built.

For example, suppose you want to have SCons display a "Compiling" message whenever it's compiling an object file, and a "Linking" when it's linking an executable. You could write a SConstruct file that looks like:

```
env = Environment(CCCOMSTR = "Compiling $TARGET",
                  LINKCOMSTR = "Linking $TARGET")
env.Program('foo.c')
```

Which would then yield the output:

```
% scons -Q
Compiling foo.o
Linking foo
```

SCons performs complete variable substitution on $*COMSTR variables, so they have access to all of the standard variables like $TARGET $SOURCES, etc., as well as any construction variables that happen to be configured in the construction environment used to build a specific target.

Of course, sometimes it's still important to be able to see the exact command that SCons will execute to build a target. For example, you may simply need to verify that SCons is configured to supply the right options to the compiler, or a developer may want to cut-and-paste a compile command to add a few options for a custom test.

One common way to give users control over whether or not SCons should print the actual command line or a short, configured summary is to add support for a VERBOSE command-line variable to your SConstruct file. A simple configuration for this might look like:

```
env = Environment()
if ARGUMENTS.get('VERBOSE') != '1':
    env['CCCOMSTR'] = "Compiling $TARGET"
    env['LINKCOMSTR'] = "Linking $TARGET"
env.Program('foo.c')
```

By only setting the appropriate $*COMSTR variables if the user specifies *VERBOSE=1* on the command line, the user has control over how SCons displays these particular command lines:

```
% scons -Q
Compiling foo.o
Linking foo
% scons -Q -c
Removed foo.o
Removed foo
% scons -Q VERBOSE=1
cc -o foo.o -c foo.c
cc -o foo foo.o
```

A gentle reminder here: many of the commands for building come in pairs, depending on whether the intent is to build an object for use in a shared library or not. The command strings mirror this, so it may be necessary to set, for example, both CCCOMSTR and SHCCCOMSTR to get the desired results.

9.3. Providing Build Progress Output: the Progress Function

Another aspect of providing good build output is to give the user feedback about what SCons is doing even when nothing is being built at the moment. This can be especially true for large builds when most of the targets are already up-to-date. Because SCons can take a long time making absolutely sure that every target is, in fact, up-to-date with respect to a lot of dependency files, it can be easy for users to mistakenly conclude that SCons is hung or that there is some other problem with the build.

One way to deal with this perception is to configure SCons to print something to let the user know what it's "thinking about." The Progress function allows you to specify a string that will be printed for every file that SCons is "considering" while it is traversing the dependency graph to decide what targets are or are not up-to-date.

```
Progress('Evaluating $TARGET\n')
Program('f1.c')
Program('f2.c')
```

Note that the Progress function does not arrange for a newline to be printed automatically at the end of the string (as does the Python print function), and we must specify the \n that we want printed at the end of the configured string. This configuration, then, will have SCons print that it is Evaluating each file that it encounters in turn as it traverses the dependency graph:

```
% scons -Q
Evaluating SConstruct
Evaluating f1.c
Evaluating f1.o
cc -o f1.o -c f1.c
Evaluating f1
cc -o f1 f1.o
Evaluating f2.c
Evaluating f2.o
cc -o f2.o -c f2.c
Evaluating f2
cc -o f2 f2.o
Evaluating .
```

Of course, normally you don't want to add all of these additional lines to your build output, as that can make it difficult for the user to find errors or other important messages. A more useful way to display this progress might be to have the file names printed directly to the user's screen, not to the same standard output stream where build output is printed, and to use a carriage return character (\r) so that each file name gets re-printed on the same line. Such a configuration would look like:

```
Progress('$TARGET\r',
         file=open('/dev/tty', 'w'),
         overwrite=True)
Program('f1.c')
Program('f2.c')
```

Note that we also specified the *overwrite=True* argument to the Progress function, which causes SCons to "wipe out" the previous string with space characters before printing the next Progress string. Without the

`overwrite=True` argument, a shorter file name would not overwrite all of the charactes in a longer file name that precedes it, making it difficult to tell what the actual file name is on the output. Also note that we opened up the `/dev/tty` file for direct access (on POSIX) to the user's screen. On Windows, the equivalent would be to open the `con:` file name.

Also, it's important to know that although you can use `$TARGET` to substitute the name of the node in the string, the `Progress` function does *not* perform general variable substitution (because there's not necessarily a construction environment involved in evaluating a node like a source file, for example).

You can also specify a list of strings to the `Progress` function, in which case SCons will display each string in turn. This can be used to implement a "spinner" by having SCons cycle through a sequence of strings:

```
Progress(['-\r', '\\\r', '|\r', '/\r'], interval=5)
Program('f1.c')
Program('f2.c')
```

Note that here we have also used the *`interval=`* keyword argument to have SCons only print a new "spinner" string once every five evaluated nodes. Using an *`interval=`* count, even with strings that use `$TARGET` like our examples above, can be a good way to lessen the work that SCons expends printing `Progress` strings, while still giving the user feedback that indicates SCons is still working on evaluating the build.

Lastly, you can have direct control over how to print each evaluated node by passing a Python function (or other Python callable) to the `Progress` function. Your function will be called for each evaluated node, allowing you to implement more sophisticated logic like adding a counter:

```
screen = open('/dev/tty', 'w')
count = 0
def progress_function(node)
    count += 1
    screen.write('Node %4d: %s\r' % (count, node))

Progress(progress_function)
```

Of course, if you choose, you could completely ignore the `node` argument to the function, and just print a count, or anything else you wish.

(Note that there's an obvious follow-on question here: how would you find the total number of nodes that *will be* evaluated so you can tell the user how close the build is to finishing? Unfortunately, in the general case, there isn't a good way to do that, short of having SCons evaluate its dependency graph twice, first to count the total and the second time to actually build the targets. This would be necessary because you can't know in advance which target(s) the user actually requested to be built. The entire build may consist of thousands of Nodes, for example, but maybe the user specifically requested that only a single object file be built.)

9.4. Printing Detailed Build Status: the `GetBuildFailures` Function

SCons, like most build tools, returns zero status to the shell on success and nonzero status on failure. Sometimes it's useful to give more information about the build status at the end of the run, for instance to print an informative message, send an email, or page the poor slob who broke the build.

SCons provides a GetBuildFailures method that you can use in a python atexit function to get a list of objects describing the actions that failed while attempting to build targets. There can be more than one if you're using -j. Here's a simple 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)
```

The atexit.register call registers print_build_failures as an atexit callback, to be called before SCons exits. When that function is called, it calls GetBuildFailures to fetch the list of failed objects. See the man page for the detailed contents of the returned objects; some of the more useful attributes are .node, .errstr, .filename, and .command. The filename is not necessarily the same file as the node; the node is the target that was being built when the error occurred, while the filenameis the file or dir that actually caused the error. Note: only call GetBuildFailures at the end of the build; calling it at any other time is undefined.

Here is a more complete example showing how to turn each element of GetBuildFailures into a string:

```
# Make the build fail if we pass fail=1 on the command line
if ARGUMENTS.get('fail', 0):
    Command('target', 'source', ['/bin/false'])

def bf_to_str(bf):
    """Convert an element of GetBuildFailures() to a string
    in a useful way."""
    import SCons.Errors
    if bf is None: # unknown targets product None in list
        return '(unknown tgt)'
    elif isinstance(bf, SCons.Errors.StopError):
        return str(bf)
    elif bf.node:
        return str(bf.node) + ': ' + bf.errstr
    elif bf.filename:
        return bf.filename + ': ' + bf.errstr
    return 'unknown failure: ' + bf.errstr
import atexit

def build_status():
    """Convert the build status to a 2-tuple, (status, msg)."""
    from SCons.Script import GetBuildFailures
    bf = GetBuildFailures()
    if bf:
        # bf is normally a list of build failures; if an element is None,
        # it's because of a target that scons doesn't know anything about.
        status = 'failed'
        failures_message = "\n".join(["Failed building %s" % bf_to_str(x)
                            for x in bf if x is not None])
    else:
        # if bf is None, the build completed successfully.
```

```
        status = 'ok'
        failures_message = ''
    return (status, failures_message)

def display_build_status():
    """Display the build status.  Called by atexit.
    Here you could do all kinds of complicated things."""
    status, failures_message = build_status()
    if status == 'failed':
       print("FAILED!!!!")  # could display alert, ring bell, etc.
    elif status == 'ok':
       print("Build succeeded.")
    print(failures_message)

atexit.register(display_build_status)
```

When this runs, you'll see the appropriate output:

```
% scons -Q
scons: `.' is up to date.
Build succeeded.
% scons -Q fail=1
scons: *** [target] Source `source' not found, needed by target `target'.
FAILED!!!!
Failed building target: Source `source' not found, needed by target `target'.
```

10 Controlling a Build From the Command Line

SCons provides a number of ways for you as the writer of the `SConscript` files to give you (and your users) the ability to control the build execution. The arguments that can be specified on the command line are broken down into three types:

Options
: Command-line options always begin with one or two `-` (hyphen) characters. SCons provides ways for you to examine and set options values from within your `SConscript` files, as well as the ability to define your own custom options. See Section 10.1, “Command-Line Options”, below.

Variables
: Any command-line argument containing an `=` (equal sign) is considered a variable setting with the form `variable=`*`value`*. SCons provides direct access to all of the command-line variable settings, the ability to apply command-line variable settings to construction environments, and functions for configuring specific types of variables (Boolean values, path names, etc.) with automatic validation of the specified values. See Section 10.2, “Command-Line `variable=`*`value`* Build Variables”, below.

Targets
: Any command-line argument that is not an option or a variable setting (does not begin with a hyphen and does not contain an equal sign) is considered a target that the you are telling SCons to build. SCons provides access to the list of specified targets, as well as ways to set the default list of targets from within the `SConscript` files. See Section 10.3, “Command-Line Targets”, below.

10.1. Command-Line Options

SCons has many *command-line options* that control its behavior. An SCons command-line option always begins with one or two hyphen (`-`) characters.

10.1.1. Not Having to Specify Command-Line Options Each Time: the SCONSFLAGS Environment Variable

You may find yourself using the same command-line options every time you run SCons. For example, you might find it saves time to specify **`-j 2`** to have SCons run up to two build commands in parallel. To avoid having to type **`-j 2`** by hand every time, you can set the external environment variable `SCONSFLAGS` to a string containing `-j 2`, as well as any other command-line options that you want SCons to always use. `SCONSFLAGS` is an exception to the usual rule that SCons itself avoids looking at environment variables from the shell you are running.

If, for example, you are using a POSIX shell such as **bash** or **zsh** and you always want SCons to use the `-Q` option, you can set the `SCONSFLAGS` environment as follows:

```
% scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
    ... [build output] ...
scons: done building targets.
% export SCONSFLAGS="-Q"
% scons
    ... [build output] ...
```

For csh-style shells on POSIX systems you can set the `SCONSFLAGS` environment variable as follows:

```
$ setenv SCONSFLAGS "-Q"
```

For the Windows command shell (**cmd**) you can set the `SCONSFLAGS` environment variable as follows:

```
C:\Users\foo> set SCONSFLAGS="-Q"
```

To set `SCONSFLAGS` more permanently you can add the setting to the shell's startup file on POSIX systems, and on Windows you can use the `System Properties` control panel applet to select `Environment Variables` and set it there.

10.1.2. Getting Values Set by Command-Line Options: the `GetOption` Function

SCons provides the `GetOption` function to get the values set by the various command-line options.

One use case for `GetOption` is to check whether or not the `-h` or `--help` option has been specified. Normally, SCons does not print its help text until after it has read all of the SConscript files, because it's possible that help text has been added by some subsidiary SConscript file deep in the source tree hierarchy. Of course, reading all of the SConscript files takes extra time. If you know that your configuration does not define any additional help text in subsidiary SConscript files, you can speed up displaying the command-line help by using the `GetOption` function to load the subsidiary SConscript files only if the `-h` or `--help` option has *not* been specified like this:

```
if not GetOption('help'):
    SConscript('src/SConscript', export='env')
```

In general, the string that you pass to the `GetOption` function to fetch the value of a command-line option setting is the same as the "most common" long option name (beginning with two hyphen characters), although there are some exceptions. The list of SCons command-line options and the `GetOption` strings for fetching them, are available in the Section 10.1.4, “Strings for Getting or Setting Values of SCons Command-Line Options” section, below.

`GetOption` can be used to retrieve the values of options defined by calls to `AddOption`. A `GetOption` call must appear after the `AddOption` call for that option. If the `AddOption` call supplied a *`dest`* keyword argument, a string

with that name is what to pass as the argument to GetOption, otherwise it is a (possibly modified) version of the first long option name - see AddOption.

10.1.3. Setting Values of Command-Line Options: the SetOption Function

You can also set the values of SCons command-line options from within the SConscript files by using the SetOption function. The strings that you use to set the values of SCons command-line options are available in the Section 10.1.4, “Strings for Getting or Setting Values of SCons Command-Line Options” section, below.

One use of the SetOption function is to specify a value for the -j or --jobs option, so that you get the improved performance of a parallel build without having to specify the option by hand. A complicating factor is that a good value for the -j option is somewhat system-dependent. One rough guideline is that the more processors your system has, the higher you want to set the -j value, in order to take advantage of the number of CPUs.

For example, suppose the administrators of your development systems have standardized on setting a NUM_CPU environment variable to the number of processors on each system. A little bit of Python code to access the environment variable and the SetOption function provides the right level of flexibility:

```
import os

num_cpu = int(os.environ.get('NUM_CPU', 2))
SetOption('num_jobs', num_cpu)
print("running with -j %s" % GetOption('num_jobs'))
```

The above snippet of code sets the value of the --jobs option to the value specified in the NUM_CPU environment variable. (This is one of the exception cases where the string is spelled differently from the from command-line option. The string for fetching or setting the --jobs value is *num_jobs* for historical reasons.) The code in this example prints the *num_jobs* value for illustrative purposes. It uses a default value of 2 to provide some minimal parallelism even on single-processor systems:

```
% scons -Q
running with -j 2
scons: `.' is up to date.
```

But if the NUM_CPU environment variable is set, then use that for the default number of jobs:

```
% export NUM_CPU="4"
% scons -Q
running with -j 4
scons: `.' is up to date.
```

But any explicit -j or --jobs value you specify on the command line is used first, regardless of whether or not the NUM_CPU environment variable is set:

```
% scons -Q -j 7
running with -j 7
scons: `.' is up to date.
% export NUM_CPU="4"
% scons -Q -j 3
running with -j 3
```

```
scons: `.' is up to date.
```

10.1.4. Strings for Getting or Setting Values of SCons Command-Line Options

The strings that you can pass to the `GetOption` and `SetOption` functions usually correspond to the first long-form option name (that is, name beginning with two hyphen characters: `--`), after replacing any remaining hyphen characters with underscores.

`SetOption` is not currently supported for options added with `AddOption`.

The full list of strings and the variables they correspond to is as follows:

String for `GetOption` and `SetOption`	Command-Line Option(s)
`cache_debug`	`--cache-debug`
`cache_disable`	`--cache-disable`
`cache_force`	`--cache-force`
`cache_show`	`--cache-show`
`clean`	`-c`, `--clean`, `--remove`
`config`	`--config`
`directory`	`-C`, `--directory`
`diskcheck`	`--diskcheck`
`duplicate`	`--duplicate`
`file`	`-f`, `--file`, `--makefile` , `--sconstruct`
`help`	`-h`, `--help`
`ignore_errors`	`--ignore-errors`
`implicit_cache`	`--implicit-cache`
`implicit_deps_changed`	`--implicit-deps-changed`
`implicit_deps_unchanged`	`--implicit-deps-unchanged`
`interactive`	`--interact`, `--interactive`
`keep_going`	`-k`, `--keep-going`
`max_drift`	`--max-drift`
`no_exec`	`-n`, `--no-exec`, `--just-print`, `--dry-run`, `--recon`
`no_site_dir`	`--no-site-dir`
`num_jobs`	`-j`, `--jobs`
`profile_file`	`--profile`
`question`	`-q`, `--question`
`random`	`--random`
`repository`	`-Y`, `--repository`, `--srcdir`
`silent`	`-s`, `--silent`, `--quiet`
`site_dir`	`--site-dir`

String for GetOption and SetOption	Command-Line Option(s)
stack_size	--stack-size
taskmastertrace_file	--taskmastertrace
warn	--warn --warning

10.1.5. Adding Custom Command-Line Options: the AddOption Function

SCons also allows you to define your own command-line options with the AddOption function. The AddOption function takes the same arguments as the add_option method from the standard Python library module *optparse*. [1]

Once you add a custom command-line option with the AddOption function, the value of the option (if any) is immediately available using the standard GetOption function. The argument to GetOption must be the name of the variable which holds the option. If the *dest* keyword argument to AddOption is specified, the value is the variable name. given. If not given, it is the name (without the leading hyphens) of the first long option name given to AddOption after replacing any remaining hyphen characters with underscores, since hyphens are not legal in Python identifier names.

SetOption is not currently supported for options added with AddOption.

One useful example of using this functionality is to provide a --prefix to help describe where to install files:

```
AddOption(
    '--prefix',
    dest='prefix',
    type='string',
    nargs=1,
    action='store',
    metavar='DIR',
    help='installation prefix',
)

env = Environment(PREFIX=GetOption('prefix'))

installed_foo = env.Install('$PREFIX/usr/bin', 'foo.in')
Default(installed_foo)
```

The above code uses the GetOption function to set the $PREFIX construction variable to a value you specify with a command-line option of --prefix. Because $PREFIX expands to a null string if it's not initialized, running SCons without the option of --prefix installs the file in the /usr/bin/ directory:

```
% scons -Q -n
Install file: "foo.in" as "/usr/bin/foo.in"
```

But specifying --prefix=/tmp/install on the command line causes the file to be installed in the /tmp/install/usr/bin/ directory:

```
% scons -Q -n --prefix=/tmp/install
Install file: "foo.in" as "/tmp/install/usr/bin/foo.in"
```

[1] The AddOption function is, in fact, implemented using a subclass of optparse.OptionParser.

Note

Option-arguments separated from long options by whitespace, rather than by an =, cannot be correctly resolved by SCons. While `--input=ARG` is clearly opt followed by arg, for `--input ARG` it is not possible to tell without instructions whether *ARG* is an argument belonging to the *input* option or a positional argument. SCons treats positional arguments as either command-line build options or command-line targets which are made available for use in an `SConscript` (see the immediately following sections for details). Thus, they must be collected before `SConscript` processing takes place. Since `AddOption` calls, which provide the processing instructions to resolve any ambiguity, happen in an `SConscript`, SCons does not know in time for options added this way, and unexpected things happen, such as option-arguments assigned as targets and/or exceptions due to missing option-arguments.

As a result, this usage style should be avoided when invoking **scons**. For single-argument options, use the `--input=ARG` form on the command line. For multiple-argument options (*nargs* greater than one), set *nargs* to one in `AddOption` calls and either: combine the option-arguments into one word with a separator, and parse the result in your own code (see the built-in `--debug` option, which allows specifying multiple arguments as a single comma-separated word, for an example of such usage); or allow the option to be specified multiple times by setting `action='append'`. Both methods can be supported at the same time.

10.2. Command-Line variable=*value* Build Variables

You may want to control various aspects of your build by allowing variable=*value* values to be specified on the command line. For example, suppose you want to be able to build a debug version of a program by running SCons as follows:

```
% scons -Q debug=1
```

SCons provides an `ARGUMENTS` dictionary that stores all of the variable=*value* assignments from the command line. This allows you to modify aspects of your build in response to specifications on the command line. (Note that unless you want to require a variable *always* be specified you probably want to use the Python dictionary `get` method, which allows you to designate a default value to be used if there is no specification on the command line.)

The following code sets the `$CCFLAGS` construction variable in response to the `debug` flag being set in the `ARGUMENTS` dictionary:

```
env = Environment()
debug = ARGUMENTS.get('debug', 0)
if int(debug):
    env.Append(CCFLAGS='-g')
env.Program('prog.c')
```

This results in the `-g` compiler option being used when `debug=1` is used on the command line:

```
% scons -Q debug=0
cc -o prog.o -c prog.c
cc -o prog prog.o
% scons -Q debug=0
scons: `.' is up to date.
```

```
% scons -Q debug=1
cc -o prog.o -c -g prog.c
cc -o prog prog.o
% scons -Q debug=1
scons: `.' is up to date.
```

SCons keeps track of the precise command line used to build each object file, and as a result can determine that the object and executable files need rebuilding when the value of the *debug* argument has changed.

The ARGUMENTS dictionary has two minor drawbacks. First, because it is a dictionary, it can only store one value for each specified keyword, and thus only "remembers" the last setting for each keyword on the command line. This makes the ARGUMENTS dictionary less than ideal if you want to allow specifying multiple values on the command line for a given keyword. Second, it does not preserve the order in which the variable settings were specified, which is a problem if you want the configuration to behave differently in response to the order in which the build variable settings were specified on the command line.

To accomodate these requirements, SCons provides an ARGLIST variable that gives you direct access to variable=*value* settings on the command line, in the exact order they were specified, and without removing any duplicate settings. Each element in the ARGLIST variable is itself a two-element list containing the keyword and the value of the setting, and you must loop through, or otherwise select from, the elements of ARGLIST to process the specific settings you want in whatever way is appropriate for your configuration. For example, the following code lets you add to the CPPDEFINES construction variable by specifying multiple define= settings on the command line:

```
cppdefines = []
for key, value in ARGLIST:
    if key == 'define':
        cppdefines.append(value)
env = Environment(CPPDEFINES=cppdefines)
env.Object('prog.c')
```

Yields the following output:

```
% scons -Q define=FOO
cc -o prog.o -c -DFOO prog.c
% scons -Q define=FOO define=BAR
cc -o prog.o -c -DFOO -DBAR prog.c
```

Note that the ARGLIST and ARGUMENTS variables do not interfere with each other, but rather provide slightly different views into how you specified variable=*value* settings on the command line. You can use both variables in the same SCons configuration. In general, the ARGUMENTS dictionary is more convenient to use, (since you can just fetch variable settings through Python dictionary access), and the ARGLIST list is more flexible (since you can examine the specific order in which the command-line variable settings were given).

10.2.1. Controlling Command-Line Build Variables

Being able to use a command-line build variable like debug=1 is handy, but it can be a chore to write specific Python code to recognize each such variable, check for errors and provide appropriate messages, and apply the values to a construction variable. To help with this, SCons provides a Variables class to define such build variables easily, and a mechanism to apply the build variables to a construction environment. This allows you to control how the build variables affect construction environments.

For example, suppose that you want to set a RELEASE construction variable on the command line whenever the time comes to build a program for release, and that the value of this variable should be added to the command line with the

appropriate define to pass the value to the C compiler. Here's how you might do that by setting the appropriate value in a dictionary for the `$CPPDEFINES` construction variable:

```
vars = Variables(None, ARGUMENTS)
vars.Add('RELEASE', default=0)
env = Environment(variables=vars, CPPDEFINES={'RELEASE_BUILD': '${RELEASE}'})
env.Program(['foo.c', 'bar.c'])
```

This `SConstruct` file first creates a `Variables` object which uses the values from the command-line options dictionary `ARGUMENTS` (the `vars=Variables(None, ARGUMENTS)` call). It then uses the object's `Add` method to indicate that the `RELEASE` variable can be set on the command line, and that if not set the default value is `0`. The newly created `Variables` object is passed to the `Environment` call used to create the construction environment using a `variables` keyword argument. This then allows you to set the `RELEASE` build variable on the command line and have the variable show up in the command line used to build each object from a C source file:

```
% scons -Q RELEASE=1
cc -o bar.o -c -DRELEASE_BUILD=1 bar.c
cc -o foo.o -c -DRELEASE_BUILD=1 foo.c
cc -o foo foo.o bar.o
```

Historical note: In old SCons (prior to 0.98.1), these build variables were known as "command-line build options." At that time, class was named `Options` and the predefined functions to construct options were named `BoolOption`, `EnumOption`, `ListOption`, `PathOption`, `PackageOption` and `AddOptions` (contrast with the current names in Section 10.2.4, “Pre-Defined Build Variable Functions”, below). You may encounter these names in older `SConscript` files, wiki pages, blog entries, StackExchange articles, etc. These old names no longer work, but a mental substitution of “Variable” for “Option” allows the concepts to transfer to current usage models.

10.2.2. Providing Help for Command-Line Build Variables

To make command-line build variables most useful, you ideally want to provide some help text to describe the available variables when the you ask for help (run **`scons -h`**). You can write this text by hand, but SCons provides some assistance. Variables objects provide a `GenerateHelpText` method the generate text that describes the various variables that have been added to it. The default text includes the help string itself plus other information such as allowed values. (The generated text can also be customized by replacing the `FormatVariableHelpText` method). You then pass the output from this method to the `Help` function:

```
vars = Variables(None, ARGUMENTS)
vars.Add('RELEASE', help='Set to 1 to build for release', default=0)
env = Environment(variables=vars)
Help(vars.GenerateHelpText(env))
```

SCons now displays some useful text when the `-h` option is used:

```
% scons -Q -h

RELEASE: Set to 1 to build for release
    default: 0
    actual: 0

Use scons -H for help about command-line options.
```

You can see the help output shows the default value as well as the current actual value of the build variable.

10.2.3. Reading Build Variables From a File

Being able to to specify the value of a build variable on the command line is useful, but can still become tedious if you have to specify the variable every time you run SCons. To make this easier, you can provide customized build variable settings in a local file by providing a file name when the `Variables` object is created:

```
vars = Variables('custom.py')
vars.Add('RELEASE', help='Set to 1 to build for release', default=0)
env = Environment(variables=vars, CPPDEFINES={'RELEASE_BUILD': '${RELEASE}'})
env.Program(['foo.c', 'bar.c'])
Help(vars.GenerateHelpText(env))
```

This then allows you to control the `RELEASE` variable by setting it in the `custom.py` file:

```
RELEASE = 1
```

Note that this file is actually executed like a Python script. Now when you run SCons:

```
% scons -Q
cc -o bar.o -c -DRELEASE_BUILD=1 bar.c
cc -o foo.o -c -DRELEASE_BUILD=1 foo.c
cc -o foo foo.o bar.o
```

And if you change the contents of `custom.py` to:

```
RELEASE = 0
```

The object files are rebuilt appropriately with the new variable:

```
% scons -Q
cc -o bar.o -c -DRELEASE_BUILD=0 bar.c
cc -o foo.o -c -DRELEASE_BUILD=0 foo.c
cc -o foo foo.o bar.o
```

Finally, you can combine both methods with:

```
vars = Variables('custom.py', ARGUMENTS)
```

where values in the option file `custom.py` get overwritten by the ones specified on the command line.

10.2.4. Pre-Defined Build Variable Functions

SCons provides a number of convenience functions that provide ready-made behaviors for various types of command-line build variables. These functions all return a tuple which is ready to be passed to the `Add` or `AddVariables` method call. You are of course free to define your own behaviors as well.

10.2.4.1. True/False Values: the `BoolVariable` Build Variable Function

It is often handy to be able to specify a variable that controls a simple Boolean variable with a `true` or `false` value. It would be even more handy to accomodate different preferences for how to represent `true` or `false` values. The `BoolVariable` function makes it easy to accomodate these common representations of `true` or `false`.

The `BoolVariable` function takes three arguments: the name of the build variable, the default value of the build variable, and the help string for the variable. It then returns appropriate information for passing to the `Add` method of a `Variables` object, like so:

```
vars = Variables('custom.py')
vars.Add(BoolVariable('RELEASE', help='Set to build for release', default=0))
env = Environment(variables=vars, CPPDEFINES={'RELEASE_BUILD': '${RELEASE}'})
env.Program('foo.c')
```

With this build variable in place, the `RELEASE` variable can now be enabled by setting it to the value `yes` or `t`:

```
% scons -Q RELEASE=yes foo.o
cc -o foo.o -c -DRELEASE_BUILD=True foo.c
```

```
% scons -Q RELEASE=t foo.o
cc -o foo.o -c -DRELEASE_BUILD=True foo.c
```

Other values that equate to `true` include `y`, `1`, `on` and `all`.

Conversely, `RELEASE` may now be given a `false` value by setting it to `no` or `f`:

```
% scons -Q RELEASE=no foo.o
cc -o foo.o -c -DRELEASE_BUILD=False foo.c
```

```
% scons -Q RELEASE=f foo.o
cc -o foo.o -c -DRELEASE_BUILD=False foo.c
```

Other values that equate to `false` include `n`, `0`, `off` and `none`.

Lastly, if you try to specify any other value, SCons supplies an appropriate error message:

```
% scons -Q RELEASE=bad_value foo.o

scons: *** Error converting option: RELEASE
Invalid value for boolean option: bad_value
File "/home/my/project/SConstruct", line 3, in <module>
```

10.2.4.2. Single Value From a Selection: the `EnumVariable` Build Variable Function

Suppose that you want to allow setting a `COLOR` variable that selects a background color to be displayed by an application, but that you want to restrict the choices to a specific set of allowed colors. You can set this up quite easily using the `EnumVariable` function, which takes a list of `allowed_values` in addition to the variable name, default value, and help text arguments:

```
vars = Variables('custom.py')
vars.Add(
    EnumVariable(
        'COLOR',
        help='Set background color',
        default='red',
        allowed_values=('red', 'green', 'blue'),
    )
)
env = Environment(variables=vars, CPPDEFINES={'COLOR': '"${COLOR}"'})
env.Program('foo.c')
Help(vars.GenerateHelpText(env))
```

You can now explicitly set the `COLOR` build variable to any of the specified allowed values:

```
% scons -Q COLOR=red foo.o
cc -o foo.o -c -DCOLOR="red" foo.c
% scons -Q COLOR=blue foo.o
cc -o foo.o -c -DCOLOR="blue" foo.c
% scons -Q COLOR=green foo.o
cc -o foo.o -c -DCOLOR="green" foo.c
```

But, importantly, an attempt to set `COLOR` to a value that's not in the list generates an error message:

```
% scons -Q COLOR=magenta foo.o

scons: *** Invalid value for option COLOR: magenta.  Valid values are: ('red', 'green', 'b
File "/home/my/project/SConstruct", line 10, in <module>
```

This example can also serve to further illustrate help generation: the help message here picks up not only the *`help`* text, but augments it with information gathered from *`allowed_values`* and *`default`*:

```
% scons -Q -h

COLOR: Set background color (red|green|blue)
    default: red
    actual: red

Use scons -H for help about command-line options.
```

The `EnumVariable` function also provides a way to map alternate names to allowed values. Suppose, for example, you want to allow the word `navy` to be used as a synonym for `blue`. You do this by adding a `map` dictionary that maps its key values to the desired allowed value:

```
vars = Variables('custom.py')
vars.Add(
    EnumVariable(
        'COLOR',
        help='Set background color',
        default='red',
        allowed_values=('red', 'green', 'blue'),
        map={'navy': 'blue'},
    )
```

```
)
env = Environment(variables=vars, CPPDEFINES={'COLOR': '"${COLOR}"'})
env.Program('foo.c')
```

Now you can supply `navy` on the command line, and SCons translates that into `blue` when it comes time to use the `COLOR` variable to build a target:

```
% scons -Q COLOR=navy foo.o
cc -o foo.o -c -DCOLOR="blue" foo.c
```

By default, when using the `EnumVariable` function, the allowed values are case-sensitive:

```
% scons -Q COLOR=Red foo.o

scons: *** Invalid value for option COLOR: Red.  Valid values are: ('red', 'green', 'blue'
File "/home/my/project/SConstruct", line 10, in <module>
% scons -Q COLOR=BLUE foo.o

scons: *** Invalid value for option COLOR: BLUE.  Valid values are: ('red', 'green', 'blue
File "/home/my/project/SConstruct", line 10, in <module>
% scons -Q COLOR=nAvY foo.o

scons: *** Invalid value for option COLOR: nAvY.  Valid values are: ('red', 'green', 'blue
File "/home/my/project/SConstruct", line 10, in <module>
```

The `EnumVariable` function can take an additional `ignorecase` keyword argument that, when set to `1`, tells SCons to allow case differences when the values are specified:

```
vars = Variables('custom.py')
vars.Add(
    EnumVariable(
        'COLOR',
        help='Set background color',
        default='red',
        allowed_values=('red', 'green', 'blue'),
        map={'navy': 'blue'},
        ignorecase=1,
    )
)
env = Environment(variables=vars, CPPDEFINES={'COLOR': '"${COLOR}"'})
env.Program('foo.c')
```

Which yields the output:

```
% scons -Q COLOR=Red foo.o
cc -o foo.o -c -DCOLOR="Red" foo.c
% scons -Q COLOR=BLUE foo.o
cc -o foo.o -c -DCOLOR="BLUE" foo.c
% scons -Q COLOR=nAvY foo.o
cc -o foo.o -c -DCOLOR="blue" foo.c
% scons -Q COLOR=green foo.o
cc -o foo.o -c -DCOLOR="green" foo.c
```

Notice that an `ignorecase` value of `1` preserves the case-spelling supplied, only ignoring the case for matching. If you want SCons to translate the names into lower-case, regardless of the case used by the user, specify an `ignorecase` value of `2`:

```
vars = Variables('custom.py')
vars.Add(
    EnumVariable(
        'COLOR',
        help='Set background color',
        default='red',
        allowed_values=('red', 'green', 'blue'),
        map={'navy': 'blue'},
        ignorecase=2,
    )
)
env = Environment(variables=vars, CPPDEFINES={'COLOR': '"${COLOR}"'})
env.Program('foo.c')
```

Now SCons uses values of `red`, `green` or `blue` regardless of how those values are spelled on the command line:

```
% scons -Q COLOR=Red foo.o
cc -o foo.o -c -DCOLOR="red" foo.c
% scons -Q COLOR=nAvY foo.o
cc -o foo.o -c -DCOLOR="blue" foo.c
% scons -Q COLOR=GREEN foo.o
cc -o foo.o -c -DCOLOR="green" foo.c
```

10.2.4.3. Multiple Values From a List: the `ListVariable` Build Variable Function

Another way in which you might want to control a build variable is to specify a list of allowed values, of which one or more can be chosen (where `EnumVariable` allows exactly one value to be chosen). SCons provides this through the `ListVariable` function. If, for example, you want to be able to set a `COLORS` variable to one or more of the allowed values:

```
vars = Variables('custom.py')
vars.Add(
    ListVariable(
        'COLORS', help='List of colors', default=0, names=['red', 'green', 'blue']
    )
)
env = Environment(variables=vars, CPPDEFINES={'COLORS': '"${COLORS}"'})
env.Program('foo.c')
```

You can now specify a comma-separated list of allowed values, which get translated into a space-separated list for passing to the build commands:

```
% scons -Q COLORS=red,blue foo.o
cc -o foo.o -c -DCOLORS="red -Dblue" foo.c
% scons -Q COLORS=blue,green,red foo.o
cc -o foo.o -c -DCOLORS="blue -Dgreen -Dred" foo.c
```

In addition, the `ListVariable` function lets you specify explicit keywords of `all` or `none` to select all of the allowed values, or none of them, respectively:

```
% scons -Q COLORS=all foo.o
cc -o foo.o -c -DCOLORS="red -Dgreen -Dblue" foo.c
% scons -Q COLORS=none foo.o
cc -o foo.o -c -DCOLORS="" foo.c
```

And, of course, an illegal value still generates an error message:

```
% scons -Q COLORS=magenta foo.o

scons: *** Error converting option: COLORS
Invalid value(s) for option: magenta
File "/home/my/project/SConstruct", line 7, in <module>
```

You can use this last characteristic as a way to enforce at least one of your valid options being chosen by specifying the valid values with the *`names`* parameter and then giving a value not in that list as the *`default`* parameter - that way if no value is given on the command line, the default is chosen, SCons errors out as this is invalid. The example is, in fact, set up that way by using `0` as the default:

```
% scons -Q foo.o

scons: *** Error converting option: COLORS
Invalid value(s) for option: 0
File "/home/my/project/SConstruct", line 7, in <module>
```

This technique works for `EnumVariable` as well.

10.2.4.4. Path Names: the `PathVariable` Build Variable Function

SCons provides a `PathVariable` function to make it easy to create a build variable to control an expected path name. If, for example, you need to define a preprocessor macro that controls the location of a configuration file:

```
vars = Variables('custom.py')
vars.Add(
    PathVariable(
        'CONFIG', help='Path to configuration file', default='/etc/my_config'
    )
)
env = Environment(variables=vars, CPPDEFINES={'CONFIG_FILE': '"$CONFIG"'})
env.Program('foo.c')
```

This allows you to override the `CONFIG` build variable on the command line as necessary:

```
% scons -Q foo.o
cc -o foo.o -c -DCONFIG_FILE="/etc/my_config" foo.c
% scons -Q CONFIG=/usr/local/etc/other_config foo.o
scons: `foo.o' is up to date.
```

By default, `PathVariable` checks to make sure that the specified path exists and generates an error if it doesn't:

```
% scons -Q CONFIG=/does/not/exist foo.o

scons: *** Path for option CONFIG does not exist: /does/not/exist
```

```
File "/home/my/project/SConstruct", line 7, in <module>
```

`PathVariable` provides a number of methods that you can use to change this behavior. If you want to ensure that any specified paths are, in fact, files and not directories, use the `PathVariable.PathIsFile` method as the validation function:

```
vars = Variables('custom.py')
vars.Add(
    PathVariable(
        'CONFIG',
        help='Path to configuration file',
        default='/etc/my_config',
        validator=PathVariable.PathIsFile,
    )
)
env = Environment(variables=vars, CPPDEFINES={'CONFIG_FILE': '"$CONFIG"'})
env.Program('foo.c')
```

Conversely, to ensure that any specified paths are directories and not files, use the `PathVariable.PathIsDir` method as the validation function:

```
vars = Variables('custom.py')
vars.Add(
    PathVariable(
        'DBDIR',
        help='Path to database directory',
        default='/var/my_dbdir',
        validator=PathVariable.PathIsDir,
    )
)
env = Environment(variables=vars, CPPDEFINES={'DBDIR': '"$DBDIR"'})
env.Program('foo.c')
```

If you want to make sure that any specified paths are directories, and you would like the directory created if it doesn't already exist, use the `PathVariable.PathIsDirCreate` method as the validation function:

```
vars = Variables('custom.py')
vars.Add(
    PathVariable(
        'DBDIR',
        help='Path to database directory',
        default='/var/my_dbdir',
        validator=PathVariable.PathIsDirCreate,
    )
)
env = Environment(variables=vars, CPPDEFINES={'DBDIR': '"$DBDIR"'})
env.Program('foo.c')
```

Lastly, if you don't care whether the path exists, is a file, or a directory, use the `PathVariable.PathAccept` method to accept any path you supply:

```
vars = Variables('custom.py')
vars.Add(
    PathVariable(
        'OUTPUT',
        help='Path to output file or directory',
        default=None,
        validator=PathVariable.PathAccept,
    )
)
env = Environment(variables=vars, CPPDEFINES={'OUTPUT': '"$OUTPUT"'})
env.Program('foo.c')
```

10.2.4.5. Enabled/Disabled Path Names: the `PackageVariable` Build Variable Function

Sometimes you want to give even more control over a path name variable, allowing them to be explicitly enabled or disabled by using `yes` or `no` keywords, in addition to allowing supplying an explicit path name. SCons provides the `PackageVariable` function to support this:

```
vars = Variables("custom.py")
vars.Add(
    PackageVariable("PACKAGE", help="Location package", default="/opt/location")
)
env = Environment(variables=vars, CPPDEFINES={"PACKAGE": '"$PACKAGE"'})
env.Program("foo.c")
```

When the `SConscript` file uses the `PackageVariable` function, you can still use the default or supply an overriding path name, but you can now explicitly set the specified variable to a value that indicates the package should be enabled (in which case the default should be used) or disabled:

```
% scons -Q foo.o
cc -o foo.o -c -DPACKAGE="/opt/location" foo.c
% scons -Q PACKAGE=/usr/local/location foo.o
cc -o foo.o -c -DPACKAGE="/usr/local/location" foo.c
% scons -Q PACKAGE=yes foo.o
cc -o foo.o -c -DPACKAGE="True" foo.c
% scons -Q PACKAGE=no foo.o
cc -o foo.o -c -DPACKAGE="False" foo.c
```

10.2.5. Adding Multiple Command-Line Build Variables at Once

Lastly, SCons provides a way to add multiple build variables to a `Variables` object at once. Instead of having to call the `Add` method multiple times, you can call the `AddVariables` method with the build variables to be added to the object. Each build variable is specified as either a tuple of arguments, or as a call to one of the pre-defined functions for pre-packaged command-line build variables, which returns such a tuple. Note that an individual tuple cannot take keyword arguments in the way that a call to `Add` or one of the build variable functions can. The order of variables given to `AddVariables` does not matter.

```
vars = Variables()
vars.AddVariables(
    ('RELEASE', 'Set to 1 to build for release', 0),
    ('CONFIG', 'Configuration file', '/etc/my_config'),
    BoolVariable('warnings', help='compilation with -Wall and similiar', default=1),
    EnumVariable(
        'debug',
        help='debug output and symbols',
        default='no',
        allowed_values=('yes', 'no', 'full'),
        map={},
        ignorecase=0,
    ),
    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),
)
```

10.2.6. Handling Unknown Command-Line Build Variables: the UnknownVariables Function

Humans, of course, occasionally misspell variable names in their command-line settings. SCons does not generate an error or warning for any unknown variables specified on the command line, because it can not reliably tell whether a given "misspelled" variable is really unknown and a potential problem or not. After all, you might be processing arguments directly using ARGUMENTS or ARGLIST with some Python code in your SConscript file.

If, however, you are using a Variables object to define a specific set of command-line build variables that you expect to be able to set, you may want to provide an error message or warning of your own if a variable setting is specified that is *not* among the defined list of variable names known to the Variables object. You can do this by calling the UnknownVariables method of the Variables object to get the settings Variables did not recognize:

```
vars = Variables(None)
vars.Add('RELEASE', help='Set to 1 to build for release', default=0)
env = Environment(variables=vars, CPPDEFINES={'RELEASE_BUILD': '${RELEASE}'})
unknown = vars.UnknownVariables()
if unknown:
    print("Unknown variables: %s" % " ".join(unknown.keys()))
    Exit(1)
env.Program('foo.c')
```

The UnknownVariables method returns a dictionary containing the keywords and values of any variables specified on the command line that are *not* among the variables known to the Variables object (from having been specified

using the `Variables` object's `Add` method). The example above, checks whether the dictionary returned by `UnknownVariables` is non-empty, and if so prints the Python list containing the names of the unknown variables and then calls the `Exit` function to terminate SCons:

```
% scons -Q NOT_KNOWN=foo
Unknown variables: NOT_KNOWN
```

Of course, you can process the items in the dictionary returned by the `UnknownVariables` function in any way appropriate to your build configuration, including just printing a warning message but not exiting, logging an error somewhere, etc.

Note that you must delay the call of `UnknownVariables` until after you have applied the `Variables` object to a construction environment with the *`variables=`* keyword argument of an `Environment` call: the variables in the object are not fully processed until this has happened.

10.3. Command-Line Targets

10.3.1. Fetching Command-Line Targets: the `COMMAND_LINE_TARGETS` Variable

SCons provides a `COMMAND_LINE_TARGETS` variable that lets you fetch the list of targets that were specified on the command line. You can use the targets to manipulate the build in any way you wish. As a simple example, suppose that you want to print a reminder whenever a specific program is built. You can do this by checking for the target in the `COMMAND_LINE_TARGETS` list:

```
if 'bar' in COMMAND_LINE_TARGETS:
    print("Don't forget to copy `bar' to the archive!")
Default(Program('foo.c'))
Program('bar.c')
```

Now, running SCons with the default target works as usual, but explicity specifying the `bar` target on the command line generates the warning message:

```
% scons -Q
cc -o foo.o -c foo.c
cc -o foo foo.o
% scons -Q bar
Don't forget to copy `bar' to the archive!
cc -o bar.o -c bar.c
cc -o bar bar.o
```

Another practical use for the `COMMAND_LINE_TARGETS` variable might be to speed up a build by only reading certain subsidiary `SConscript` files if a specific target is requested.

10.3.2. Controlling the Default Targets: the `Default` Function

You can control which targets SCons builds by default - that is, when there are no targets specified on the command line. As mentioned previously, SCons normally builds every target in or below the current directory unless you explicitly specify one or more targets on the command line. Sometimes, however, you may want to specify that only certain programs, or programs in certain directories, should be built by default. You do this with the `Default` function:

```
env = Environment()
hello = env.Program('hello.c')
env.Program('goodbye.c')
Default(hello)
```

This SConstruct file knows how to build two programs, hello and goodbye, but only builds the hello program by default:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q
scons: `hello' is up to date.
% scons -Q goodbye
cc -o goodbye.o -c goodbye.c
cc -o goodbye goodbye.o
```

Note that, even when you use the Default function in your SConstruct file, you can still explicitly specify the current directory (.) on the command line to tell SCons to build everything in (or below) the current directory:

```
% scons -Q .
cc -o goodbye.o -c goodbye.c
cc -o goodbye goodbye.o
cc -o hello.o -c hello.c
cc -o hello hello.o
```

You can also call the Default function more than once, in which case each call adds to the list of targets to be built by default:

```
env = Environment()
prog1 = env.Program('prog1.c')
Default(prog1)
prog2 = env.Program('prog2.c')
prog3 = env.Program('prog3.c')
Default(prog3)
```

Or you can specify more than one target in a single call to the Default function:

```
env = Environment()
prog1 = env.Program('prog1.c')
prog2 = env.Program('prog2.c')
prog3 = env.Program('prog3.c')
Default(prog1, prog3)
```

Either of these last two examples build only the prog1 and prog3 programs by default:

```
% scons -Q
cc -o prog1.o -c prog1.c
cc -o prog1 prog1.o
cc -o prog3.o -c prog3.c
```

```
cc -o prog3 prog3.o
% scons -Q .
cc -o prog2.o -c prog2.c
cc -o prog2 prog2.o
```

You can list a directory as an argument to Default:

```
env = Environment()
env.Program(['prog1/main.c', 'prog1/foo.c'])
env.Program(['prog2/main.c', 'prog2/bar.c'])
Default('prog1')
```

In which case only the target(s) in that directory are built by default:

```
% scons -Q
cc -o prog1/foo.o -c prog1/foo.c
cc -o prog1/main.o -c prog1/main.c
cc -o prog1/main prog1/main.o prog1/foo.o
% scons -Q
scons: `prog1' is up to date.
% scons -Q .
cc -o prog2/bar.o -c prog2/bar.c
cc -o prog2/main.o -c prog2/main.c
cc -o prog2/main prog2/main.o prog2/bar.o
```

Lastly, if for some reason you don't want any targets built by default, you can use the Python None variable:

```
env = Environment()
prog1 = env.Program('prog1.c')
prog2 = env.Program('prog2.c')
Default(None)
```

Which would produce build output like:

```
% scons -Q
scons: *** No targets specified and no Default() targets found.  Stop.
Found nothing to build
% scons -Q .
cc -o prog1.o -c prog1.c
cc -o prog1 prog1.o
cc -o prog2.o -c prog2.c
cc -o prog2 prog2.o
```

10.3.2.1. Fetching the List of Default Targets: the DEFAULT_TARGETS Variable

SCons provides a DEFAULT_TARGETS variable that lets you get at the current list of default targets specified by calls to the Default function or method. The DEFAULT_TARGETS variable has two important differences from the COMMAND_LINE_TARGETS variable. First, the DEFAULT_TARGETS variable is a list of internal SCons nodes, so you need to convert the list elements to strings if you want to print them or look for a specific target name. You can do this easily by calling the str on the elements in a list comprehension:

```
prog1 = Program('prog1.c')
Default(prog1)
print("DEFAULT_TARGETS is %s" % [str(t) for t in DEFAULT_TARGETS])
```

(Keep in mind that all of the manipulation of the DEFAULT_TARGETS list takes place during the first phase when SCons is reading up the SConscript files, which is obvious if you leave off the -Q flag when you run SCons:)

```
% scons
scons: Reading SConscript files ...
DEFAULT_TARGETS is ['prog1']
scons: done reading SConscript files.
scons: Building targets ...
cc -o prog1.o -c prog1.c
cc -o prog1 prog1.o
scons: done building targets.
```

Second, the contents of the DEFAULT_TARGETS list changes in response to calls to the Default function, as you can see from the following SConstruct file:

```
prog1 = Program('prog1.c')
Default(prog1)
print("DEFAULT_TARGETS is now %s" % [str(t) for t in DEFAULT_TARGETS])
prog2 = Program('prog2.c')
Default(prog2)
print("DEFAULT_TARGETS is now %s" % [str(t) for t in DEFAULT_TARGETS])
```

Which yields the output:

```
% scons
scons: Reading SConscript files ...
DEFAULT_TARGETS is now ['prog1']
DEFAULT_TARGETS is now ['prog1', 'prog2']
scons: done reading SConscript files.
scons: Building targets ...
cc -o prog1.o -c prog1.c
cc -o prog1 prog1.o
cc -o prog2.o -c prog2.c
cc -o prog2 prog2.o
scons: done building targets.
```

In practice, this simply means that you need to pay attention to the order in which you call the Default function and refer to the DEFAULT_TARGETS list, to make sure that you don't examine the list before you have added the default targets you expect to find in it.

10.3.3. Fetching the List of Build Targets, Regardless of Origin: the BUILD_TARGETS Variable

You have already seen the COMMAND_LINE_TARGETS variable, which contains a list of targets specified on the command line, and the DEFAULT_TARGETS variable, which contains a list of targets specified via calls to the Default method or function. Sometimes, however, you want a list of whatever targets SCons tries to build, regardless of whether the targets came from the command line or a Default call. You could code this up by hand, as follows:

```
if COMMAND_LINE_TARGETS:
    targets = COMMAND_LINE_TARGETS
else:
    targets = DEFAULT_TARGETS
```

SCons, however, provides a convenient BUILD_TARGETS variable that eliminates the need for this by-hand manipulation. Essentially, the BUILD_TARGETS variable contains a list of the command-line targets, if any were specified, and if no command-line targets were specified, it contains a list of the targets specified via the Default method or function.

Because BUILD_TARGETS may contain a list of SCons nodes, you must convert the list elements to strings if you want to print them or look for a specific target name, just like the DEFAULT_TARGETS list:

```
prog1 = Program('prog1.c')
Program('prog2.c')
Default(prog1)
print("BUILD_TARGETS is %s" % [str(t) for t in BUILD_TARGETS])
```

Notice how the value of BUILD_TARGETS changes depending on whether a target is specified on the command line - BUILD_TARGETS takes from DEFAULT_TARGETS only if there are no COMMAND_LINE_TARGETS:

```
% scons -Q
BUILD_TARGETS is ['prog1']
cc -o prog1.o -c prog1.c
cc -o prog1 prog1.o
% scons -Q prog2
BUILD_TARGETS is ['prog2']
cc -o prog2.o -c prog2.c
cc -o prog2 prog2.o
% scons -Q -c .
BUILD_TARGETS is ['.']
Removed prog1.o
Removed prog1
Removed prog2.o
Removed prog2
```

11 Installing Files in Other Directories: the `Install` Builder

Once a program is built, it is often appropriate to install it in another directory for public use. You use the `Install` method to arrange for a program, or any other file, to be copied into a destination directory:

```
env = Environment()
hello = env.Program('hello.c')
env.Install('/usr/bin', hello)
```

Note, however, that installing a file is still considered a type of file "build." This is important when you remember that the default behavior of SCons is to build files in or below the current directory. If, as in the example above, you are installing files in a directory outside of the top-level `SConstruct` file's directory tree, you must specify that directory (or a higher directory, such as `/`) for it to install anything there:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q /usr/bin
Install file: "hello" as "/usr/bin/hello"
```

It can, however, be cumbersome to remember (and type) the specific destination directory in which the program (or other file) should be installed. A call to `Default` can be used to add the directory to the list of default targets, removing the need to type it, but sometimes you don't want to install on every build. This is an area where the `Alias` function comes in handy, allowing you, for example, to create a pseudo-target named `install` that can expand to the specified destination directory:

```
env = Environment()
hello = env.Program('hello.c')
env.Install('/usr/bin', hello)
env.Alias('install', '/usr/bin')
```

This then yields the more natural ability to install the program in its destination as a separate invocation, as follows:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q install
Install file: "hello" as "/usr/bin/hello"
```

11.1. Installing Multiple Files in a Directory

You can install multiple files into a directory simply by calling the `Install` function multiple times:

```
env = Environment()
hello = env.Program('hello.c')
goodbye = env.Program('goodbye.c')
env.Install('/usr/bin', hello)
env.Install('/usr/bin', goodbye)
env.Alias('install', '/usr/bin')
```

Or, more succinctly, listing the multiple input files in a list (just like you can do with any other builder):

```
env = Environment()
hello = env.Program('hello.c')
goodbye = env.Program('goodbye.c')
env.Install('/usr/bin', [hello, goodbye])
env.Alias('install', '/usr/bin')
```

Either of these two examples yields:

```
% scons -Q install
cc -o goodbye.o -c goodbye.c
cc -o goodbye goodbye.o
Install file: "goodbye" as "/usr/bin/goodbye"
cc -o hello.o -c hello.c
cc -o hello hello.o
Install file: "hello" as "/usr/bin/hello"
```

11.2. Installing a File Under a Different Name

The `Install` method preserves the name of the file when it is copied into the destination directory. If you need to change the name of the file when you copy it, use the `InstallAs` function:

```
env = Environment()
hello = env.Program('hello.c')
env.InstallAs('/usr/bin/hello-new', hello)
env.Alias('install', '/usr/bin')
```

This installs the `hello` program with the name `hello-new` as follows:

```
% scons -Q install
cc -o hello.o -c hello.c
```

```
cc -o hello hello.o
Install file: "hello" as "/usr/bin/hello-new"
```

11.3. Installing Multiple Files Under Different Names

If you have multiple files that all need to be installed with different file names, you can either call the `InstallAs` function multiple times, or as a shorthand, you can supply same-length lists for both the target and source arguments:

```
env = Environment()
hello = env.Program('hello.c')
goodbye = env.Program('goodbye.c')
env.InstallAs(['/usr/bin/hello-new',
               '/usr/bin/goodbye-new'],
               [hello, goodbye])
env.Alias('install', '/usr/bin')
```

In this case, the `InstallAs` function loops through both lists simultaneously, and copies each source file into its corresponding target file name:

```
% scons -Q install
cc -o goodbye.o -c goodbye.c
cc -o goodbye goodbye.o
Install file: "goodbye" as "/usr/bin/goodbye-new"
cc -o hello.o -c hello.c
cc -o hello hello.o
Install file: "hello" as "/usr/bin/hello-new"
```

11.4. Installing a Shared Library

If a shared library is created with the `$SHLIBVERSION` variable set, **scons** will create symbolic links as needed based on that variable. To properly install such a library including the symbolic links, use the `InstallVersionedLib` function.

For example, on a Linux system, this instruction:

```
foo =  env.SharedLibrary(target="foo", source="foo.c", SHLIBVERSION="1.2.3")
```

Will produce a shared library `libfoo.so.1.2.3` and symbolic links `libfoo.so` and `libfoo.so.1` which point to `libfoo.so.1.2.3`. You can use the Node returned by the `SharedLibrary` builder in order to install the library and its symbolic links in one go without having to list them individually:

```
env.InstallVersionedLib(target="lib", source=foo)
```

On systems which expect a shared library to be installed both with a name that indicates the version, for run-time resolution, and as a plain name, for link-time resolution, the `InstallVersionedLib` function can be used. Symbolic links appropriate to the type of system will be generated based on symlinks of the source library.

12 Platform-Independent File System Manipulation

SCons provides a number of platform-independent functions, called `factories`, that perform common file system manipulations like copying, moving or deleting files and directories, or making directories. These functions are `factories` because they don't perform the action at the time they're called, they each return an `Action` object that can be executed at the appropriate time.

12.1. Copying Files or Directories: The Copy Factory

Suppose you want to arrange to make a copy of a file, and don't have a suitable pre-existing builder. [1] One way would be to use the `Copy` action factory in conjunction with the `Command` builder:

```
Command("file.out", "file.in", Copy("$TARGET", "$SOURCE"))
```

Notice that the action returned by the `Copy` factory will expand the `$TARGET` and `$SOURCE` strings at the time `file.out` is built, and that the order of the arguments is the same as that of a builder itself--that is, target first, followed by source:

```
% scons -Q
Copy("file.out", "file.in")
```

You can, of course, name a file explicitly instead of using `$TARGET` or `$SOURCE`:

```
Command("file.out", [], Copy("$TARGET", "file.in"))
```

Which executes as:

```
% scons -Q
Copy("file.out", "file.in")
```

[1] Unfortunately, in the early days of SCons design, we used the name `Copy` for the function that returns a copy of the environment, otherwise that would be the logical choice for a Builder that copies a file or directory tree to a target location.

The usefulness of the Copy factory becomes more apparent when you use it in a list of actions passed to the Command builder. For example, suppose you needed to run a file through a utility that only modifies files in-place, and can't "pipe" input to output. One solution is to copy the source file to a temporary file name, run the utility, and then copy the modified temporary file to the target, which the Copy factory makes extremely easy:

```
Command(
    "file.out",
    "file.in",
    action=[
        Copy("tempfile", "$SOURCE"),
        "modify tempfile",
        Copy("$TARGET", "tempfile"),
    ],
)
```

The output then looks like:

```
% scons -Q
Copy("tempfile", "file.in")
modify tempfile
Copy("file.out", "tempfile")
```

The Copy factory has a third optional argument which controls how symlinks are copied.

```
# Symbolic link shallow copied as a new symbolic link:
Command("LinkIn", "LinkOut", Copy("$TARGET", "$SOURCE"[, True]))

# Symbolic link target copied as a file or directory:
Command("LinkIn", "FileOrDirectoryOut", Copy("$TARGET", "$SOURCE", False))
```

12.2. Deleting Files or Directories: The Delete Factory

If you need to delete a file, then the Delete factory can be used in much the same way as the Copy factory. For example, if we want to make sure that the temporary file in our last example doesn't exist before we copy to it, we could add Delete to the beginning of the command list:

```
Command(
    "file.out",
    "file.in",
    action=[
        Delete("tempfile"),
        Copy("tempfile", "$SOURCE"),
        "modify tempfile",
        Copy("$TARGET", "tempfile"),
    ],
)
```

```
```

Which then executes as follows:

```
% scons -Q
Delete("tempfile")
Copy("tempfile", "file.in")
modify tempfile
Copy("file.out", "tempfile")
```

Of course, like all of these Action factories, the Delete factory also expands $TARGET and $SOURCE variables appropriately. For example:

```
Command(
    "file.out",
    "file.in",
    action=[
        Delete("$TARGET"),
        Copy("$TARGET", "$SOURCE"),
    ],
)
```

Executes as:

```
% scons -Q
Delete("file.out")
Copy("file.out", "file.in")
```

Note, however, that you typically don't need to call the Delete factory explicitly in this way; by default, SCons deletes its target(s) for you before executing any action.

One word of caution about using the Delete factory: it has the same variable expansions available as any other factory, including the $SOURCE variable. Specifying Delete("$SOURCE") is not something you usually want to do!

12.3. Moving (Renaming) Files or Directories: The Move Factory

The Move factory allows you to rename a file or directory. For example, if we don't want to copy the temporary file, we could use:

```
Command(
    "file.out",
    "file.in",
    action=[
        Copy("tempfile", "$SOURCE"),
        "modify tempfile",
        Move("$TARGET", "tempfile"),
    ],
)
```

Which would execute as:

```
% scons -Q
Copy("tempfile", "file.in")
modify tempfile
Move("file.out", "tempfile")
```

12.4. Updating the Modification Time of a File: The Touch Factory

If you just need to update the recorded modification time for a file, use the Touch factory:

```
Command(
    "file.out",
    "file.in",
    action=[
        Copy("$TARGET", "$SOURCE"),
        Touch("$TARGET"),
    ]
)
```

Which executes as:

```
% scons -Q
Copy("file.out", "file.in")
Touch("file.out")
```

12.5. Creating a Directory: The Mkdir Factory

If you need to create a directory, use the Mkdir factory. For example, if we need to process a file in a temporary directory in which the processing tool will create other files that we don't care about, you could use:

```
Command(
    "file.out",
    "file.in",
    action=[
        Delete("tempdir"),
        Mkdir("tempdir"),
        Copy("tempdir/${SOURCE.file}", "$SOURCE"),
        "process tempdir",
        Move("$TARGET", "tempdir/output_file"),
        Delete("tempdir"),
    ],
)
```

Which executes as:

```
% scons -Q
Delete("tempdir")
```

```
Mkdir("tempdir")
Copy("tempdir/file.in", "file.in")
process tempdir
Move("file.out", "tempdir/output_file")
scons: *** [file.out] tempdir/output_file: No such file or directory
```

12.6. Changing File or Directory Permissions: The Chmod Factory

To change permissions on a file or directory, use the Chmod factory. The permission argument uses POSIX-style permission bits and should typically be expressed as an octal, not decimal, number:

```
Command(
    "file.out",
    "file.in",
    action=[
        Copy("$TARGET", "$SOURCE"),
        Chmod("$TARGET", 0o755),
    ]
)
```

Which executes:

```
% scons -Q
Copy("file.out", "file.in")
Chmod("file.out", 0755)
```

12.7. Executing an action immediately: the Execute Function

We've been showing you how to use Action factories in the Command function. You can also execute an Action returned by a factory (or actually, any Action) at the time the SConscript file is read by using the Execute function. For example, if we need to make sure that a directory exists before we build any targets,

```
Execute(Mkdir('/tmp/my_temp_directory'))
```

Notice that this will create the directory while the SConscript file is being read:

```
% scons
scons: Reading SConscript files ...
Mkdir("/tmp/my_temp_directory")
scons: done reading SConscript files.
scons: Building targets ...
scons: `.' is up to date.
scons: done building targets.
```

If you're familiar with Python, you may wonder why you would want to use this instead of just calling the native Python os.mkdir() function. The advantage here is that the Mkdir action will behave appropriately if the user

specifies the SCons -n or -q options--that is, it will print the action but not actually make the directory when -n is specified, or make the directory but not print the action when -q is specified.

The Execute function returns the exit status or return value of the underlying action being executed. It will also print an error message if the action fails and returns a non-zero value. SCons will *not*, however, actually stop the build if the action fails. If you want the build to stop in response to a failure in an action called by Execute, you must do so by explicitly checking the return value and calling the Exit function (or a Python equivalent):

```
if Execute(Mkdir('/tmp/my_temp_directory')):
    # A problem occurred while making the temp directory.
    Exit(1)
```

13 Controlling Removal of Targets

There are two occasions when SCons will, by default, remove target files. The first is when SCons determines that an target file needs to be rebuilt and removes the existing version of the target before executing The second is when SCons is invoked with the `-c` option to "clean" a tree of its built targets. These behaviours can be suppressed with the `Precious` and `NoClean` functions, respectively.

13.1. Preventing target removal during build: the `Precious` Function

By default, SCons removes targets before building them. Sometimes, however, this is not what you want. For example, you may want to update a library incrementally, not by having it deleted and then rebuilt from all of the constituent object files. In such cases, you can use the `Precious` method to prevent SCons from removing the target before it is built:

```
env = Environment(RANLIBCOM='')
lib = env.Library('foo', ['f1.c', 'f2.c', 'f3.c'])
env.Precious(lib)
```

Although the output doesn't look any different, SCons does not, in fact, delete the target library before rebuilding it:

```
% scons -Q
cc -o f1.o -c f1.c
cc -o f2.o -c f2.c
cc -o f3.o -c f3.c
ar rc libfoo.a f1.o f2.o f3.o
```

SCons will, however, still delete files marked as `Precious` when the `-c` option is used.

13.2. Preventing target removal during clean: the `NoClean` Function

By default, SCons removes all built targets when invoked with the `-c` option to clean a source tree of built targets. Sometimes, however, this is not what you want. For example, you may want to remove only intermediate generated

files (such as object files), but leave the final targets (the libraries) untouched. In such cases, you can use the NoClean method to prevent SCons from removing a target during a clean:

```
env = Environment(RANLIBCOM='')
lib = env.Library('foo', ['f1.c', 'f2.c', 'f3.c'])
env.NoClean(lib)
```

Notice that the libfoo.a is not listed as a removed file:

```
% scons -Q
cc -o f1.o -c f1.c
cc -o f2.o -c f2.c
cc -o f3.o -c f3.c
ar rc libfoo.a f1.o f2.o f3.o
% scons -c
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Cleaning targets ...
Removed f1.o
Removed f2.o
Removed f3.o
scons: done cleaning targets.
```

13.3. Removing additional files during clean: the Clean Function

There may be additional files that you want removed when the -c option is used, but which SCons doesn't know about because they're not normal target files. For example, perhaps a command you invoke creates a log file as part of building the target file you want. You would like the log file cleaned, but you don't want to have to teach SCons that the command "builds" two files.

You can use the Clean function to arrange for additional files to be removed when the -c option is used. Notice, however, that the Clean function takes two arguments, and the *second* argument is the name of the additional file you want cleaned (foo.log in this example):

```
t = Command('foo.out', 'foo.in', 'build -o $TARGET $SOURCE')
Clean(t, 'foo.log')
```

The first argument is the target with which you want the cleaning of this additional file associated. In the above example, we've used the return value from the Command function, which represents the foo.out target. Now whenever the foo.out target is cleaned by the -c option, the foo.log file will be removed as well:

```
% scons -Q
build -o foo.out foo.in
% scons -Q -c
Removed foo.out
Removed foo.log
```

14 Hierarchical Builds

The source code for large software projects rarely stays in a single directory, but is nearly always divided into a hierarchy of directories. Organizing a large software build using SCons involves creating a hierarchy of build scripts using the `SConscript` function.

14.1. `SConscript` Files

As we've already seen, the build script at the top of the tree is called `SConstruct`. The top-level `SConstruct` file can use the `SConscript` function to include other subsidiary scripts in the build. These subsidiary scripts can, in turn, use the `SConscript` function to include still other scripts in the build. By convention, these subsidiary scripts are usually named `SConscript`. For example, a top-level `SConstruct` file might arrange for four subsidiary scripts to be included in the build as follows:

```
SConscript(['drivers/display/SConscript',
            'drivers/mouse/SConscript',
            'parser/SConscript',
            'utilities/SConscript'])
```

In this case, the `SConstruct` file lists all of the `SConscript` files in the build explicitly. (Note, however, that not every directory in the tree necessarily has an `SConscript` file.) Alternatively, the `drivers` subdirectory might contain an intermediate `SConscript` file, in which case the `SConscript` call in the top-level `SConstruct` file would look like:

```
SConscript(['drivers/SConscript',
            'parser/SConscript',
            'utilities/SConscript'])
```

And the subsidiary `SConscript` file in the `drivers` subdirectory would look like:

```
SConscript(['display/SConscript',
            'mouse/SConscript'])
```

Whether you list all of the `SConscript` files in the top-level `SConstruct` file, or place a subsidiary `SConscript` file in intervening directories, or use some mix of the two schemes, is up to you and the needs of your software.

14.2. Path Names Are Relative to the SConscript Directory

Subsidiary SConscript files make it easy to create a build hierarchy because all of the file and directory names in a subsidiary SConscript files are interpreted relative to the directory in which the SConscript file lives. Typically, this allows the SConscript file containing the instructions to build a target file to live in the same directory as the source files from which the target will be built, making it easy to update how the software is built whenever files are added or deleted (or other changes are made).

For example, suppose we want to build two programs prog1 and prog2 in two separate directories with the same names as the programs. One typical way to do this would be with a top-level SConstruct file like this:

```
SConscript(['prog1/SConscript',
            'prog2/SConscript'])
```

And subsidiary SConscript files that look like this:

```
env = Environment()
env.Program('prog1', ['main.c', 'foo1.c', 'foo2.c'])
```

And this:

```
env = Environment()
env.Program('prog2', ['main.c', 'bar1.c', 'bar2.c'])
```

Then, when we run SCons in the top-level directory, our build looks like:

```
% scons -Q
cc -o prog1/foo1.o -c prog1/foo1.c
cc -o prog1/foo2.o -c prog1/foo2.c
cc -o prog1/main.o -c prog1/main.c
cc -o prog1/prog1 prog1/main.o prog1/foo1.o prog1/foo2.o
cc -o prog2/bar1.o -c prog2/bar1.c
cc -o prog2/bar2.o -c prog2/bar2.c
cc -o prog2/main.o -c prog2/main.c
cc -o prog2/prog2 prog2/main.o prog2/bar1.o prog2/bar2.o
```

Notice the following: First, you can have files with the same names in multiple directories, like main.c in the above example. Second, unlike standard recursive use of Make, SCons stays in the top-level directory (where the SConstruct file lives) and issues commands that use the path names from the top-level directory to the target and source files within the hierarchy.

14.3. Top-Level Path Names in Subsidiary SConscript Files

If you need to use a file from another directory, it's sometimes more convenient to specify the path to a file in another directory from the top-level SConstruct directory, even when you're using that file in a subsidiary SConscript

file in a subdirectory. You can tell SCons to interpret a path name as relative to the top-level `SConstruct` directory, not the local directory of the `SConscript` file, by appending a # (hash mark) to the beginning of the path name:

```
env = Environment()
env.Program('prog', ['main.c', '#lib/foo1.c', 'foo2.c'])
```

In this example, the `lib` directory is directly underneath the top-level `SConstruct` directory. If the above `SConscript` file is in a subdirectory named `src/prog`, the output would look like:

```
% scons -Q
cc -o lib/foo1.o -c lib/foo1.c
cc -o src/prog/foo2.o -c src/prog/foo2.c
cc -o src/prog/main.o -c src/prog/main.c
cc -o src/prog/prog src/prog/main.o lib/foo1.o src/prog/foo2.o
```

(Notice that the `lib/foo1.o` object file is built in the same directory as its source file. See Chapter 15, *Separating Source and Build Trees: Variant Directories*, below, for information about how to build the object file in a different subdirectory.)

14.4. Absolute Path Names

Of course, you can always specify an absolute path name for a file--for example:

```
env = Environment()
env.Program('prog', ['main.c', '/usr/joe/lib/foo1.c', 'foo2.c'])
```

Which, when executed, would yield:

```
% scons -Q
cc -o src/prog/foo2.o -c src/prog/foo2.c
cc -o src/prog/main.o -c src/prog/main.c
cc -o /usr/joe/lib/foo1.o -c /usr/joe/lib/foo1.c
cc -o src/prog/prog src/prog/main.o /usr/joe/lib/foo1.o src/prog/foo2.o
```

(As was the case with top-relative path names, notice that the `/usr/joe/lib/foo1.o` object file is built in the same directory as its source file. See Chapter 15, *Separating Source and Build Trees: Variant Directories*, below, for information about how to build the object file in a different subdirectory.)

14.5. Sharing Environments (and Other Variables) Between `SConscript` Files

In the previous example, each of the subsidiary `SConscript` files created its own construction environment by calling `Environment` separately. This obviously works fine, but if each program must be built with the same construction variables, it's cumbersome and error-prone to initialize separate construction environments in the same way over and over in each subsidiary `SConscript` file.

SCons supports the ability to *export* variables from an `SConscript` file so they can be *imported* by other `SConscript` files, thus allowing you to share common initialized values throughout your build hierarchy.

14.5.1. Exporting Variables

There are two ways to export a variable from an `SConscript` file. The first way is to call the `Export` function. `Export` is pretty flexible - in the simplest form, you pass it a string that represents the name of the variable, and `Export` stores that with its value:

```
env = Environment()
Export('env')
```

You may export more than one variable name at a time:

```
env = Environment()
debug = ARGUMENTS['debug']
Export('env', 'debug')
```

Because a Python identifier cannot contain spaces, `Export` assumes a string containing spaces is is a shortcut for multiple variable names to export and splits it up for you:

```
env = Environment()
debug = ARGUMENTS['debug']
Export('env debug')
```

You can also pass `Export` a dictionary of values. This form allows the opportunity to export a variable from the current scope under a different name - in this example, the value of `foo` is exported under the name `"bar"`:

```
env = Environment()
foo = "FOO"
args = {"env": env, "bar": foo}
Export(args)
```

`Export` will also accept arguments in keyword style. This form adds the ability to create exported variables that have not actually been set locally in the SConscript file. When used this way, the key is the intended variable name, not a string representation as with the other forms:

```
Export(MODE="DEBUG", TARGET="arm")
```

The styles can be mixed, though Python function calling syntax requires all non-keyword arguments to precede any keyword arguments in the call.

The `Export` function adds the variables to a global location from which other `SConscript` files can import. Calls to `Export` are cumulative. When you call `Export` you are actually updating a Python dictionary, so it is fine to export a variable you have already exported, but when doing so, the previous value is lost.

The other way to export is you can specify a list of variables as a second argument to the `SConscript` function call:

```
```

```
SConscript('src/SConscript', 'env')
```

Or (preferably, for readability) using the `exports` keyword argument:

```
SConscript('src/SConscript', exports='env')
```

These calls export the specified variables to only the listed `SConscript` file(s). You may specify more than one `SConscript` file in a list:

```
SConscript(['src1/SConscript',
            'src2/SConscript'], exports='env')
```

This is functionally equivalent to calling the `SConscript` function multiple times with the same `exports` argument, one per `SConscript` file.

14.5.2. Importing Variables

Once a variable has been exported from a calling `SConscript` file, it may be used in other `SConscript` files by calling the `Import` function:

```
Import('env')
env.Program('prog', ['prog.c'])
```

The `Import` call makes the previously defined `env` variable available to the `SConscript` file. Assuming `env` is a construction environment, after import it can be used to build programs, libraries, etc. The use case of passing around a construction environment is extremely common in larger **scons** builds.

Like the `Export` function, the `Import` function can be called with multiple variable names:

```
Import('env', 'debug')
env = env.Clone(DEBUG=debug)
env.Program('prog', ['prog.c'])
```

In this example, we pull in the common construction environment `env`, and use the value of the `debug` variable to make a modified copy by passing that to a `Clone` call.

The `Import` function will (like `Export`) split a string containing white-space into separate variable names:

```
Import('env debug')
env = env.Clone(DEBUG=debug)
env.Program('prog', ['prog.c'])
```

`Import` prefers a local definition to a global one, so that if there is a global export of `foo`, *and* the calling SConscript has exported `foo` to this SConscript, the import will find the `foo` exported to this SConscript.

Lastly, as a special case, you may import all of the variables that have been exported by supplying an asterisk to the Import function:

```
Import('*')
env = env.Clone(DEBUG=debug)
env.Program('prog', ['prog.c'])
```

If you're dealing with a lot of SConscript files, this can be a lot simpler than keeping arbitrary lists of imported variables up to date in each file.

14.5.3. Returning Values From an SConscript File

Sometimes, you would like to be able to use information from a subsidiary SConscript file in some way. For example, suppose that you want to create one library from object files built by several subsidiary SConscript files. You can do this by using the Return function to return values from the subsidiary SConscript files to the calling file. Like Import and Export, Return takes a string representation of the variable name, not the variable name itself.

If, for example, we have two subdirectories foo and bar that should each contribute an object file to a library, what we'd like to be able to do is collect the object files from the subsidiary SConscript calls like this:

```
env = Environment()
Export('env')
objs = []
for subdir in ['foo', 'bar']:
    o = SConscript('%s/SConscript' % subdir)
    objs.append(o)
env.Library('prog', objs)
```

We can do this by using the Return function in the foo/SConscript file like this:

```
Import('env')
obj = env.Object('foo.c')
Return('obj')
```

(The corresponding bar/SConscript file should be pretty obvious.) Then when we run SCons, the object files from the subsidiary subdirectories are all correctly archived in the desired library:

```
% scons -Q
cc -o bar/bar.o -c bar/bar.c
cc -o foo/foo.o -c foo/foo.c
ar rc libprog.a foo/foo.o bar/bar.o
ranlib libprog.a
```

15 Separating Source and Build Trees: Variant Directories

It's often useful to keep any built files completely separate from the source files. Consider if you have a project to build software for a variety of different controller hardware. The boards are able to share a lot of code, so it makes sense to keep them in the same source tree, but certain build options in the source code and header files differ. If you build "Controller A" first, then "Controller B", on the second build everything would have to be rebuilt, because SCons sees that the build instructions differ, and thus the targets that depend on those different instructions are not valid for the current build. Now when you go back and build for "Controller A", things have to be rebuilt from scratch again for the same reason. However, if you can separate the places the output files go, this problem can be avoided. You can even set up to do both builds in one invocation of SCons.

You can enable this separation by creating one or more *variant directory* trees that are used to hold the built objects files, libraries, and executable programs, etc. for a specific flavor, or variant, of build. SCons provides two ways to do this, one through the `SConscript` function that we've already seen, and the second through a more flexible `VariantDir` function.

Historical note: the `VariantDir` function used to be called `BuildDir`, a name which was removed because the SCons functionality differs from a familiar model of a "build directory" implemented by other build systems like GNU Autotools. You might still find references to the old name on the Internet in postings about SCons, but it no longer works.

15.1. Specifying a Variant Directory Tree as Part of an `SConscript` Call

The most straightforward way to establish a variant directory tree relies the fact that the usual way to set up a build hierarchy is to have an SConscript file in the source subdirectory. If you pass a `variant_dir` argument to the `SConscript` function call:

```
SConscript('src/SConscript', variant_dir='build')
```

SCons will then build all of the files in the `build` subdirectory:

```
% ls src
SConscript  hello.c
% scons -Q
```

```
cc -o build/hello.o -c build/hello.c
cc -o build/hello build/hello.o
% ls src
SConscript  hello.c
% ls build
SConscript  hello  hello.c  hello.o
```

No files were built in `src`, they went to `build`. The build output might show a bit of a surprise: the object file `build/hello.o` and the executable file `build/hello` were built in the `build` subdirectory, as expected. But even though our `hello.c` file lives in the `src` subdirectory, SCons has actually compiled a `build/hello.c` file to create the object file, and that file is now seen in `build`.

What's happened is that SCons has *duplicated* the `hello.c` file from the `src` subdirectory to the `build` subdirectory, and built the program from there (it also duplicated `SConscript`). The next section explains why SCons does this.

15.2. Why SCons Duplicates Source Files in a Variant Directory Tree

The important thing to understand is that when you set up a variant directory, SCons performs the build *in that directory*. It turns out it's easiest to ensure where build products end up by just building in place. Since the build is happening in a place different from where the sources are, the most straightforward way to guarantee a correct build is for SCons to copy them there.

The most direct reason to duplicate source files in variant directories is simply that some tools (mostly older versions) are written to only build their output files in the same directory as the source files. In this case, the choices are either to build the output file in the source directory and move it to the variant directory, or to duplicate the source files in the variant directory.

Additionally, relative references between files can cause problems if we don't just duplicate the hierarchy of source files in the variant directory. You can see this at work in use of the C preprocessor `#include` mechanism with double quotes, not angle brackets:

```
#include "file.h"
```

The *de facto* standard behavior for most C compilers in this case is to first look in the same directory as the source file that contains the `#include` line, then to look in the directories in the preprocessor search path. Add to this that the SCons implementation of support for code repositories (described below) means not all of the files will be found in the same directory hierarchy, and the simplest way to make sure that the right include file is found is to duplicate the source files into the variant directory, which provides a correct build regardless of the original location(s) of the source files.

Although source-file duplication guarantees a correct build even in these end-cases, it *can* usually be safely disabled. The next section describes how you can disable the duplication of source files in the variant directory.

15.3. Telling SCons to Not Duplicate Source Files in the Variant Directory Tree

In most cases and with most tool sets, SCons can place its target files in a build subdirectory *without* duplicating the source files and everything will work just fine. You can disable the default SCons behavior by specifying `duplicate=False` when you call the `SConscript` function:

```
SConscript('src/SConscript', variant_dir='build', duplicate=False)
```

When this flag is specified, SCons uses the variant directory like most people expect--that is, the output files are placed in the variant directory while the source files stay in the source directory:

```
% ls src
SConscript
hello.c
% scons -Q
cc -c src/hello.c -o build/hello.o
cc -o build/hello build/hello.o
% ls build
hello
hello.o
```

15.4. The VariantDir Function

Use the VariantDir function to establish that target files should be built in a separate directory from the source files:

```
VariantDir('build', 'src')
env = Environment()
env.Program('build/hello.c')
```

Note that when you're not using an SConscript file in the src subdirectory, you must actually specify that the program must be built from the build/hello.c file that SCons will duplicate in the build subdirectory.

When using the VariantDir function directly, SCons still duplicates the source files in the variant directory by default:

```
% ls src
hello.c
% scons -Q
cc -o build/hello.o -c build/hello.c
cc -o build/hello build/hello.o
% ls build
hello  hello.c  hello.o
```

You can specify the same duplicate=False argument that you can specify for an SConscript call:

```
VariantDir('build', 'src', duplicate=False)
env = Environment()
env.Program('build/hello.c')
```

In which case SCons will disable duplication of the source files:

```
% ls src
hello.c
```

```
% scons -Q
cc -o build/hello.o -c src/hello.c
cc -o build/hello build/hello.o
% ls build
hello  hello.o
```

15.5. Using VariantDir With an SConscript File

Even when using the VariantDir function, it's more natural to use it with a subsidiary SConscript file, because then you don't have to adjust your individual build instructions to use the variant directory path. For example, if the src/SConscript looks like this:

```
env = Environment()
env.Program('hello.c')
```

Then our SConstruct file could look like:

```
VariantDir('build', 'src')
SConscript('build/SConscript')
```

Yielding the following output:

```
% ls src
SConscript  hello.c
% scons -Q
cc -o build/hello.o -c build/hello.c
cc -o build/hello build/hello.o
% ls build
SConscript  hello  hello.c  hello.o
```

Notice that this is completely equivalent to the use of SConscript that we learned about in the previous section.

15.6. Using Glob with VariantDir

The Glob file name pattern matching function works just as usual when using VariantDir. For example, if the src/SConscript looks like this:

```
env = Environment()
env.Program('hello', Glob('*.c'))
```

Then with the same SConstruct file as in the previous section, and source files f1.c and f2.c in src, we would see the following output:

```
% ls src
SConscript  f1.c  f2.c  f2.h
% scons -Q
```

```
cc -o build/f1.o -c build/f1.c
cc -o build/f2.o -c build/f2.c
cc -o build/hello build/f1.o build/f2.o
% ls build
SConscript  f1.c  f1.o  f2.c  f2.h  f2.o  hello
```

The `Glob` function returns Nodes in the `build/` tree, as you'd expect.

15.7. Variant Build Examples

The `variant_dir` keyword argument of the `SConscript` function provides everything we need to show how easy it is to create variant builds using SCons. Suppose, for example, that we want to build a program for both Windows and Linux platforms, but that we want to build it in directory on a network share with separate side-by-side build directories for the Windows and Linux versions of the program. We have to do a little bit of work to construct paths, to make sure unwanted location dependencies don't creep in. The top-relative path reference can be useful here. To avoid writing conditional code based on platform, we can build the *`variant_dir`* path dynamically:

```
platform = ARGUMENTS.get('OS', Platform())

include = "#export/$PLATFORM/include"
lib = "#export/$PLATFORM/lib"
bin = "#export/$PLATFORM/bin"

env = Environment(
    PLATFORM=platform,
    BINDIR=bin,
    INCDIR=include,
    LIBDIR=lib,
    CPPPATH=[include],
    LIBPATH=[lib],
    LIBS='world',
)

Export('env')

env.SConscript('src/SConscript', variant_dir='build/$PLATFORM')
```

This SConstruct file, when run on a Linux system, yields:

```
% scons -Q OS=linux
Install file: "build/linux/world/world.h" as "export/linux/include/world.h"
cc -o build/linux/hello/hello.o -c -Iexport/linux/include build/linux/hello/hello.c
cc -o build/linux/world/world.o -c -Iexport/linux/include build/linux/world/world.c
ar rc build/linux/world/libworld.a build/linux/world/world.o
ranlib build/linux/world/libworld.a
Install file: "build/linux/world/libworld.a" as "export/linux/lib/libworld.a"
cc -o build/linux/hello/hello build/linux/hello/hello.o -Lexport/linux/lib -lworld
Install file: "build/linux/hello/hello" as "export/linux/bin/hello"
```

The same SConstruct file on Windows would build:

```
C:\>scons -Q OS=windows
```

```
Install file: "build/windows/world/world.h" as "export/windows/include/world.h"
cl /Fobuild\windows\hello\hello.obj /c build\windows\hello\hello.c /nologo /Iexport\window
cl /Fobuild\windows\world\world.obj /c build\windows\world\world.c /nologo /Iexport\window
lib /nologo /OUT:build\windows\world\world.lib build\windows\world\world.obj
Install file: "build/windows/world/world.lib" as "export/windows/lib/world.lib"
link /nologo /OUT:build\windows\hello\hello.exe /LIBPATH:export\windows\lib world.lib buil
embedManifestExeCheck(target, source, env)
Install file: "build/windows/hello/hello.exe" as "export/windows/bin/hello.exe"
```

In order to build several variants at once when using the *variant_dir* argument to SConscript, you can call the function repeatedely - this example does so in a loop. Note that the SConscript trick of passing a list of script files, or a list of source directories, does not work with *variant_dir*, SCons allows only a single SConscript to be given if *variant_dir* is used.

```
env = Environment(OS=ARGUMENTS.get('OS'))
for os in ['newell', 'post']:
    SConscript('src/SConscript', variant_dir='build/' + os)
```

16 Building From Code Repositories

Often, a software project will have one or more central repositories, directory trees that contain source code, or derived files, or both. You can eliminate additional unnecessary rebuilds of files by having SCons use files from one or more code repositories to build files in your local build tree.

16.1. The `Repository` Method

It's often useful to allow multiple programmers working on a project to build software from source files and/or derived files that are stored in a centrally-accessible repository, a directory copy of the source code tree. (Note that this is not the sort of repository maintained by a source code management system like BitKeeper, CVS, or Subversion.) You use the `Repository` method to tell SCons to search one or more central code repositories (in order) for any source files and derived files that are not present in the local build tree:

```
env = Environment()
env.Program('hello.c')
Repository('/usr/repository1', '/usr/repository2')
```

Multiple calls to the `Repository` method will simply add repositories to the global list that SCons maintains, with the exception that SCons will automatically eliminate the current directory and any non-existent directories from the list.

16.2. Finding source files in repositories

The above example specifies that SCons will first search for files under the `/usr/repository1` tree and next under the `/usr/repository2` tree. SCons expects that any files it searches for will be found in the same position relative to the top-level directory. In the above example, if the `hello.c` file is not found in the local build tree, SCons will search first for a `/usr/repository1/hello.c` file and then for a `/usr/repository2/hello.c` file to use in its place.

So given the `SConstruct` file above, if the `hello.c` file exists in the local build directory, SCons will rebuild the `hello` program as normal:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
```

If, however, there is no local hello.c file, but one exists in /usr/repository1, SCons will recompile the hello program from the source file it finds in the repository:

```
% scons -Q
cc -o hello.o -c /usr/repository1/hello.c
cc -o hello hello.o
```

And similarly, if there is no local hello.c file and no /usr/repository1/hello.c, but one exists in /usr/repository2:

```
% scons -Q
cc -o hello.o -c /usr/repository2/hello.c
cc -o hello hello.o
```

16.3. Finding #include files in repositories

We've already seen that SCons will scan the contents of a source file for #include file names and realize that targets built from that source file also depend on the #include file(s). For each directory in the $CPPPATH list, SCons will actually search the corresponding directories in any repository trees and establish the correct dependencies on any #include files that it finds in repository directory.

Unless the C compiler also knows about these directories in the repository trees, though, it will be unable to find the #include files. If, for example, the hello.c file in our previous example includes the hello.h in its current directory, and the hello.h only exists in the repository:

```
% scons -Q
cc -o hello.o -c hello.c
hello.c:1: hello.h: No such file or directory
```

In order to inform the C compiler about the repositories, SCons will add appropriate -I flags to the compilation commands for each directory in the $CPPPATH list. So if we add the current directory to the construction environment $CPPPATH like so:

```
env = Environment(CPPPATH = ['.'])
env.Program('hello.c')
Repository('/usr/repository1')
```

Then re-executing SCons yields:

```
% scons -Q
cc -o hello.o -c -I. -I/usr/repository1 hello.c
cc -o hello hello.o
```

The order of the -I options replicates, for the C preprocessor, the same repository-directory search path that SCons uses for its own dependency analysis. If there are multiple repositories and multiple $CPPPATH directories, SCons will add the repository directories to the beginning of each $CPPPATH directory, rapidly multiplying the number of -I flags. If, for example, the $CPPPATH contains three directories (and shorter repository path names!):

```
env = Environment(CPPPATH = ['dir1', 'dir2', 'dir3'])
env.Program('hello.c')
Repository('/r1', '/r2')
```

Then we'll end up with nine `-I` options on the command line, three (for each of the `$CPPPATH` directories) times three (for the local directory plus the two repositories):

```
% scons -Q
cc -o hello.o -c -Idir1 -I/r1/dir1 -I/r2/dir1 -Idir2 -I/r1/dir2 -I/r2/dir2 -Idir3 -I/r1/di
cc -o hello hello.o
```

16.3.1. Limitations on #include files in repositories

SCons relies on the C compiler's `-I` options to control the order in which the preprocessor will search the repository directories for `#include` files. This causes a problem, however, with how the C preprocessor handles `#include` lines with the file name included in double-quotes.

As we've seen, SCons will compile the `hello.c` file from the repository if it doesn't exist in the local directory. If, however, the `hello.c` file in the repository contains a `#include` line with the file name in double quotes:

```
#include "hello.h"
int
main(int argc, char *argv[])
{
    printf(HELLO_MESSAGE);
    return (0);
}
```

Then the C preprocessor will *always* use a `hello.h` file from the repository directory first, even if there is a `hello.h` file in the local directory, despite the fact that the command line specifies `-I` as the first option:

```
% scons -Q
cc -o hello.o -c -I. -I/usr/repository1 /usr/repository1/hello.c
cc -o hello hello.o
```

This behavior of the C preprocessor--always search for a `#include` file in double-quotes first in the same directory as the source file, and only then search the `-I`--can not, in general, be changed. In other words, it's a limitation that must be lived with if you want to use code repositories in this way. There are three ways you can possibly work around this C preprocessor behavior:

1. Some modern versions of C compilers do have an option to disable or control this behavior. If so, add that option to `$CFLAGS` (or `$CXXFLAGS` or both) in your construction environment(s). Make sure the option is used for all construction environments that use C preprocessing!

2. Change all occurrences of `#include "file.h"` to `#include <file.h>`. Use of `#include` with angle brackets does not have the same behavior--the `-I` directories are searched first for `#include` files--which gives SCons direct control over the list of directories the C preprocessor will search.

3. Require that everyone working with compilation from repositories check out and work on entire directories of files, not individual files. (If you use local wrapper scripts around your source code control system's command, you could add logic to enforce this restriction there.

16.4. Finding the SConstruct file in repositories

SCons will also search in repositories for the SConstruct file and any specified SConscript files. This poses a problem, though: how can SCons search a repository tree for an SConstruct file if the SConstruct file itself contains the information about the pathname of the repository? To solve this problem, SCons allows you to specify repository directories on the command line using the -Y option:

```
% scons -Q -Y /usr/repository1 -Y /usr/repository2
```

When looking for source or derived files, SCons will first search the repositories specified on the command line, and then search the repositories specified in the SConstruct or SConscript files.

16.5. Finding derived files in repositories

If a repository contains not only source files, but also derived files (such as object files, libraries, or executables), SCons will perform its normal MD5 signature calculation to decide if a derived file in a repository is up-to-date, or the derived file must be rebuilt in the local build directory. For the SCons signature calculation to work correctly, a repository tree must contain the .sconsign files that SCons uses to keep track of signature information.

Usually, this would be done by a build integrator who would run SCons in the repository to create all of its derived files and .sconsign files, or who would run SCons in a separate build directory and copy the resulting tree to the desired repository:

```
% cd /usr/repository1
% scons -Q
cc -o file1.o -c file1.c
cc -o file2.o -c file2.c
cc -o hello.o -c hello.c
cc -o hello hello.o file1.o file2.o
```

(Note that this is safe even if the SConstruct file lists /usr/repository1 as a repository, because SCons will remove the current build directory from its repository list for that invocation.)

Now, with the repository populated, we only need to create the one local source file we're interested in working with at the moment, and use the -Y option to tell SCons to fetch any other files it needs from the repository:

```
% cd $HOME/build
% edit hello.c
% scons -Q -Y /usr/repository1
cc -c -o hello.o hello.c
cc -o hello hello.o /usr/repository1/file1.o /usr/repository1/file2.o
```

Notice that SCons realizes that it does not need to rebuild local copies file1.o and file2.o files, but instead uses the already-compiled files from the repository.

16.6. Guaranteeing local copies of files

If the repository tree contains the complete results of a build, and we try to build from the repository without any files in our local tree, something moderately surprising happens:

```
% mkdir $HOME/build2
% cd $HOME/build2
% scons -Q -Y /usr/all/repository hello
scons: `hello' is up-to-date.
```

Why does SCons say that the `hello` program is up-to-date when there is no `hello` program in the local build directory? Because the repository (not the local directory) contains the up-to-date `hello` program, and SCons correctly determines that nothing needs to be done to rebuild that up-to-date copy of the file.

There are, however, many times when you want to ensure that a local copy of a file always exists. A packaging or testing script, for example, may assume that certain generated files exist locally. To tell SCons to make a copy of any up-to-date repository file in the local build directory, use the `Local` function:

```
env = Environment()
hello = env.Program('hello.c')
Local(hello)
```

If we then run the same command, SCons will make a local copy of the program from the repository copy, and tell you that it is doing so:

```
% scons -Y /usr/all/repository hello
Local copy of hello from /usr/all/repository/hello
scons: `hello' is up-to-date.
```

(Notice that, because the act of making the local copy is not considered a "build" of the `hello` file, SCons still reports that it is up-to-date.)

17 Extending SCons: Writing Your Own Builders

Although SCons provides many useful methods for building common software products (programs, libraries, documents, etc.), you frequently want to be able to build some other type of file not supported directly by SCons. Fortunately, SCons makes it very easy to define your own `Builder` objects for any custom file types you want to build. (In fact, the SCons interfaces for creating `Builder` objects are flexible enough and easy enough to use that all of the the SCons built-in `Builder` objects are created using the mechanisms described in this section.)

17.1. Writing Builders That Execute External Commands

The simplest `Builder` to create is one that executes an external command. For example, if we want to build an output file by running the contents of the input file through a command named `foobuild`, creating that `Builder` might look like:

```
bld = Builder(action = 'foobuild < $SOURCE > $TARGET')
```

All the above line does is create a free-standing `Builder` object. The next section will show us how to actually use it.

17.2. Attaching a Builder to a Construction Environment

A `Builder` object isn't useful until it's attached to a construction environment so that we can call it to arrange for files to be built. This is done through the `$BUILDERS` construction variable in an environment. The `$BUILDERS` variable is a Python dictionary that maps the names by which you want to call various `Builder` objects to the objects themselves. For example, if we want to call the `Builder` we just defined by the name `Foo`, our `SConstruct` file might look like:

```
bld = Builder(action='foobuild < $SOURCE > $TARGET')
env = Environment(BUILDERS={'Foo': bld})
```

With the `Builder` attached to our construction environment with the name `Foo`, we can now actually call it like so:

```
env.Foo('file.foo', 'file.input')
```

Then when we run SCons it looks like:

```
% scons -Q
foobuild < file.input > file.foo
```

Note, however, that the default `$BUILDERS` variable in a construction environment comes with a default set of `Builder` objects already defined: `Program`, `Library`, etc. And when we explicitly set the `$BUILDERS` variable when we create the construction environment, the default `Builder`s are no longer part of the environment:

```
bld = Builder(action='foobuild < $SOURCE > $TARGET')
env = Environment(BUILDERS={'Foo': bld})
env.Foo('file.foo', 'file.input')
env.Program('hello.c')
```

```
% scons -Q
AttributeError: 'SConsEnvironment' object has no attribute 'Program':
  File "/home/my/project/SConstruct", line 7:
    env.Program('hello.c')
```

To be able to use both our own defined `Builder` objects and the default `Builder` objects in the same construction environment, you can either add to the `$BUILDERS` variable using the `Append` function:

```
env = Environment()
bld = Builder(action='foobuild < $SOURCE > $TARGET')
env.Append(BUILDERS={'Foo': bld})
env.Foo('file.foo', 'file.input')
env.Program('hello.c')
```

Or you can explicitly set the appropriately-named key in the `$BUILDERS` dictionary:

```
env = Environment()
bld = Builder(action='foobuild < $SOURCE > $TARGET')
env['BUILDERS']['Foo'] = bld
env.Foo('file.foo', 'file.input')
env.Program('hello.c')
```

Either way, the same construction environment can then use both the newly-defined `Foo Builder` and the default `Program Builder`:

```
% scons -Q
foobuild < file.input > file.foo
cc -o hello.o -c hello.c
cc -o hello hello.o
```

17.3. Letting SCons Handle The File Suffixes

By supplying additional information when you create a `Builder`, you can let SCons add appropriate file suffixes to the target and/or the source file. For example, rather than having to specify explicitly that you want the `Foo Builder` to build the `file.foo` target file from the `file.input` source file, you can give the `.foo` and `.input` suffixes to the `Builder`, making for more compact and readable calls to the `Foo Builder`:

```
bld = Builder(
    action='foobuild < $SOURCE > $TARGET',
    suffix='.foo',
    src_suffix='.input',
)
env = Environment(BUILDERS={'Foo': bld})
env.Foo('file1')
env.Foo('file2')
```

```
% scons -Q
foobuild < file1.input > file1.foo
foobuild < file2.input > file2.foo
```

You can also supply a `prefix` keyword argument if it's appropriate to have SCons append a prefix to the beginning of target file names.

17.4. Builders That Execute Python Functions

In SCons, you don't have to call an external command to build a file. You can, instead, define a Python function that a `Builder` object can invoke to build your target file (or files). Such a builder function definition looks like:

```
def build_function(target, source, env):
    # Code to build "target" from "source"
    return None
```

The arguments of a builder function are:

target
: A list of Node objects representing the target or targets to be built by this function. The file names of these target(s) may be extracted using the Python `str` function.

source
: A list of Node objects representing the sources to be used by this function to build the targets. The file names of these source(s) may be extracted using the Python `str` function.

env
: The construction environment used for building the target(s). The function may use any of the environment's construction variables in any way to affect how it builds the targets.

The function will be constructed as a SCons FunctionAction and must return a `0` or `None` value if the target(s) are built successfully. The function may raise an exception or return any non-zero value to indicate that the build is unsuccessful. For more information on Actions see the Action Objects section of the man page.

Once you've defined the Python function that will build your target file, defining a Builder object for it is as simple as specifying the name of the function, instead of an external command, as the Builder's action argument:

```
def build_function(target, source, env):
    # Code to build "target" from "source"
    return None

bld = Builder(
    action=build_function,
    suffix='.foo',
    src_suffix='.input',
)
env = Environment(BUILDERS={'Foo': bld})
env.Foo('file')
```

And notice that the output changes slightly, reflecting the fact that a Python function, not an external command, is now called to build the target file:

```
% scons -Q
build_function(["file.foo"], ["file.input"])
```

17.5. Builders That Create Actions Using a Generator

SCons Builder objects can create an action "on the fly" by using a function called a generator. (Note: this is not the same thing as a Python generator function described in PEP 255 [https://www.python.org/dev/peps/pep-0255/]) This provides a great deal of flexibility to construct just the right list of commands to build your target. A generator looks like:

```
def generate_actions(source, target, env, for_signature):
    return 'foobuild < %s > %s' % (target[0], source[0])
```

The arguments of a generator are:

source
: A list of Node objects representing the sources to be built by the command or other action generated by this function. The file names of these source(s) may be extracted using the Python str function.

target
: A list of Node objects representing the target or targets to be built by the command or other action generated by this function. The file names of these target(s) may be extracted using the Python str function.

env
: The construction environment used for building the target(s). The generator may use any of the environment's construction variables in any way to determine what command or other action to return.

for_signature
: A flag that specifies whether the generator is being called to contribute to a build signature, as opposed to actually executing the command.

The generator must return a command string or other action that will be used to build the specified target(s) from the specified source(s).

Once you've defined a generator, you create a Builder to use it by specifying the generator keyword argument instead of action.

```
def generate_actions(source, target, env, for_signature):
    return 'foobuild < %s > %s' % (source[0], target[0])

bld = Builder(
    generator=generate_actions,
    suffix='.foo',
    src_suffix='.input',
)
env = Environment(BUILDERS={'Foo': bld})
env.Foo('file')
```

```
% scons -Q
foobuild < file.input > file.foo
```

Note that it's illegal to specify both an action and a generator for a Builder.

17.6. Builders That Modify the Target or Source Lists Using an Emitter

SCons supports the ability for a Builder to modify the lists of target(s) from the specified source(s). You do this by defining an emitter function that takes as its arguments the list of the targets passed to the builder, the list of the sources passed to the builder, and the construction environment. The emitter function should return the modified lists of targets that should be built and sources from which the targets will be built.

For example, suppose you want to define a Builder that always calls a **foobuild** program, and you want to automatically add a new target file named new_target and a new source file named new_source whenever it's called. The SConstruct file might look like this:

```
def modify_targets(target, source, env):
    target.append('new_target')
    source.append('new_source')
    return target, source

bld = Builder(
    action='foobuild $TARGETS - $SOURCES',
    suffix='.foo',
    src_suffix='.input',
    emitter=modify_targets,
)
env = Environment(BUILDERS={'Foo': bld})
env.Foo('file')
```

And would yield the following output:

```
% scons -Q
foobuild file.foo new_target - file.input new_source
```

One very flexible thing that you can do is use a construction variable to specify different emitter functions for different construction variable. To do this, specify a string containing a construction variable expansion as the emitter when you call the `Builder` function, and set that construction variable to the desired emitter function in different construction environments:

```
bld = Builder(
    action='./my_command $SOURCES > $TARGET',
    suffix='.foo',
    src_suffix='.input',
    emitter='$MY_EMITTER',
)

def modify1(target, source, env):
    return target, source + ['modify1.in']

def modify2(target, source, env):
    return target, source + ['modify2.in']

env1 = Environment(BUILDERS={'Foo': bld}, MY_EMITTER=modify1)
env2 = Environment(BUILDERS={'Foo': bld}, MY_EMITTER=modify2)
env1.Foo('file1')
env2.Foo('file2')
```

In this example, the `modify1.in` and `modify2.in` files get added to the source lists of the different commands:

```
% scons -Q
./my_command file1.input modify1.in > file1.foo
./my_command file2.input modify2.in > file2.foo
```

17.7. Modifying a Builder by adding an Emitter

Defining an emitter to work with a custom Builder is a powerful concept, but sometimes all you really want is to be able to use an existing builder but change its concept of what targets are created. In this case, trying to recreate the logic of an existing Builder to supply a special emitter can be a lot of work. The typical case for this is when you want to use a compiler flag that causes additional files to be generated. For example the GNU linker accepts an option `-Map` which outputs a link map to the file specified by the option's argument. If this option is just supplied to the build, SCons will not consider the link map file a tracked target, which has various undesirable efffects.

To help with this, SCons provides construction variables which correspond to a few standard builders: `$PROGEMITTER` for `Program`; `$LIBEMITTER` for `Library`; `$SHLIBEMITTER` for `SharedLibrary` and `$LDMODULEEMITTER` for `LoadableModule`;. Adding an emitter to one of these will cause it to be invoked in addition to any existing emitter for the corresponding builder.

This example adds map creation as a linker flag, and modifies the standard `Program` emitter to know that map generation is a side-effect:

```
env = Environment()
map_filename = "${TARGET.name}.map"
```

```
def map_emitter(target, source, env):
    target.append(map_filename)
    return target, source

env.Append(LINKFLAGS="-Wl,-Map={},--cref".format(map_filename))
env.Append(PROGEMITTER=map_emitter)
env.Program('hello.c')
```

If you run this example, adding an option to tell SCons to dump some information about the dependencies it knows, it shows the map file option in use, and that SCons indeed knows about the map file, it's not just a silent side effect of the compiler:

```
% scons -Q --tree=prune
cc -o hello.o -c hello.c
cc -o hello -Wl,-Map=hello.map,--cref hello.o
+-.
  +-SConstruct
  +-hello
  | +-hello.o
  |   +-hello.c
  +-hello.c
  +-hello.map
  | +-[hello.o]
  +-[hello.o]
```

17.8. Where To Put Your Custom Builders and Tools

The `site_scons` directories give you a place to put Python modules and packages that you can import into your `SConscript` files (`site_scons`), add-on tools that can integrate into SCons (`site_scons/site_tools`), and a `site_scons/site_init.py` file that gets read before any `SConstruct` or `SConscript` file, allowing you to change SCons's default behavior.

Each system type (Windows, Mac, Linux, etc.) searches a canonical set of directories for `site_scons`; see the man page for details. The top-level SConstruct's `site_scons` dir is always searched last, and its dir is placed first in the tool path so it overrides all others.

If you get a tool from somewhere (the SCons wiki or a third party, for instance) and you'd like to use it in your project, a `site_scons` dir is the simplest place to put it. Tools come in two flavors; either a Python function that operates on an `Environment` or a Python module or package containing two functions, `exists()` and `generate()`.

A single-function Tool can just be included in your `site_scons/site_init.py` file where it will be parsed and made available for use. For instance, you could have a `site_scons/site_init.py` file like this:

```
def TOOL_ADD_HEADER(env):
    """A Tool to add a header from $HEADER to the source file"""
    add_header = Builder(
        action=['echo "$HEADER" > $TARGET', 'cat $SOURCE >> $TARGET']
    )
    env.Append(BUILDERS={'AddHeader': add_header})
```

```
    env['HEADER'] = ''  # set default value
```

and a SConstruct like this:

```
# Use TOOL_ADD_HEADER from site_scons/site_init.py
env=Environment(tools=['default', TOOL_ADD_HEADER], HEADER="=====")
env.AddHeader('tgt', 'src')
```

The TOOL_ADD_HEADER tool method will be called to add the AddHeader tool to the environment.

A more full-fledged tool with exists() and generate() methods can be installed either as a module in the file site_scons/site_tools/toolname.py or as a package in the directory site_scons/site_tools/toolname. In the case of using a package, the exists() and generate() are in the file site_scons/site_tools/toolname/__init__.py. (In all the above case toolname is replaced by the name of the tool.) Since site_scons/site_tools is automatically added to the head of the tool search path, any tool found there will be available to all environments. Furthermore, a tool found there will override a built-in tool of the same name, so if you need to change the behavior of a built-in tool, site_scons gives you the hook you need.

Many people have a library of utility Python functions they'd like to include in SConscripts; just put that module in site_scons/my_utils.py or any valid Python module name of your choice. For instance you can do something like this in site_scons/my_utils.py to add build_id and MakeWorkDir functions:

```
from SCons.Script import *  # for Execute and Mkdir

def build_id():
    """Return a build ID (stub version)"""
    return "100"

def MakeWorkDir(workdir):
    """Create the specified dir immediately"""
    Execute(Mkdir(workdir))
```

And then in your SConscript or any sub-SConscript anywhere in your build, you can import my_utils and use it:

```
import my_utils
print("build_id=" + my_utils.build_id())
my_utils.MakeWorkDir('/tmp/work')
```

Note that although you can put this library in site_scons/site_init.py, it is no better there than site_scons/my_utils.py since you still have to import that module into your SConscript. Also note that in order to refer to objects in the SCons namespace such as Environment or Mkdir or Execute in any file other than a SConstruct or SConscript you always need to do

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

This is true in modules in site_scons such as site_scons/site_init.py as well.

You can use any of the user- or machine-wide site dirs such as `~/.scons/site_scons` instead of `./site_scons`, or use the `--site-dir` option to point to your own dir. `site_init.py` and `site_tools` will be located under that dir. To avoid using a `site_scons` dir at all, even if it exists, use the `--no-site-dir` option.

18 Not Writing a Builder: the Command Builder

Creating a `Builder` and attaching it to a construction environment allows for a lot of flexibility when you want to re-use actions to build multiple files of the same type. This can, however, be cumbersome if you only need to execute one specific command to build a single file (or group of files). For these situations, SCons supports a `Command` builder that arranges for a specific action to be executed to build a specific file or files. This looks a lot like the other builders (like `Program`, `Object`, etc.), but takes as an additional argument the command to be executed to build the file:

```
env = Environment()
env.Command('foo.out', 'foo.in', "sed 's/x/y/' < $SOURCE > $TARGET")
```

When executed, SCons runs the specified command, substituting `$SOURCE` and `$TARGET` as expected:

```
% scons -Q
sed 's/x/y/' < foo.in > foo.out
```

This is often more convenient than creating a `Builder` object and adding it to the `$BUILDERS` variable of a construction environment.

Note that the action you specify to the `Command` `Builder` can be any legal SCons `Action`, such as a Python function:

```
env = Environment()

def build(target, source, env):
    # Whatever it takes to build
    return None

env.Command('foo.out', 'foo.in', build)
```

Which executes as follows:

```
% scons -Q
build(["foo.out"], ["foo.in"])
```

Note that `$SOURCE` and `$TARGET` are expanded in the source and target as well, so you can write:

```
env.Command('${SOURCE.basename}.out', 'foo.in', build)
```

which does the same thing as the previous example, but allows you to avoid repeating yourself.

It may be helpful to use the *action* keyword to specify the action, is this makes things more clear to the reader:

```
env.Command('${SOURCE.basename}.out', 'foo.in', action=build)
```

The method described in Section 9.2, “Controlling How SCons Prints Build Commands: the $*COMSTR Variables” for controlling build output works well when used with pre-defined builders which have pre-defined *COMSTR variables for that purpose, but that is not the case when calling Command, where SCons has no specific knowledge of the action ahead of time. If the action argument to Command is not already an Action object, it will construct one for you with suitable defaults, which include a message based on the type of action. However, you can also construct the Action object yourself to pass to Command, which gives you much more control. Here's an evolution of the example from above showing this approach:

```
env = Environment()

def build(target, source, env):
    # Whatever it takes to build
    return None

act = Action(build, cmdstr="Building ${TARGET}")
env.Command('foo.out', 'foo.in', action=act)
```

Which executes as follows:

```
% scons -Q
Building foo.out
```

19 Extending SCons: Pseudo-Builders and the AddMethod function

The `AddMethod` function is used to add a method to an environment. It's typically used to add a "pseudo-builder," a function that looks like a `Builder` but wraps up calls to multiple other `Builder`s or otherwise processes its arguments before calling one or more `Builder`s. In the following example, we want to install the program into the standard `/usr/bin` directory hierarchy, but also copy it into a local `install/bin` directory from which a package might be built:

```
def install_in_bin_dirs(env, source):
    """Install source in both bin dirs"""
    i1 = env.Install("$BIN", source)
    i2 = env.Install("$LOCALBIN", source)
    return [i1[0], i2[0]] # Return a list, like a normal builder
env = Environment(BIN='/usr/bin', LOCALBIN='#install/bin')
env.AddMethod(install_in_bin_dirs, "InstallInBinDirs")
env.InstallInBinDirs(Program('hello.c')) # installs hello in both bin dirs
```

This produces the following:

```
% scons -Q /
cc -o hello.o -c hello.c
cc -o hello hello.o
Install file: "hello" as "/usr/bin/hello"
Install file: "hello" as "install/bin/hello"
```

As mentioned, a pseudo-builder also provides more flexibility in parsing arguments than you can get with a `Builder`. The next example shows a pseudo-builder with a named argument that modifies the filename, and a separate argument for the resource file (rather than having the builder figure it out by file extension). This example also demonstrates using the global `AddMethod` function to add a method to the global Environment class, so it will be used in all subsequently created environments.

```
def BuildTestProg(env, testfile, resourcefile, testdir="tests"):
    """Build the test program;
```

```
    prepends "test_" to src and target,
    and puts target into testdir."""
    srcfile = "test_%s.c" % testfile
    target = "%s/test_%s" % (testdir, testfile)
    if env['PLATFORM'] == 'win32':
        resfile = env.RES(resourcefile)
        p = env.Program(target, [srcfile, resfile])
    else:
        p = env.Program(target, srcfile)
    return p
AddMethod(Environment, BuildTestProg)

env = Environment()
env.BuildTestProg('stuff', resourcefile='res.rc')
```

This produces the following on Linux:

```
% scons -Q
cc -o test_stuff.o -c test_stuff.c
cc -o tests/test_stuff test_stuff.o
```

And the following on Windows:

```
C:\>scons -Q
rc /nologo /fores.res res.rc
cl /Fotest_stuff.obj /c test_stuff.c /nologo
link /nologo /OUT:tests\test_stuff.exe test_stuff.obj res.res
embedManifestExeCheck(target, source, env)
```

Using `AddMethod` is better than just adding an instance method to a construction environment because it gets called as a proper method, and because `AddMethod` provides for copying the method to any clones of the construction environment instance.

20 Extending SCons: Writing Your Own Scanners

SCons has built-in scanners that know how to look in C, Fortran and IDL source files for information about other files that targets built from those files depend on--for example, in the case of files that use the C preprocessor, the `.h` files that are specified using `#include` lines in the source. You can use the same mechanisms that SCons uses to create its built-in scanners to write scanners of your own for file types that SCons does not know how to scan "out of the box."

20.1. A Simple Scanner Example

Suppose, for example, that we want to create a simple scanner for `.foo` files. A `.foo` file contains some text that will be processed, and can include other files on lines that begin with `include` followed by a file name:

```
include filename.foo
```

Scanning a file will be handled by a Python function that you must supply. Here is a function that will use the Python `re` module to scan for the `include` lines in our example:

```
import re

include_re = re.compile(r'^include\s+(\S+)$', re.M)

def kfile_scan(node, env, path, arg):
    contents = node.get_text_contents()
    return env.File(include_re.findall(contents))
```

It is important to note that you have to return a list of File nodes from the scanner function, simple strings for the file names won't do. As in the examples we are showing here, you can use the `File` function of your current Environment in order to create nodes on the fly from a sequence of file names with relative paths.

The scanner function must accept the four specified arguments and return a list of implicit dependencies. Presumably, these would be dependencies found from examining the contents of the file, although the function can perform any manipulation at all to generate the list of dependencies.

node
An SCons node object representing the file being scanned. The path name to the file can be used by converting the node to a string using the `str()` function, or an internal SCons `get_text_contents()` object method can be used to fetch the contents.

env
The construction environment in effect for this scan. The scanner function may choose to use construction variables from this environment to affect its behavior.

path
A list of directories that form the search path for included files for this scanner. This is how SCons handles the `$CPPPATH` and `$LIBPATH` variables.

arg
An optional argument that you can choose to have passed to this scanner function by various scanner instances.

A Scanner object is created using the `Scanner` function, which typically takes an `skeys` argument to associate the type of file suffix with this scanner. The Scanner object must then be associated with the `$SCANNERS` construction variable of a construction environment, typically by using the `Append` method:

```
kscan = Scanner(function = kfile_scan,
                skeys = ['.k'])
env.Append(SCANNERS = kscan)
```

When we put it all together, it looks like:

```
import re

include_re = re.compile(r'^include\s+(\S+)$', re.M)

def kfile_scan(node, env, path):
    contents = node.get_text_contents()
    includes = include_re.findall(contents)
    return env.File(includes)

kscan = Scanner(function = kfile_scan,
                skeys = ['.k'])

env = Environment(ENV = {'PATH' : '/usr/local/bin'})
env.Append(SCANNERS = kscan)

env.Command('foo', 'foo.k', 'kprocess < $SOURCES > $TARGET')
```

20.2. Adding a search path to a scanner: `FindPathDirs`

Many scanners need to search for included files or dependencies using a path variable; this is how `$CPPPATH` and `$LIBPATH` work. The path to search is passed to your scanner as the `path` argument. Path variables may be lists of nodes, semicolon-separated strings, or even contain SCons variables which need to be expanded. Fortunately, SCons

provides the `FindPathDirs` function which itself returns a function to expand a given path (given as a SCons construction variable name) to a list of paths at the time the scanner is called. Deferring evaluation until that point allows, for instance, the path to contain $TARGET references which differ for each file scanned.

Using `FindPathDirs` is quite easy. Continuing the above example, using KPATH as the construction variable with the search path (analogous to `$CPPPATH`), we just modify the `Scanner` constructor call to include a path keyword arg:

```
kscan = Scanner(function = kfile_scan,
                skeys = ['.k'],
                path_function = FindPathDirs('KPATH'))
```

FindPathDirs returns a callable object that, when called, will essentially expand the elements in env['KPATH'] and tell the scanner to search in those dirs. It will also properly add related repository and variant dirs to the search list. As a side note, the returned method stores the path in an efficient way so lookups are fast even when variable substitutions may be needed. This is important since many files get scanned in a typical build.

20.3. Using scanners with Builders

One approach for the use of scanners is with builders. There are two optional parameters we can use with a builder `source_scanner` and `target_scanner`.

```


def kfile_scan(node, env, path, arg):
    contents = node.get_text_contents()
    return env.File(include_re.findall(contents))

kscan = Scanner(function = kfile_scan,
            skeys = ['.k'],
            path_function = FindPathDirs('KPATH'))

def build_function(target, source, env):
    # Code to build "target" from "source"
    return None

bld = Builder(action = build_function,
            suffix = '.foo',
            source_scanner = kscan
            src_suffix = '.input')
env = Environment(BUILDERS = {'Foo' : bld})
env.Foo('file')


```

An emitter function can modify the list of sources or targets passed to the action function when the builder is triggered.

A scanner function will not affect the list of sources or targets seen by the builder during the build action. The scanner function will however affect if the builder should be rebuilt (if any of the files sourced by the scanner have changed for example).

21 Multi-Platform Configuration (Autoconf Functionality)

SCons has integrated support for build configuration similar in style to GNU Autoconf, but designed to be transparently multi-platform. The configuration system can help figure out if external build requirements such as system libraries or header files are available on the build system. This section describes how to use this SCons feature. (See also the SCons man page for additional information).

21.1. Configure Contexts

The basic framework for multi-platform build configuration in SCons is to create a configure context inside a construction environment by calling the `Configure` function, perform the desired checks for libraries, functions, header files, etc., and then call the configure context's `Finish` method to finish off the configuration:

```
env = Environment()
conf = Configure(env)
# Checks for libraries, header files, etc. go here!
env = conf.Finish()
```

The `Finish` call is required; if a new context is created while a context is active, even in a different construction environment, **scons** will complain and exit.

SCons provides a number of pre-defined basic checks, as well as a mechanism for adding your own custom checks.

There are a few possible strategies for failing configure checks. Some checks may be for features without which you cannot proceed. The simple approach here is just to exit SCons at that point - a number of the examples in this chapter are coded that way. If there are multiple hard requirements, however, it may be friendlier to the user to set a flag in case of any fails of hard requirements and accumulate a record of them, so that on the completion of the configure context they can all be listed prior to failing the build - as it can be frustrating to have to iterate through the setup, fixing one new requirement each iteration. Other checks may be for features which you can do without, and here the strategy will usually be to set a construction variable which the rest of the build can examine for its absence/presence, or to set particular compiler flags, library lists, etc. as appropriate for the circumstances, so you can proceed with the build appropriately based on available features.

Note that SCons uses its own dependency mechanism to determine when a check needs to be run--that is, SCons does not run the checks every time it is invoked, but caches the values returned by previous checks and uses the cached values unless something has changed. This saves a tremendous amount of developer time while working on cross-platform build issues.

The next sections describe the basic checks that SCons supports, as well as how to add your own custom checks.

21.2. Checking for the Existence of Header Files

Testing the existence of a header file requires knowing what language the header file is. This information is supplied in the `language` keyword parameter to the `CheckHeader` method. Since **scons** grew up in a world of C/C++ code, a configure context also has a `CheckCHeader` method that specifically checks for the existence of a C header file:

```
env = Environment()
conf = Configure(env)
if not conf.CheckCHeader('math.h'):
    print('Math.h must be installed!')
    Exit(1)
if conf.CheckCHeader('foo.h'):
    conf.env.Append(CPPDEFINES='HAS_FOO_H')
env = conf.Finish()
```

As shown in the example, depending on the circumstances you can choose to terminate the build if a given header file doesn't exist, or you can modify the construction environment based on the presence or absence of a header file (the same applies to any other check). If there are a many elements to check for, it may be friendlier for the user if you do not terminate on the first failure, but track the problems found until the end and report on all of them, that way the user does not have to iterate multiple times, each time finding one new dependency that needs to be installed.

If you need to check for the existence a C++ header file, use the `CheckCXXHeader` method:

```
env = Environment()
conf = Configure(env)
if not conf.CheckCXXHeader('vector.h'):
    print('vector.h must be installed!')
    Exit(1)
env = conf.Finish()
```

21.3. Checking for the Availability of a Function

Check for the availability of a specific function using the `CheckFunc` method:

```
env = Environment()
conf = Configure(env)
if not conf.CheckFunc('strcpy'):
    print('Did not find strcpy(), using local version')
    conf.env.Append(CPPDEFINES=('strcpy','my_local_strcpy'))
env = conf.Finish()
```

21.4. Checking for the Availability of a Library

Check for the availability of a library using the `CheckLib` method. You only specify the base part of the library name, you don't need to add a `lib` prefix or a `.a` or `.lib` suffix:

```
env = Environment()
conf = Configure(env)
if not conf.CheckLib('m'):
    print('Did not find libm.a or m.lib, exiting!')
    Exit(1)
env = conf.Finish()
```

Because the ability to use a library successfully often depends on having access to a header file that describes the library's interface, you can check for a library *and* a header file at the same time by using the `CheckLibWithHeader` method:

```
env = Environment()
conf = Configure(env)
if not conf.CheckLibWithHeader('m', 'math.h', language='c'):
    print('Did not find libm.a or m.lib, exiting!')
    Exit(1)
env = conf.Finish()
```

This is essentially shorthand for separate calls to the `CheckHeader` and `CheckLib` functions.

21.5. Checking for the Availability of a `typedef`

Check for the availability of a `typedef` by using the `CheckType` method:

```
env = Environment()
conf = Configure(env)
if not conf.CheckType('off_t'):
    print('Did not find off_t typedef, assuming int')
    conf.env.Append(CPPDEFINES=('off_t','int'))
env = conf.Finish()
```

You can also add a string that will be placed at the beginning of the test file that will be used to check for the `typedef`. This provide a way to specify files that must be included to find the `typedef`:

```
env = Environment()
conf = Configure(env)
if not conf.CheckType('off_t', '#include <sys/types.h>\n'):
    print('Did not find off_t typedef, assuming int')
    conf.env.Append(CPPDEFINES=('off_t','int'))
env = conf.Finish()
```

21.6. Checking the size of a datatype

Check the size of a datatype by using the `CheckTypeSize` method:

```
env = Environment()
conf = Configure(env)
int_size = conf.CheckTypeSize('unsigned int')
print('sizeof unsigned int is', int_size)
env = conf.Finish()
```

```
% scons -Q
sizeof unsigned int is 4
scons: `.' is up to date.
```

21.7. Checking for the Presence of a program

Check for the presence of a program by using the `CheckProg` method:

```
env = Environment()
conf = Configure(env)
if not conf.CheckProg('foobar'):
  print('Unable to find the program foobar on the system')
  Exit(1)
env = conf.Finish()
```

21.8. Extending SCons: Adding Your Own Custom Checks

A custom check is a Python function that checks for a certain condition to exist on the running system, usually using methods that SCons supplies to take care of the details of checking whether a compilation succeeds, a link succeeds, a program is runnable, etc. A simple custom check for the existence of a specific library might look as follows:

```
mylib_test_source_file = """
#include <mylib.h>
int main(int argc, char **argv)
{
    MyLibrary mylib(argc, argv);
    return 0;
}
"""

def CheckMyLibrary(context):
```

```
    context.Message('Checking for MyLibrary...')
    result = context.TryLink(mylib_test_source_file, '.c')
    context.Result(result)
    return result
```

The `Message` and `Result` methods should typically begin and end a custom check to let the user know what's going on: the `Message` call prints the specified message (with no trailing newline) and the `Result` call prints `yes` if the check succeeds and `no` if it doesn't. The `TryLink` method actually tests for whether the specified program text will successfully link.

(Note that a custom check can modify its check based on any arguments you choose to pass it, or by using or modifying the configure context environment in the `context.env` attribute.)

This custom check function is then attached to the configure context by passing a dictionary to the `Configure` call that maps a name of the check to the underlying function:

```
env = Environment()
conf = Configure(env, custom_tests={'CheckMyLibrary': CheckMyLibrary})
```

You'll typically want to make the check and the function name the same, as we've done here, to avoid potential confusion.

We can then put these pieces together and actually call the `CheckMyLibrary` check as follows:

```
mylib_test_source_file = """
#include <mylib.h>
int main(int argc, char **argv)
{
    MyLibrary mylib(argc, argv);
    return 0;
}
"""

def CheckMyLibrary(context):
    context.Message('Checking for MyLibrary... ')
    result = context.TryLink(mylib_test_source_file, '.c')
    context.Result(result)
    return result

env = Environment()
conf = Configure(env, custom_tests={'CheckMyLibrary': CheckMyLibrary})
if not conf.CheckMyLibrary():
    print('MyLibrary is not installed!')
    Exit(1)
env = conf.Finish()

# We would then add actual calls like Program() to build
# something using the "env" construction environment.
```

If MyLibrary is not installed on the system, the output will look like:

```
% scons
scons: Reading SConscript file ...
Checking for MyLibrary... no
MyLibrary is not installed!
```

If MyLibrary is installed, the output will look like:

```
% scons
scons: Reading SConscript file ...
Checking for MyLibrary... yes
scons: done reading SConscript
scons: Building targets ...
    .
    .
    .
```

21.9. Not Configuring When Cleaning Targets

Using multi-platform configuration as described in the previous sections will run the configuration commands even when invoking **`scons -c`** to clean targets:

```
% scons -Q -c
Checking for MyLibrary... yes
Removed foo.o
Removed foo
```

Although running the platform checks when removing targets doesn't hurt anything, it's usually unnecessary. You can avoid this by using the `GetOption` method to check whether the `-c` (clean) option has been invoked on the command line:

```
env = Environment()
if not env.GetOption('clean'):
    conf = Configure(env, custom_tests={'CheckMyLibrary': CheckMyLibrary})
    if not conf.CheckMyLibrary():
        print('MyLibrary is not installed!')
        Exit(1)
    env = conf.Finish()
```

```
% scons -Q -c
Removed foo.o
Removed foo
```

22 Caching Built Files

On multi-developer software projects, you can sometimes speed up every developer's builds a lot by allowing them to share the derived files that they build. After all, it is relatively rare that any in-progress change affects more than a few derived files, most will be unchanged. SCons makes this easy and reliable.

22.1. Specifying the Shared Cache Directory

To enable sharing of derived files, use the `CacheDir` function in any `SConscript` file:

```
CacheDir('/usr/local/build_cache')
```

The cache directory you specify must be readable and writable by all developers who will be sharing derived files. It should also be in some central location that all builds will be able to access. In environments where developers are using separate systems (like individual workstations) for builds, this directory would typically be on a shared or NFS-mounted file system. While SCons will create the specified cache directory as needed, in this multi user scenario it is usually best to create it ahead of time so the access rights can be set up correctly.

Here's what happens: When a build has a `CacheDir` specified, every time a file is built, it is stored in that cache directory along with its build signature. On subsequent builds, before an action is invoked to build a file, SCons will check the shared cache directory to see if a file with the exact same build signature already exists. [1] If so, the derived file will not be built locally, but will be copied into the local build directory from the shared cache directory, like this:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q -c
Removed hello.o
Removed hello
% scons -Q
Retrieved `hello.o' from cache
Retrieved `hello' from cache
```

[1] A few inner details: SCons tracks two main kinds of cryptographic hashes: *content signatures*, which are a hash of the contents of a file; and *build signatures*, which are a hash of the elements needed to build a target, such as the exact command line, the contents of the sources, and possibly information about tools used in the build. The hash function produces a unique signature from its inputs, no other set of inputs can produce that same signature. The build signature from building a target is used as the filename of the target file in the shared cache - that way lookups are efficient, just compute a build signature and see if a file exists with that as the name.

Note that the `CacheDir` feature requires that build signatures be calculated, even if you configure SCons to use timestamps to decide if files are up to date (see the Chapter 6, *Dependencies* chapter for information about the `Decider` function), since the build signature is used to determine if a target file exists in the cache. Consequently, using `CacheDir` may reduce or eliminate any potential performance improvements from using timestamps for up-to-date decisions.

22.2. Keeping Build Output Consistent

One potential drawback to using a shared cache is that the output printed by SCons can be inconsistent from invocation to invocation, because any given file may be rebuilt one time and retrieved from the shared cache the next time. This can make analyzing build output more difficult, especially for automated scripts that expect consistent output each time.

If, however, you use the `--cache-show` option, SCons will print the command line that it *would* have executed to build the file, even when it is retrieving the file from the shared cache. This makes the build output consistent every time the build is run:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q -c
Removed hello.o
Removed hello
% scons -Q --cache-show
cc -o hello.o -c hello.c
cc -o hello hello.o
```

The trade-off, of course, is that you no longer know whether or not SCons has retrieved a derived file from cache or has rebuilt it locally.

22.3. Not Using the Shared Cache for Specific Files

You may want to disable caching for certain specific files in your configuration. For example, if you only want to put executable files in a central cache, but not the intermediate object files, you can use the `NoCache` function to specify that the object files should not be cached:

```
env = Environment()
obj = env.Object('hello.c')
env.Program('hello.c')
CacheDir('cache')
NoCache('hello.o')
```

Then when you run **scons** after cleaning the built targets, it will recompile the object file locally (since it doesn't exist in the shared cache directory), but still realize that the shared cache directory contains an up-to-date executable program that can be retrieved instead of re-linking:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q -c
```

```
Removed hello.o
Removed hello
% scons -Q
cc -o hello.o -c hello.c
Retrieved `hello' from cache
```

22.4. Disabling the Shared Cache

Retrieving an already-built file from the shared cache is usually a significant time-savings over rebuilding the file, but how much of a savings (or even whether it saves time at all) can depend a great deal on your system or network configuration. For example, retrieving cached files from a busy server over a busy network might end up being slower than rebuilding the files locally.

In these cases, you can specify the `--cache-disable` command-line option to tell SCons to not retrieve already-built files from the shared cache directory:

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q -c
Removed hello.o
Removed hello
% scons -Q
Retrieved `hello.o' from cache
Retrieved `hello' from cache
% scons -Q -c
Removed hello.o
Removed hello
% scons -Q --cache-disable
cc -o hello.o -c hello.c
cc -o hello hello.o
```

22.5. Populating a Shared Cache With Already-Built Files

Sometimes, you may have one or more derived files already built in your local build tree that you wish to make available to other people doing builds. For example, you may find it more effective to perform integration builds with the cache disabled (per the previous section) and only populate the shared cache directory with the built files after the integration build has completed successfully. This way, the cache will only get filled up with derived files that are part of a complete, successful build not with files that might be later overwritten while you debug integration problems.

In this case, you can use the the `--cache-force` option to tell SCons to put all derived files in the cache, even if the files already exist in your local tree from having been built by a previous invocation:

```
% scons -Q --cache-disable
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q -c
Removed hello.o
Removed hello
% scons -Q --cache-disable
```

```
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q --cache-force
scons: `.' is up to date.
% scons -Q
scons: `.' is up to date.
```

Notice how the above sample run demonstrates that the `--cache-disable` option avoids putting the built `hello.o` and `hello` files in the cache, but after using the `--cache-force` option, the files have been put in the cache for the next invocation to retrieve.

22.6. Minimizing Cache Contention: the `--random` Option

If you allow multiple builds to update the shared cache directory simultaneously, two builds that occur at the same time can sometimes start "racing" with one another to build the same files in the same order. If, for example, you are linking multiple files into an executable program:

```
Program('prog',
        ['f1.c', 'f2.c', 'f3.c', 'f4.c', 'f5.c'])
```

SCons will normally build the input object files on which the program depends in their normal, sorted order:

```
% scons -Q
cc -o f5.o -c f5.c
cc -o f2.o -c f2.c
cc -o f4.o -c f4.c
cc -o f1.o -c f1.c
cc -o f3.o -c f3.c
cc -o prog f1.o f2.o f3.o f4.o f5.o
```

But if two such builds take place simultaneously, they may each look in the cache at nearly the same time and both decide that `f1.o` must be rebuilt and pushed into the shared cache directory, then both decide that `f2.o` must be rebuilt (and pushed into the shared cache directory), then both decide that `f3.o` must be rebuilt... This won't cause any actual build problems--both builds will succeed, generate correct output files, and populate the cache--but it does represent wasted effort.

To alleviate such contention for the cache, you can use the `--random` command-line option to tell SCons to build dependencies in a random order:

```
% scons -Q --random
cc -o f3.o -c f3.c
cc -o f1.o -c f1.c
cc -o f5.o -c f5.c
cc -o f2.o -c f2.c
cc -o f4.o -c f4.c
cc -o prog f1.o f2.o f3.o f4.o f5.o
```

Multiple builds using the `--random` option will usually build their dependencies in different, random orders, which minimizes the chances for a lot of contention for same-named files in the shared cache directory. Multiple simultaneous

builds might still race to try to build the same target file on occasion, but long sequences of inefficient contention should be rare.

Note, of course, the `--random` option will cause the output that SCons prints to be inconsistent from invocation to invocation, which may be an issue when trying to compare output from different build runs.

If you want to make sure dependencies will be built in a random order without having to specify the `--random` on very command line, you can use the `SetOption` function to set the `random` option within any `SConscript` file:

```
SetOption('random', 1)
Program('prog',
        ['f1.c', 'f2.c', 'f3.c', 'f4.c', 'f5.c'])
```

22.7. Using a Custom CacheDir Class

SCons' internal `CacheDir` class can be extended to support customization around the details of caching behaviors, for example using compressed cache files, encrypted cache files, gathering statistics and data, or many other aspects.

To create your own custom `CacheDir` class, your custom class must be a subclass of SCons internal `SCons.CacheDir.CacheDir` class. You can then pass your custom `CacheDir` class to the `CacheDir` method or set the environment construction variable `$CACHEDIR_CLASS` to the class before configuring the cache in that environment. SCons will internally invoke and use your custom class when performing cache operations. The below example shows a simple use case of overriding the copy_from_cache method to record the total number of bytes pulled from the cache.

```
        import SCons
        import os

        class CustomCacheDir(SCons.CacheDir.CacheDir):
            total_retrieved = 0

            @classmethod
            def copy_from_cache(cls, env, src, dst):
                # record total bytes pulled from cache
                cls.total_retrieved += os.stat(src).st_size
                super().copy_from_cache(env, src, dst)

        env = Environment()
        env.CacheDir('scons-cache', CustomCacheDir)
        # ...
```

23 Alias Targets

We've already seen how you can use the `Alias` function to create a target named `install`:

```
env = Environment()
hello = env.Program('hello.c')
env.Install('/usr/bin', hello)
env.Alias('install', '/usr/bin')
```

You can then use this alias on the command line to tell SCons more naturally that you want to install files:

```
% scons -Q install
cc -o hello.o -c hello.c
cc -o hello hello.o
Install file: "hello" as "/usr/bin/hello"
```

Like other `Builder` methods, though, the `Alias` method returns an object representing the alias being built. You can then use this object as input to anothother `Builder`. This is especially useful if you use such an object as input to another call to the `Alias` `Builder`, allowing you to create a hierarchy of nested aliases:

```
env = Environment()
p = env.Program('foo.c')
l = env.Library('bar.c')
env.Install('/usr/bin', p)
env.Install('/usr/lib', l)
ib = env.Alias('install-bin', '/usr/bin')
il = env.Alias('install-lib', '/usr/lib')
env.Alias('install', [ib, il])
```

This example defines separate `install`, `install-bin`, and `install-lib` aliases, allowing you finer control over what gets installed:

```
% scons -Q install-bin
cc -o foo.o -c foo.c
cc -o foo foo.o
Install file: "foo" as "/usr/bin/foo"
% scons -Q install-lib
```

```
cc -o bar.o -c bar.c
ar rc libbar.a bar.o
ranlib libbar.a
Install file: "libbar.a" as "/usr/lib/libbar.a"
% scons -Q -c /
Removed foo.o
Removed foo
Removed /usr/bin/foo
Removed bar.o
Removed libbar.a
Removed /usr/lib/libbar.a
% scons -Q install
cc -o foo.o -c foo.c
cc -o foo foo.o
Install file: "foo" as "/usr/bin/foo"
cc -o bar.o -c bar.c
ar rc libbar.a bar.o
ranlib libbar.a
Install file: "libbar.a" as "/usr/lib/libbar.a"
```

24 Java Builds

So far, we've been using examples of building C and C++ programs to demonstrate the features of SCons. SCons also supports building Java programs, but Java builds are handled slightly differently, which reflects the ways in which the Java compiler and tools build programs differently than other languages' tool chains.

24.1. Building Java Class Files: the Java Builder

The basic activity when programming in Java, of course, is to take one or more `.java` files containing Java source code and to call the Java compiler to turn them into one or more `.class` files. In SCons, you do this by giving the `Java` Builder a target directory in which to put the `.class` files, and a source directory that contains the `.java` files:

```
Java('classes', 'src')
```

If the `src` directory contains three `.java` source files, then running SCons might look like this:

```
% scons -Q
javac -d classes -sourcepath src src/Example1.java src/Example2.java src/Example3.java
```

SCons will actually search the `src` directory tree for all of the `.java` files. The Java compiler will then create the necessary class files in the `classes` subdirectory, based on the class names found in the `.java` files.

24.2. How SCons Handles Java Dependencies

In addition to searching the source directory for `.java` files, SCons actually runs the `.java` files through a stripped-down Java parser that figures out what classes are defined. In other words, SCons knows, without you having to tell it, what `.class` files will be produced by the javac call. So our one-liner example from the preceding section:

```
Java('classes', 'src')
```

Will not only tell you reliably that the `.class` files in the `classes` subdirectory are up-to-date:

```
% scons -Q
javac -d classes -sourcepath src src/Example1.java src/Example2.java src/Example3.java
```

```
% scons -Q classes
scons: `classes' is up to date.
```

But it will also remove all of the generated .class files, even for inner classes, without you having to specify them manually. For example, if our Example1.java and Example3.java files both define additional classes, and the class defined in Example2.java has an inner class, running **scons -c** will clean up all of those .class files as well:

```
% scons -Q
javac -d classes -sourcepath src src/Example1.java src/Example2.java src/Example3.java
% scons -Q -c classes
Removed classes/Example1.class
Removed classes/AdditionalClass1.class
Removed classes/Example2$Inner2.class
Removed classes/Example2.class
Removed classes/Example3.class
Removed classes/AdditionalClass3.class
```

To ensure correct handling of .class dependencies in all cases, you need to tell SCons which Java version is being used. This is needed because Java 1.5 changed the .class file names for nested anonymous inner classes. Use the JAVAVERSION construction variable to specify the version in use. With Java 1.6, the one-liner example can then be defined like this:

```
Java('classes', 'src', JAVAVERSION='1.6')
```

See JAVAVERSION in the man page for more information.

24.3. Building Java Archive (.jar) Files: the Jar Builder

After building the class files, it's common to collect them into a Java archive (.jar) file, which you do by calling the Jar Builder. If you want to just collect all of the class files within a subdirectory, you can just specify that subdirectory as the Jar source:

```
Java(target = 'classes', source = 'src')
Jar(target = 'test.jar', source = 'classes')
```

SCons will then pass that directory to the jar command, which will collect all of the underlying .class files:

```
% scons -Q
javac -d classes -sourcepath src src/Example1.java src/Example2.java src/Example3.java
jar cf test.jar classes
```

If you want to keep all of the .class files for multiple programs in one location, and only archive some of them in each .jar file, you can pass the Jar builder a list of files as its source. It's extremely simple to create multiple .jar files this way, using the lists of target class files created by calls to the Java builder as sources to the various Jar calls:

```
prog1_class_files = Java(target = 'classes', source = 'prog1')
prog2_class_files = Java(target = 'classes', source = 'prog2')
```

```
Jar(target = 'prog1.jar', source = prog1_class_files)
Jar(target = 'prog2.jar', source = prog2_class_files)
```

This will then create `prog1.jar` and `prog2.jar` next to the subdirectories that contain their `.java` files:

```
% scons -Q
javac -d classes -sourcepath prog1 prog1/Example1.java prog1/Example2.java
javac -d classes -sourcepath prog2 prog2/Example3.java prog2/Example4.java
jar cf prog1.jar -C classes Example1.class -C classes Example2.class
jar cf prog2.jar -C classes Example3.class -C classes Example4.class
```

24.4. Building C Header and Stub Files: the JavaH Builder

You can generate C header and source files for implementing native methods, by using the `JavaH` Builder. There are several ways of using the `JavaH` Builder. One typical invocation might look like:

```
classes = Java(target = 'classes', source = 'src/pkg/sub')
JavaH(target = 'native', source = classes)
```

The source is a list of class files generated by the call to the `Java` Builder, and the target is the output directory in which we want the C header files placed. The target gets converted into the `-d` when SCons runs javah:

```
% scons -Q
javac -d classes -sourcepath src/pkg/sub src/pkg/sub/Example1.java src/pkg/sub/Example2.ja
javah -d native -classpath classes pkg.sub.Example1 pkg.sub.Example2 pkg.sub.Example3
```

In this case, the call to javah will generate the header files `native/pkg_sub_Example1.h`, `native/pkg_sub_Example2.h` and `native/pkg_sub_Example3.h`. Notice that SCons remembered that the class files were generated with a target directory of `classes`, and that it then specified that target directory as the `-classpath` option to the call to javah.

Although it's more convenient to use the list of class files returned by the `Java` Builder as the source of a call to the `JavaH` Builder, you *can* specify the list of class files by hand, if you prefer. If you do, you need to set the `$JAVACLASSDIR` construction variable when calling `JavaH`:

```
Java(target='classes', source='src/pkg/sub')
class_file_list = [
    'classes/pkg/sub/Example1.class',
    'classes/pkg/sub/Example2.class',
    'classes/pkg/sub/Example3.class',
]
JavaH(target='native', source=class_file_list, JAVACLASSDIR='classes')
```

The `$JAVACLASSDIR` value then gets converted into the `-classpath` when SCons runs javah:

```
% scons -Q
javac -d classes -sourcepath src/pkg/sub src/pkg/sub/Example1.java src/pkg/sub/Example2.ja
javah -d native -classpath classes pkg.sub.Example1 pkg.sub.Example2 pkg.sub.Example3
```

Lastly, if you don't want a separate header file generated for each source file, you can specify an explicit File Node as the target of the JavaH Builder:

```
classes = Java(target='classes', source='src/pkg/sub')
JavaH(target=File('native.h'), source=classes)
```

Because SCons assumes by default that the target of the JavaH builder is a directory, you need to use the File function to make sure that SCons doesn't create a directory named native.h. When a file is used, though, SCons correctly converts the file name into the javah -o option:

```
% scons -Q
javac -d classes -sourcepath src/pkg/sub src/pkg/sub/Example1.java src/pkg/sub/Example2.ja
javah -o native.h -classpath classes pkg.sub.Example1 pkg.sub.Example2 pkg.sub.Example3
```

Note that the the **javah** command was removed from the JDK as of JDK 10, and the approved method (available since JDK 8) is to use **javac** to generate native headers at the same time as the Java source code is compiled.. As such the JavaH builder is of limited utility in later Java versions.

24.5. Building RMI Stub and Skeleton Class Files: the RMIC Builder

You can generate Remote Method Invocation stubs by using the RMIC Builder. The source is a list of directories, typically returned by a call to the Java Builder, and the target is an output directory where the _Stub.class and _Skel.class files will be placed:

```
classes = Java(target = 'classes', source = 'src/pkg/sub')
RMIC(target = 'outdir', source = classes)
```

As it did with the JavaH Builder, SCons remembers the class directory and passes it as the -classpath option to rmic:

```
% scons -Q
javac -d classes -sourcepath src/pkg/sub src/pkg/sub/Example1.java src/pkg/sub/Example2.ja
rmic -d outdir -classpath classes pkg.sub.Example1 pkg.sub.Example2
```

This example would generate the files outdir/pkg/sub/Example1_Skel.class, outdir/pkg/sub/Example1_Stub.class, outdir/pkg/sub/Example2_Skel.class and outdir/pkg/sub/Example2_Stub.class.

25 Internationalization and localization with gettext

The `gettext` toolset supports internationalization and localization of SCons-based projects. Builders provided by `gettext` automatize generation and updates of translation files. You can manage translations and translation templates similarly to how it's done with autotools.

25.1. Prerequisites

To follow examples provided in this chapter set up your operating system to support two or more languages. In following examples we use locales `en_US`, `de_DE`, and `pl_PL`.

Ensure, that you have GNU gettext utilities [http://www.gnu.org/software/gettext/manual/gettext.html] installed on your system.

To edit translation files you may wish to install poedit [http://www.poedit.net/] editor.

25.2. Simple project

Let's start with a very simple project, the "Hello world" program for example

```
/* hello.c */
#include <stdio.h>
int main(int argc, char* argv[])
{
  printf("Hello world\n");
  return 0;
}
```

Prepare a `SConstruct` to compile the program as usual.

```
# SConstruct
env = Environment()
hello = Program(["hello.c"])
```

Now we'll convert the project to a multi-lingual one. If you don't already have GNU gettext utilities [http://www.gnu.org/software/gettext/manual/gettext.html] installed, install them from your preffered package repository, or download from http://ftp.gnu.org/gnu/gettext/ [http://ftp.gnu.org/gnu/gettext/]. For the purpose of this example, you should have following three locales installed on your system: `en_US`, `de_DE` and `pl_PL`. On debian, for example, you may enable certain locales through **dpkg-reconfigure locales**.

First prepare the `hello.c` program for internationalization. Change the previous code so it reads as follows:

```
/* hello.c */
#include <stdio.h>
#include <libintl.h>
#include <locale.h>
int main(int argc, char* argv[])
{
  bindtextdomain("hello", "locale");
  setlocale(LC_ALL, "");
  textdomain("hello");
  printf(gettext("Hello world\n"));
  return 0;
}
```

Detailed recipes for such conversion can be found at http://www.gnu.org/software/gettext/manual/gettext.html#Sources [http://www.gnu.org/software/gettext/manual/gettext.html#Sources]. The `gettext("...")` has two purposes. First, it marks messages for the **xgettext(1)** program, which we will use to extract from the sources the messages for localization. Second, it calls the `gettext` library internals to translate the message at runtime.

Now we shall instruct SCons how to generate and maintain translation files. For that, use the `Translate` builder and `MOFiles` builder. The first one takes source files, extracts internationalized messages from them, creates so-called `POT` file (translation template), and then creates `PO` translation files, one for each requested language. Later, during the development lifecycle, the builder keeps all these files up-to date. The `MOFiles` builder compiles the `PO` files to binary form. Then install the `MO` files under directory called `locale`.

The completed `SConstruct` is as follows:

```
# SConstruct
env = Environment( tools = ['default', 'gettext'] )
hello = env.Program(["hello.c"])
env['XGETTEXTFLAGS'] = [
  '--package-name=%s' % 'hello',
  '--package-version=%s' % '1.0',
]
po = env.Translate(["pl","en", "de"], ["hello.c"], POAUTOINIT = 1)
mo = env.MOFiles(po)
InstallAs(["locale/en/LC_MESSAGES/hello.mo"], ["en.mo"])
InstallAs(["locale/pl/LC_MESSAGES/hello.mo"], ["pl.mo"])
InstallAs(["locale/de/LC_MESSAGES/hello.mo"], ["de.mo"])
```

Generate the translation files with **scons po-update**. You should see the output from SCons simillar to this:

```
user@host:$ scons po-update
```

```
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
Entering '/home/ptomulik/projects/tmp'
xgettext --package-name=hello --package-version=1.0 -o - hello.c
Leaving '/home/ptomulik/projects/tmp'
Writting 'messages.pot' (new file)
msginit --no-translator -l pl -i messages.pot -o pl.po
Created pl.po.
msginit --no-translator -l en -i messages.pot -o en.po
Created en.po.
msginit --no-translator -l de -i messages.pot -o de.po
Created de.po.
scons: done building targets.
```

If everything is right, you should see following new files.

```
user@host:$ ls *.po*
de.po  en.po  messages.pot  pl.po
```

Open `en.po` in **poedit** and provide the English translation to message `"Hello world\n"`. Do the same for `de.po` (deutsch) and `pl.po` (polish). Let the translations be, for example:

- `en: "Welcome to beautiful world!\n"`
- `de: "Hallo Welt!\n"`
- `pl: "Witaj swiecie!\n"`

Now compile the project by executing **scons**. The output should be similar to this:

```
user@host:$ scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
msgfmt -c -o de.mo de.po
msgfmt -c -o en.mo en.po
gcc -o hello.o -c hello.c
gcc -o hello hello.o
Install file: "de.mo" as "locale/de/LC_MESSAGES/hello.mo"
Install file: "en.mo" as "locale/en/LC_MESSAGES/hello.mo"
msgfmt -c -o pl.mo pl.po
Install file: "pl.mo" as "locale/pl/LC_MESSAGES/hello.mo"
scons: done building targets.
```

SCons automatically compiled the `PO` files to binary format `MO`, and the `InstallAs` lines installed these files under `locale` folder.

Your program should be now ready. You may try it as follows (linux):

```
```

```
user@host:$ LANG=en_US.UTF-8 ./hello
Welcome to beautiful world
```

```
user@host:$ LANG=de_DE.UTF-8 ./hello
Hallo Welt
```

```
user@host:$ LANG=pl_PL.UTF-8 ./hello
Witaj swiecie
```

To demonstrate the further life of translation files, let's change Polish translation (**poedit pl.po**) to `"Witaj drogi swiecie\n"`. Run **scons** to see how scons reacts to this

```
user@host:$scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
msgfmt -c -o pl.mo pl.po
Install file: "pl.mo" as "locale/pl/LC_MESSAGES/hello.mo"
scons: done building targets.
```

Now, open `hello.c` and add another one `printf` line with new message.

```
/* hello.c */
#include <stdio.h>
#include <libintl.h>
#include <locale.h>
int main(int argc, char* argv[])
{
  bindtextdomain("hello", "locale");
  setlocale(LC_ALL, "");
  textdomain("hello");
  printf(gettext("Hello world\n"));
  printf(gettext("and good bye\n"));
  return 0;
}
```

Compile project with **scons**. This time, the **msgmerge(1)** program is used by SCons to update `PO` file. The output from compilation is like:

```
user@host:$scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
Entering '/home/ptomulik/projects/tmp'
xgettext --package-name=hello --package-version=1.0 -o - hello.c
```

```
Leaving '/home/ptomulik/projects/tmp'
Writting 'messages.pot' (messages in file were outdated)
msgmerge --update de.po messages.pot
... done.
msgfmt -c -o de.mo de.po
msgmerge --update en.po messages.pot
... done.
msgfmt -c -o en.mo en.po
gcc -o hello.o -c hello.c
gcc -o hello hello.o
Install file: "de.mo" as "locale/de/LC_MESSAGES/hello.mo"
Install file: "en.mo" as "locale/en/LC_MESSAGES/hello.mo"
msgmerge --update pl.po messages.pot
... done.
msgfmt -c -o pl.mo pl.po
Install file: "pl.mo" as "locale/pl/LC_MESSAGES/hello.mo"
scons: done building targets.
```

The next example demonstrates what happens if we change the source code in such way that the internationalized messages do not change. The answer is that none of translation files (`POT`, `PO`) are touched (i.e. no content changes, no creation/modification time changed and so on). Let's append another line to the program (after the last printf), so its code becomes:

```
/* hello.c */
#include <stdio.h>
#include <libintl.h>
#include <locale.h>
int main(int argc, char* argv[])
{
  bindtextdomain("hello", "locale");
  setlocale(LC_ALL, "");
  textdomain("hello");
  printf(gettext("Hello world\n"));
  printf(gettext("and good bye\n"));
  printf("----------------\n");
  return a;
}
```

Compile the project. You'll see on your screen

```
user@host:$scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
Entering '/home/ptomulik/projects/tmp'
xgettext --package-name=hello --package-version=1.0 -o - hello.c
Leaving '/home/ptomulik/projects/tmp'
Not writting 'messages.pot' (messages in file found to be up-to-date)
gcc -o hello.o -c hello.c
gcc -o hello hello.o
scons: done building targets.
```

As you see, the internationalized messages ditn't change, so the `POT` and the rest of translation files have not even been touched.

26 Miscellaneous Functionality

SCons supports a lot of additional functionality that doesn't readily fit into the other chapters.

26.1. Verifying the Python Version: the `EnsurePythonVersion` Function

Although the SCons code itself will run on any 2.x Python version 2.7 or later, you are perfectly free to make use of Python syntax and modules from later versions when writing your `SConscript` files or your own local modules. If you do this, it's usually helpful to configure SCons to exit gracefully with an error message if it's being run with a version of Python that simply won't work with your code. This is especially true if you're going to use SCons to build source code that you plan to distribute publicly, where you can't be sure of the Python version that an anonymous remote user might use to try to build your software.

SCons provides an `EnsurePythonVersion` function for this. You simply pass it the major and minor versions numbers of the version of Python you require:

```
EnsurePythonVersion(2, 5)
```

And then SCons will exit with the following error message when a user runs it with an unsupported earlier version of Python:

```
% scons -Q
Python 2.5 or greater required, but you have Python 2.3.6
```

26.2. Verifying the SCons Version: the `EnsureSConsVersion` Function

You may, of course, write your `SConscript` files to use features that were only added in recent versions of SCons. When you publicly distribute software that is built using SCons, it's helpful to have SCons verify the version being used and exit gracefully with an error message if the user's version of SCons won't work with your `SConscript`

files. SCons provides an EnsureSConsVersion function that verifies the version of SCons in the same the EnsurePythonVersion function verifies the version of Python, by passing in the major and minor versions numbers of the version of SCons you require:

```
EnsureSConsVersion(1, 0)
```

And then SCons will exit with the following error message when a user runs it with an unsupported earlier version of SCons:

```
% scons -Q
SCons 1.0 or greater required, but you have SCons 0.98.5
```

26.3. Explicitly Terminating SCons While Reading SConscript Files: the Exit Function

SCons supports an Exit function which can be used to terminate SCons while reading the SConscript files, usually because you've detected a condition under which it doesn't make sense to proceed:

```
if ARGUMENTS.get('FUTURE'):
    print("The FUTURE option is not supported yet!")
    Exit(2)
env = Environment()
env.Program('hello.c')
```

```
% scons -Q FUTURE=1
The FUTURE option is not supported yet!
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
```

The Exit function takes as an argument the (numeric) exit status that you want SCons to exit with. If you don't specify a value, the default is to exit with 0, which indicates successful execution.

Note that the Exit function is equivalent to calling the Python sys.exit function (which the it actually calls), but because Exit is a SCons function, you don't have to import the Python sys module to use it.

26.4. Searching for Files: the FindFile Function

The FindFile function searches for a file in a list of directories. If there is only one directory, it can be given as a simple string. The function returns a File node if a matching file exists, or None if no file is found. (See the documentation for the Glob function for an alternative way of searching for entries in a directory.)

```
# one directory
```

```
print("%s"%FindFile('missing', '.'))
t = FindFile('exists', '.')
print("%s %s"%(t.__class__, t))
```

```
% scons -Q
None
<class 'SCons.Node.FS.File'> exists
scons: `.' is up to date.
```

```
# several directories
includes = [ '.', 'include', 'src/include']
headers = [ 'nonesuch.h', 'config.h', 'private.h', 'dist.h']
for hdr in headers:
    print('%-12s: %s'%(hdr, FindFile(hdr, includes)))
```

```
% scons -Q
nonesuch.h  : None
config.h    : config.h
private.h   : src/include/private.h
dist.h      : include/dist.h
scons: `.' is up to date.
```

If the file exists in more than one directory, only the first occurrence is returned.

```
print(FindFile('multiple', ['sub1', 'sub2', 'sub3']))
print(FindFile('multiple', ['sub2', 'sub3', 'sub1']))
print(FindFile('multiple', ['sub3', 'sub1', 'sub2']))
```

```
% scons -Q
sub1/multiple
sub2/multiple
sub3/multiple
scons: `.' is up to date.
```

In addition to existing files, FindFile will also find derived files (that is, non-leaf files) that haven't been built yet. (Leaf files should already exist, or the build will fail!)

```
# Neither file exists, so build will fail
Command('derived', 'leaf', 'cat >$TARGET $SOURCE')
print(FindFile('leaf', '.'))
print(FindFile('derived', '.'))
```

```
% scons -Q
leaf
derived
cat > derived leaf
```

```
# Only 'leaf' exists
```

```
Command('derived', 'leaf', 'cat >$TARGET $SOURCE')
print(FindFile('leaf', '.'))
print(FindFile('derived', '.'))
```

```
% scons -Q
leaf
derived
cat > derived leaf
```

If a source file exists, FindFile will correctly return the name in the build directory.

```
# Only 'src/leaf' exists
VariantDir('build', 'src')
print(FindFile('leaf', 'build'))
```

```
% scons -Q
build/leaf
scons: `.' is up to date.
```

26.5. Handling Nested Lists: the Flatten Function

SCons supports a Flatten function which takes an input Python sequence (list or tuple) and returns a flattened list containing just the individual elements of the sequence. This can be handy when trying to examine a list composed of the lists returned by calls to various Builders. For example, you might collect object files built in different ways into one call to the Program Builder by just enclosing them in a list, as follows:

```
objects = [
    Object('prog1.c'),
    Object('prog2.c', CCFLAGS='-DFOO'),
]
Program(objects)
```

Because the Builder calls in SCons flatten their input lists, this works just fine to build the program:

```
% scons -Q
cc -o prog1.o -c prog1.c
cc -o prog2.o -c -DFOO prog2.c
cc -o prog1 prog1.o prog2.o
```

But if you were debugging your build and wanted to print the absolute path of each object file in the objects list, you might try the following simple approach, trying to print each Node's abspath attribute:

```
objects = [
    Object('prog1.c'),
    Object('prog2.c', CCFLAGS='-DFOO'),
]
Program(objects)
```

```
for object_file in objects:
    print(object_file.abspath)
```

This does not work as expected because each call to `str` is operating an embedded list returned by each `Object` call, not on the underlying Nodes within those lists:

```
% scons -Q
AttributeError: 'NodeList' object has no attribute 'abspath':
  File "/home/my/project/SConstruct", line 8:
    print(object_file.abspath)
```

The solution is to use the `Flatten` function so that you can pass each Node to the `str` separately:

```
objects = [
    Object('prog1.c'),
    Object('prog2.c', CCFLAGS='-DFOO'),
]
Program(objects)

for object_file in Flatten(objects):
    print(object_file.abspath)
```

```
% scons -Q
/home/me/project/prog1.o
/home/me/project/prog2.o
cc -o prog1.o -c prog1.c
cc -o prog2.o -c -DFOO prog2.c
cc -o prog1 prog1.o prog2.o
```

26.6. Finding the Invocation Directory: the `GetLaunchDir` Function

If you need to find the directory from which the user invoked the **scons** command, you can use the `GetLaunchDir` function:

```
env = Environment(
    LAUNCHDIR = GetLaunchDir(),
)
env.Command('directory_build_info',
            '$LAUNCHDIR/build_info'
            Copy('$TARGET', '$SOURCE'))
```

Because SCons is usually invoked from the top-level directory in which the `SConstruct` file lives, the Python `os.getcwd()` is often equivalent. However, the SCons `-u`, `-U` and `-D` command-line options, when invoked from a subdirectory, will cause SCons to change to the directory in which the `SConstruct` file is found. When those options

are used, GetLaunchDir will still return the path to the user's invoking subdirectory, allowing the SConscript configuration to still get at configuration (or other) files from the originating directory.

26.7. Declaring Additional Outputs: the SideEffect Function

Sometimes the way an action is defined causes effects on files that SCons does not recognize as targets. The SideEffect method can be used to informs SCons about such files. This can be used just to flag a dependency for use in subsequent build steps, although there is usually a better way to do that. The primary use for the SideEffect method is to prevent two build steps from simultaneously modifying or accessing the same file in a way that could impact each other.

In this example, the rule to build file1 will also put data into log, which is used as a source for the command to generate file2, but log is unknown to SCons on a clean build: it neither exists, nor is it a target output by any builder. The SConscript uses SideEffect to inform SCons about the additional output file.

```
env = Environment()
f2 = env.Command(
    target='file2',
    source='log',
    action=Copy('$TARGET', '$SOURCE')
)
f1 = env.Command(
    target='file1',
    source=[],
    action='echo >$TARGET data1; echo >log updated file1'
)
env.SideEffect('log', f1)
```

Without the SideEffect, this build would fail with a message Source `log' not found, needed by target `file2', but now it can proceed:

```
% scons -Q
echo > file1 data1; echo >log updated file1
Copy("file2", "log")
```

However, it is better to actually identify log as a target, since in this case that's what it is:

```
env = Environment()
f2 = env.Command(
    target='file2',
    source='log',
    action=Copy('$TARGET', '$SOURCE')
)
f1 = env.Command(
    target=['file1', 'log'],
    source=[],
    action='echo >$TARGET data1; echo >log updated file1'
)
```

```
% scons -Q
echo > file1 data1; echo >log updated file1
Copy("file2", "log")
```

In general, SideEffect is not intended for the case when a command produces extra target files (that is, files which will be used as sources to other build steps). For example, the the Microsoft Visual C/C++ compiler is capable of performing incremental linking, for which it uses a status file - such that linking foo.exe also produces a foo.ilk, or uses it if it was already present, if the /INCREMENTAL option was supplied. Specifying foo.ilk as a side-effect of foo.exe is *not* a recommended use of SideEffect since foo.ilk is used by the link. SCons handles side-effect files slightly differently in its analysis of the dependency graph. When a command produces multiple output files, they should be specified as multiple targets of the call to the relevant builder function. The SideEffect function itself should really only be used when it's important to ensure that commands are not executed in parallel, such as when a "peripheral" file (such as a log file) may actually be updated by more than one command invocation.

Unfortunately, the tool which sets up the Program builder for the MSVC compiler chain does not come prebuilt with an understanding of the details of the .ilk example - that the target list would need to change in the presence of that specific option flag. Unlike the trivial example above where we could simply tell the Command builder there were two targets of the action, modifying the chain of events for a builder like Program, though not inherently complex, is definitely an advanced SCons topic. It's okay to use SideEffect here to get started, as long as it comes with an understanding that it's "not quite right". Perhaps leave a comment in the file as a reminder, if it does turn out to cause problems later.

So if the main use is to prevent parallelism problems, here is an example to illustrate. Say a program that you need to call to build a target file will also update a log file describing what the program does while building the target. The following configuration would have SCons invoke a hypothetical script named build (in the local directory) with command-line arguments telling it to write log information to a common logfile.txt file:

```
env = Environment()
env.Command(
    target='file1.out',
    source='file1.in',
    action='./build --log logfile.txt $SOURCE $TARGET'
)
env.Command(
    target='file2.out',
    source='file2.in',
    action='./build --log logfile.txt $SOURCE $TARGET'
)
```

This can cause problems when running the build in parallel if SCons decides to update both targets by running both program invocations at the same time. The multiple program invocations may interfere with each other writing to the common log file, leading at best to intermixed output in the log file, and at worst to an actual failed build (on a system like Windows, for example, where only one process at a time can open the log file for writing).

We can make sure that SCons does not run these build commands at the same time by using the SideEffect function to specify that updating the logfile.txt file is a side effect of building the specified file1 and file2 target files:

```
env = Environment()
f1 = env.Command(
    target='file1.out',
    source='file1.in',
    action='./build --log logfile.txt $SOURCE $TARGET'
```

```
)
f2 = env.Command(
    target='file2.out',
    source='file2.in',
    action='./build --log logfile.txt $SOURCE $TARGET'
)
env.SideEffect('logfile.txt', f1 + f2)
```

This makes sure the the two ./build steps are run sequentially, even with the `--jobs=2` in the command line:

```
% scons -Q --jobs=2
./build --log logfile.txt file1.in file1.out
./build --log logfile.txt file2.in file2.out
```

The `SideEffect` function can be called multiple times for the same side-effect file. In fact, the name used as a `SideEffect` does not even need to actually exist as a file on disk - SCons will still make sure that the relevant targets will be executed sequentially, not in parallel. The side effect is actually a pseudo-target, and SCons mainly cares whether nodes are listed as depending on it, not about its contents.

```
env = Environment()
f1 = env.Command('file1.out', [], action='echo >$TARGET data1')
env.SideEffect('not_really_updated', f1)
f2 = env.Command('file2.out', [], action='echo >$TARGET data2')
env.SideEffect('not_really_updated', f2)
```

```
% scons -Q --jobs=2
echo > file1.out data1
echo > file2.out data2
```

26.8. Virtual environments (virtualenvs)

Virtualenv is a tool to create isolated Python environments. A python application (such as SCons) may be executed within an activated virtualenv. The activation of virtualenv modifies current environment by defining some virtualenv-specific variables and modifying search PATH, such that executables installed within virtualenv's home directory are preferred over the ones installed outside of it.

Normally, SCons uses hard-coded PATH when searching for external executables, so it always picks-up executables from these pre-defined locations. This applies also to python interpreter, which is invoked by some custom SCons tools or test suites. This means, when running SCons in a virtualenv, an eventual invocation of python interpreter from SCons script will most probably jump out of virtualenv and execute python executable found in hard-coded SCons PATH, not the one which is executing SCons. Some users may consider this as an inconsistency.

This issue may be overcome by using the `--enable-virtualenv` option. The option automatically imports virtualenv-related environment variables to all created construction environment `env['ENV']`, and modifies SCons PATH appropriately to prefer virtualenv's executables. Setting environment variable `SCONS_ENABLE_VIRTUALENV=1` will have same effect. If virtualenv support is enabled system-vide by the environment variable, it may be suppressed with the `--ignore-virtualenv` option.

Inside of `SConscript`, a global function `Virtualenv` is available. It returns a path to virtualenv's home directory, or `None` if **scons** is not running from virtualenv. Note that this function returns a path even if **scons** is run from an unactivated virtualenv.

27 Using SCons with other build tools

Sometimes a project needs to interact with other projects in various ways. For example, many open source projects make use of components from other open source projects, and want to use those in their released form, not recode their builds into SCons. As another example, sometimes the flexibility and power of SCons is useful for managing the overall project, but developers might like faster incremental builds when making small changes by using a different tool.

This chapter shows some techniques for interacting with other projects and tools effectively from within SCons.

27.1. Creating a Compilation Database

Tooling to perform analysis and modification of source code often needs to know not only the source code itself, but also how it will be compiled, as the compilation line affects the behavior of macros, includes, etc. SCons has a record of this information once it has run, in the form of Actions associated with the sources, and can emit this information so tools can use it.

The Clang project has defined a *JSON Compilation Database*. This database is in common use as input into Clang tools and many IDEs and editors as well. See *JSON Compilation Database Format Specification* [https://clang.llvm.org/docs/JSONCompilationDatabase.html] for complete information. SCons can emit a compilation database in this format by enabling the `compilation_db` tool and calling the `CompilationDatabase` builder (*available since **scons** 4.0*).

The compilation database can be populated with source and output files either with paths relative to the top of the build, or using absolute paths. This is controlled by `COMPILATIONDB_USE_ABSPATH=(True|False)` which defaults to `False`. The entries in this file can be filtered by using `COMPILATIONDB_PATH_FILTER='pattern'` where the filter pattern is a string following the Python `fnmatch` [https://docs.python.org/3/library/fnmatch.html] syntax. This filtering can be used for outputting different build variants to different compilation database files.

The following example illustrates generating a compilation database containing absolute paths:

```
env = Environment(COMPILATIONDB_USE_ABSPATH=True)
env.Tool('compilation_db')
env.CompilationDatabase()
env.Program('hello.c')
```

```
% scons -Q
```

```
Building compilation database compile_commands.json
cc -o hello.o -c hello.c
cc -o hello hello.o
```

`compile_commands.json` contains:

```
[
{
"command": "gcc -o hello.o -c hello.c",
"directory": "/home/user/sandbox",
"file": "/home/user/sandbox/hello.c",
"output": "/home/user/sandbox/hello.o"
}
]
```

Notice that the generated database contains only an entry for the `hello.c/hello.o` pairing, and nothing for the generation of the final executable `hello` - the transformation of `hello.o` to `hello` does not have any information that affects interpretation of the source code, so it is not interesting to the compilation database.

Although it can be a little surprising at first glance, a compilation database target is, like any other target, subject to **scons** target selection rules. This means if you set a default target (that does not include the compilation database), or use command-line targets, it might not be selected for building. This can actually be an advantage, since you don't necessarily want to regenerate the compilation database every build. The following example shows selecting relative paths (the default) for output and source, and also giving a non-default name to the database. In order to be able to generate the database separately from building, an alias is set referring to the database, which can then be used as a target - here we are only building the compilation database target, not the code.

```
env = Environment()
env.Tool('compilation_db')
cdb = env.CompilationDatabase('compile_database.json')
Alias('cdb', cdb)
env.Program('test_main.c')
```

```
% scons -Q cdb
Building compilation database compile_database.json
```

`compile_database.json` contains:

```
[
{
"command": "gcc -o test_main.o -c test_main.c",
"directory": "/home/user/sandbox",
"file": "test_main.c",
"output": "test_main.o"
}
]
```

The following (incomplete) example shows using filtering to separate build variants. In the case of using variants, you want different compilation databases for each, since the build parameters differ, so the code analysis needs to see the

correct build lines for the 32-bit build and 64-bit build hinted at here. For simplicity of presentation, the example omits the setup details of the variant directories:

```
env = Environment()
env.Tool('compilation_db')

env1 = env.Clone()
env1['COMPILATIONDB_PATH_FILTER'] = 'build/linux32/*'
env1.CompilationDatabase('compile_commands-linux32.json')

env2 = env.Clone()
env2['COMPILATIONDB_PATH_FILTER'] = 'build/linux64/*'
env2.CompilationDatabase('compile_commands-linux64.json')
```

`compile_commands-linux32.json` contains:

```
[
{
"command": "gcc -m32 -o build/linux32/test_main.o -c test_main.c",
"directory": "/home/user/sandbox",
"file": "test_main.c",
"output": "build/linux32/test_main.o"
}
]
```

`compile_commands-linux64.json` contains:

```
[
{
"command": "gcc -m64 -o build/linux64/test_main.o -c test_main.c",
"directory": "/home/user/sandbox",
"file": "test_main.c",
"output": "build/linux64/test_main.o"
}
]
```

27.2. Ninja Build Generator

Note

This is an experimental new feature. It is subject to change and/or removal without depreciation 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**

To enable this feature you'll need to use one of the following

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

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

This tool will enabled creating a ninja build file from your SCons based build system. It can then invoke ninja to run your build. For most builds ninja will be significantly faster, but you may have to give up some accuracy. You are NOT advised to use this for production builds. It can however significantly speed up your build/debug/compile iterations.

It's not expected that the ninja builder will work for all builds at this point. It's still under active development. If you find that your build doesn't work with ninja please bring this to the users mailing list or devel channel on our discord server.

Specifically if your build has many (or even any) python function actions you may find that the ninja build will be slower as it will run ninja, which will then run SCons for each target created by a python action. To alleviate some of these, especially those python based actions built into SCons there is special logic to implement those actions via shell commands in the ninja build file.

Ninja Build System [https://ninja-build.org/]

Ninja File Format Specification [https://ninja-build.org/manual.html#ref_ninja_file]

28 Troubleshooting

The experience of configuring any software build tool to build a large code base usually, at some point, involves trying to figure out why the tool is behaving a certain way, and how to get it to behave the way you want. SCons is no different. This appendix contains a number of different ways in which you can get some additional insight into SCons' behavior.

Note that we're always interested in trying to improve how you can troubleshoot configuration problems. If you run into a problem that has you scratching your head, and which there just doesn't seem to be a good way to debug, odds are pretty good that someone else will run into the same problem, too. If so, please let the SCons development team know using the contact information at https://scons.org/contact.html so that we can use your feedback to try to come up with a better way to help you, and others, get the necessary insight into SCons behavior to help identify and fix configuration issues.

28.1. Why is That Target Being Rebuilt? the `--debug=explain` Option

Let's look at a simple example of a misconfigured build that causes a target to be rebuilt every time SCons is run:

```
# Intentionally misspell the output file name in the
# command used to create the file:
Command('file.out', 'file.in', 'cp $SOURCE file.oout')
```

(Note to Windows users: The POSIX cp command copies the first file named on the command line to the second file. In our example, it copies the `file.in` file to the `file.out` file.)

Now if we run SCons multiple times on this example, we see that it re-runs the cp command every time:

```
% scons -Q
cp file.in file.oout
% scons -Q
cp file.in file.oout
% scons -Q
cp file.in file.oout
```

In this example, the underlying cause is obvious: we've intentionally misspelled the output file name in the cp command, so the command doesn't actually build the `file.out` file that we've told SCons to expect. But if the

problem weren't obvious, it would be helpful to specify the `--debug=explain` option on the command line to have SCons tell us very specifically why it's decided to rebuild the target:

```
% scons -Q --debug=explain
scons: building `file.out' because it doesn't exist
cp file.in file.oout
```

If this had been a more complicated example involving a lot of build output, having SCons tell us that it's trying to rebuild the target file because it doesn't exist would be an important clue that something was wrong with the command that we invoked to build it.

Note that you can also use `--warn=target-not-built` which checks whether or not expected targets exist after a build rule is executed.

```
% scons -Q --warn=target-not-built
cp file.in file.oout

scons: warning: Cannot find target file.out after building
File "/home/bdeegan/devel/scons/git/as_scons/scripts/scons.py", line 98, in <module>
```

The `--debug=explain` option also comes in handy to help figure out what input file changed. Given a simple configuration that builds a program from three source files, changing one of the source files and rebuilding with the `--debug=explain` option shows very specifically why SCons rebuilds the files that it does:

```
% scons -Q
cc -o file1.o -c file1.c
cc -o file2.o -c file2.c
cc -o file3.o -c file3.c
cc -o prog file1.o file2.o file3.o
%     [CHANGE THE CONTENTS OF file2.c]
% scons -Q --debug=explain
scons: rebuilding `file2.o' because `file2.c' changed
cc -o file2.o -c file2.c
scons: rebuilding `prog' because `file2.o' changed
cc -o prog file1.o file2.o file3.o
```

This becomes even more helpful in identifying when a file is rebuilt due to a change in an implicit dependency, such as an incuded `.h` file. If the `file1.c` and `file3.c` files in our example both included a `hello.h` file, then changing that included file and re-running SCons with the `--debug=explain` option will pinpoint that it's the change to the included file that starts the chain of rebuilds:

```
% scons -Q
cc -o file1.o -c -I. file1.c
cc -o file2.o -c -I. file2.c
cc -o file3.o -c -I. file3.c
cc -o prog file1.o file2.o file3.o
%     [CHANGE THE CONTENTS OF hello.h]
% scons -Q --debug=explain
scons: rebuilding `file1.o' because `hello.h' changed
cc -o file1.o -c -I. file1.c
scons: rebuilding `file3.o' because `hello.h' changed
cc -o file3.o -c -I. file3.c
scons: rebuilding `prog' because:
           `file1.o' changed
           `file3.o' changed
```

```
cc -o prog file1.o file2.o file3.o
```

(Note that the `--debug=explain` option will only tell you why SCons decided to rebuild necessary targets. It does not tell you what files it examined when deciding *not* to rebuild a target file, which is often a more valuable question to answer.)

28.2. What's in That Construction Environment? the Dump Method

When you create a construction environment, SCons populates it with construction variables that are set up for various compilers, linkers and utilities that it finds on your system. Although this is usually helpful and what you want, it might be frustrating if SCons doesn't set certain variables that you expect to be set. In situations like this, it's sometimes helpful to use the construction environment `Dump` method to print all or some of the construction variables. Note that the `Dump` method *returns* the representation of the variables in the environment for you to print (or otherwise manipulate):

```
env = Environment()
print(env.Dump())
```

On a POSIX system with gcc installed, this might generate:

```
% scons
scons: Reading SConscript files ...
{ 'BUILDERS': { '_InternalInstall': <function InstallBuilderWrapper at 0x700000>,
                '_InternalInstallAs': <function InstallAsBuilderWrapper at 0x700000>,
                '_InternalInstallVersionedLib': <function InstallVersionedBuilderWrapper a
  'CONFIGUREDIR': '#/.sconf_temp',
  'CONFIGURELOG': '#/config.log',
  'CPPSUFFIXES': [ '.c',
                   '.C',
                   '.cxx',
                   '.cpp',
                   '.c++',
                   '.cc',
                   '.h',
                   '.H',
                   '.hxx',
                   '.hpp',
                   '.hh',
                   '.F',
                   '.fpp',
                   '.FPP',
                   '.m',
                   '.mm',
                   '.S',
                   '.spp',
                   '.SPP',
                   '.sx'],
  'DSUFFIXES': ['.d'],
  'Dir': <SCons.Defaults.Variable_Method_Caller object at 0x700000>,
  'Dirs': <SCons.Defaults.Variable_Method_Caller object at 0x700000>,
  'ENV': {'PATH': '/usr/local/bin:/opt/bin:/bin:/usr/bin:/snap/bin'},
```

```
  'ESCAPE': <function escape at 0x700000>,
  'File': <SCons.Defaults.Variable_Method_Caller object at 0x700000>,
  'HOST_ARCH': 'x86_64',
  'HOST_OS': 'posix',
  'IDLSUFFIXES': ['.idl', '.IDL'],
  'INSTALL': <function copyFunc at 0x700000>,
  'INSTALLVERSIONEDLIB': <function copyFuncVersionedLib at 0x700000>,
  'LIBPREFIX': 'lib',
  'LIBPREFIXES': ['$LIBPREFIX'],
  'LIBSUFFIX': '.a',
  'LIBSUFFIXES': ['$LIBSUFFIX', '$SHLIBSUFFIX'],
  'MAXLINELENGTH': 128072,
  'OBJPREFIX': '',
  'OBJSUFFIX': '.o',
  'PLATFORM': 'posix',
  'PROGPREFIX': '',
  'PROGSUFFIX': '',
  'PSPAWN': <function piped_env_spawn at 0x700000>,
  'RDirs': <SCons.Defaults.Variable_Method_Caller object at 0x700000>,
  'SCANNERS': [<SCons.Scanner.ScannerBase object at 0x700000>],
  'SHELL': 'sh',
  'SHLIBPREFIX': '$LIBPREFIX',
  'SHLIBSUFFIX': '.so',
  'SHOBJPREFIX': '$OBJPREFIX',
  'SHOBJSUFFIX': '$OBJSUFFIX',
  'SPAWN': <function subprocess_spawn at 0x700000>,
  'TARGET_ARCH': None,
  'TARGET_OS': None,
  'TEMPFILE': <class 'SCons.Platform.TempFileMunge'>,
  'TEMPFILEARGESCFUNC': <function quote_spaces at 0x700000>,
  'TEMPFILEARGJOIN': ' ',
  'TEMPFILEPREFIX': '@',
  'TOOLS': ['install', 'install'],
  '_CPPDEFFLAGS': '${_defines(CPPDEFPREFIX, CPPDEFINES, CPPDEFSUFFIX, __env__, '
                  'TARGET, SOURCE)}',
  '_CPPINCFLAGS': '${_concat(INCPREFIX, CPPPATH, INCSUFFIX, __env__, RDirs, '
                  'TARGET, SOURCE, affect_signature=False)}',
  '_LIBDIRFLAGS': '${_concat(LIBDIRPREFIX, LIBPATH, LIBDIRSUFFIX, __env__, '
                  'RDirs, TARGET, SOURCE, affect_signature=False)}',
  '_LIBFLAGS': '${_concat(LIBLINKPREFIX, LIBS, LIBLINKSUFFIX, __env__)}',
  '__DRPATH': '$_DRPATH',
  '__DSHLIBVERSIONFLAGS': '${__libversionflags(__env__,"DSHLIBVERSION","_DSHLIBVERSIONFLAG
  '__LDMODULEVERSIONFLAGS': '${__libversionflags(__env__,"LDMODULEVERSION","_LDMODULEVERSI
  '__RPATH': '$_RPATH',
  '__SHLIBVERSIONFLAGS': '${__libversionflags(__env__,"SHLIBVERSION","_SHLIBVERSIONFLAGS")
  '__lib_either_version_flag': <function __lib_either_version_flag at 0x700000>,
  '__libversionflags': <function __libversionflags at 0x700000>,
  '_concat': <function _concat at 0x700000>,
  '_defines': <function _defines at 0x700000>,
  '_stripixes': <function _stripixes at 0x700000>}
scons: done reading SConscript files.
scons: Building targets ...
scons: `.' is up to date.
scons: done building targets.
```

On a Windows system with Visual C++ the output might look like:

```
C:\>scons
scons: Reading SConscript files ...
{ 'BUILDERS': { 'Object': <SCons.Builder.CompositeBuilder object at 0x700000>,
                'PCH': <SCons.Builder.BuilderBase object at 0x700000>,
                'RES': <SCons.Builder.BuilderBase object at 0x700000>,
                'SharedObject': <SCons.Builder.CompositeBuilder object at 0x700000>,
                'StaticObject': <SCons.Builder.CompositeBuilder object at 0x700000>,
                '_InternalInstall': <function InstallBuilderWrapper at 0x700000>,
                '_InternalInstallAs': <function InstallAsBuilderWrapper at 0x700000>,
                '_InternalInstallVersionedLib': <function InstallVersionedBuilderWrapper a
  'CC': 'cl',
  'CCCOM': <SCons.Action.FunctionAction object at 0x700000>,
  'CCFLAGS': ['/nologo'],
  'CCPCHFLAGS': <function gen_ccpchflags at 0x700000>,
  'CCPDBFLAGS': ['${(PDB and "/Z7") or ""}'],
  'CFILESUFFIX': '.c',
  'CFLAGS': [],
  'CONFIGUREDIR': '#/.sconf_temp',
  'CONFIGURELOG': '#/config.log',
  'CPPDEFPREFIX': '/D',
  'CPPDEFSUFFIX': '',
  'CPPSUFFIXES': [ '.c',
                   '.C',
                   '.cxx',
                   '.cpp',
                   '.c++',
                   '.cc',
                   '.h',
                   '.H',
                   '.hxx',
                   '.hpp',
                   '.hh',
                   '.F',
                   '.fpp',
                   '.FPP',
                   '.m',
                   '.mm',
                   '.S',
                   '.spp',
                   '.SPP',
                   '.sx'],
  'CXX': '$CC',
  'CXXCOM': '${TEMPFILE("$CXX $_MSVC_OUTPUT_FLAG /c $CHANGED_SOURCES $CXXFLAGS '
            '$CCFLAGS $_CCCOMCOM","$CXXCOMSTR")}',
  'CXXFILESUFFIX': '.cc',
  'CXXFLAGS': ['$(', '/TP', '$)'],
  'DSUFFIXES': ['.d'],
  'Dir': <SCons.Defaults.Variable_Method_Caller object at 0x700000>,
  'Dirs': <SCons.Defaults.Variable_Method_Caller object at 0x700000>,
  'ENV': { 'PATH': 'C:\\WINDOWS\\System32',
           'PATHEXT': '.COM;.EXE;.BAT;.CMD',
           'SystemRoot': 'C:\\WINDOWS'},
```

```
'ESCAPE': <function escape at 0x700000>,
'File': <SCons.Defaults.Variable_Method_Caller object at 0x700000>,
'HOST_ARCH': 'x86_64',
'HOST_OS': 'win32',
'IDLSUFFIXES': ['.idl', '.IDL'],
'INCPREFIX': '/I',
'INCSUFFIX': '',
'INSTALL': <function copyFunc at 0x700000>,
'INSTALLVERSIONEDLIB': <function copyFuncVersionedLib at 0x700000>,
'LEXUNISTD': ['--nounistd'],
'LIBPREFIX': '',
'LIBPREFIXES': ['$LIBPREFIX'],
'LIBSUFFIX': '.lib',
'LIBSUFFIXES': ['$LIBSUFFIX'],
'MAXLINELENGTH': 2048,
'MSVC_SETUP_RUN': True,
'OBJPREFIX': '',
'OBJSUFFIX': '.obj',
'PCHCOM': '$CXX /Fo${TARGETS[1]} $CXXFLAGS $CCFLAGS $CPPFLAGS $_CPPDEFFLAGS '
          '$_CPPINCFLAGS /c $SOURCES /Yc$PCHSTOP /Fp${TARGETS[0]} '
          '$CCPDBFLAGS $PCHPDBFLAGS',
'PCHPDBFLAGS': ['${(PDB and "/Yd") or ""}'],
'PLATFORM': 'win32',
'PROGPREFIX': '',
'PROGSUFFIX': '.exe',
'PSPAWN': <function piped_spawn at 0x700000>,
'RC': 'rc',
'RCCOM': <SCons.Action.FunctionAction object at 0x700000>,
'RCFLAGS': ['/nologo'],
'RCSUFFIXES': ['.rc', '.rc2'],
'RDirs': <SCons.Defaults.Variable_Method_Caller object at 0x700000>,
'SCANNERS': [<SCons.Scanner.ScannerBase object at 0x700000>],
'SHCC': '$CC',
'SHCCCOM': <SCons.Action.FunctionAction object at 0x700000>,
'SHCCFLAGS': ['$CCFLAGS'],
'SHCFLAGS': ['$CFLAGS'],
'SHCXX': '$CXX',
'SHCXXCOM': '${TEMPFILE("$SHCXX $_MSVC_OUTPUT_FLAG /c $CHANGED_SOURCES '
            '$SHCXXFLAGS $SHCCFLAGS $_CCCOMCOM","$SHCXXCOMSTR")}',
'SHCXXFLAGS': ['$CXXFLAGS'],
'SHELL': None,
'SHLIBPREFIX': '',
'SHLIBSUFFIX': '.dll',
'SHOBJPREFIX': '$OBJPREFIX',
'SHOBJSUFFIX': '$OBJSUFFIX',
'SPAWN': <function spawn at 0x700000>,
'STATIC_AND_SHARED_OBJECTS_ARE_THE_SAME': 1,
'TARGET_ARCH': None,
'TARGET_OS': None,
'TEMPFILE': <class 'SCons.Platform.TempFileMunge'>,
'TEMPFILEARGESCFUNC': <function quote_spaces at 0x700000>,
'TEMPFILEARGJOIN': '\n',
'TEMPFILEPREFIX': '@',
'TOOLS': ['msvc', 'install', 'install'],
```

```
  'VSWHERE': None,
  '_CCCOMCOM': '$CPPFLAGS $_CPPDEFFLAGS $_CPPINCFLAGS $CCPCHFLAGS $CCPDBFLAGS',
  '_CPPDEFFLAGS': '${_defines(CPPDEFPREFIX, CPPDEFINES, CPPDEFSUFFIX, __env__, '
                  'TARGET, SOURCE)}',
  '_CPPINCFLAGS': '${_concat(INCPREFIX, CPPPATH, INCSUFFIX, __env__, RDirs, '
                  'TARGET, SOURCE, affect_signature=False)}',
  '_LIBDIRFLAGS': '${_concat(LIBDIRPREFIX, LIBPATH, LIBDIRSUFFIX, __env__, '
                  'RDirs, TARGET, SOURCE, affect_signature=False)}',
  '_LIBFLAGS': '${_concat(LIBLINKPREFIX, LIBS, LIBLINKSUFFIX, __env__)}',
  '_MSVC_OUTPUT_FLAG': <function msvc_output_flag at 0x700000>,
  '__DSHLIBVERSIONFLAGS': '${__libversionflags(__env__,"DSHLIBVERSION","_DSHLIBVERSIONFLAG
  '__LDMODULEVERSIONFLAGS': '${__libversionflags(__env__,"LDMODULEVERSION","_LDMODULEVERSI
  '__SHLIBVERSIONFLAGS': '${__libversionflags(__env__,"SHLIBVERSION","_SHLIBVERSIONFLAGS")
  '__lib_either_version_flag': <function __lib_either_version_flag at 0x700000>,
  '__libversionflags': <function __libversionflags at 0x700000>,
  '_concat': <function _concat at 0x700000>,
  '_defines': <function _defines at 0x700000>,
  '_stripixes': <function _stripixes at 0x700000>}
scons: done reading SConscript files.
scons: Building targets ...
scons: `.' is up to date.
scons: done building targets.
```

The construction environments in these examples have actually been restricted to just gcc and Visual C++, respectively. In a real-life situation, the construction environments will likely contain a great many more variables. Also note that we've massaged the example output above to make the memory address of all objects a constant 0x700000. In reality, you would see a different hexadecimal number for each object.

To make it easier to see just what you're interested in, the Dump method allows you to specify a specific constrcution variable that you want to disply. For example, it's not unusual to want to verify the external environment used to execute build commands, to make sure that the PATH and other environment variables are set up the way they should be. You can do this as follows:

```
env = Environment()
print(env.Dump('ENV'))
```

Which might display the following when executed on a POSIX system:

```
% scons
scons: Reading SConscript files ...
{'PATH': '/usr/local/bin:/opt/bin:/bin:/usr/bin:/snap/bin'}
scons: done reading SConscript files.
scons: Building targets ...
scons: `.' is up to date.
scons: done building targets.
```

And the following when executed on a Windows system:

```
C:\>scons
scons: Reading SConscript files ...
{ 'PATH': 'C:\\WINDOWS\\System32:/usr/bin',
  'PATHEXT': '.COM;.EXE;.BAT;.CMD',
  'SystemRoot': 'C:\\WINDOWS'}
scons: done reading SConscript files.
```

```
scons: Building targets ...
scons: `.' is up to date.
scons: done building targets.
```

28.3. What Dependencies Does SCons Know About? the `--tree` Option

Sometimes the best way to try to figure out what SCons is doing is simply to take a look at the dependency graph that it constructs based on your `SConscript` files. The `--tree` option will display all or part of the SCons dependency graph in an "ASCII art" graphical format that shows the dependency hierarchy.

For example, given the following input `SConstruct` file:

```
env = Environment(CPPPATH = ['.'])
env.Program('prog', ['f1.c', 'f2.c', 'f3.c'])
```

Running SCons with the `--tree=all` option yields:

```
% scons -Q --tree=all
cc -o f1.o -c -I. f1.c
cc -o f2.o -c -I. f2.c
cc -o f3.o -c -I. f3.c
cc -o prog f1.o f2.o f3.o
+-.
  +-SConstruct
  +-f1.c
  +-f1.o
  | +-f1.c
  | +-inc.h
  +-f2.c
  +-f2.o
  | +-f2.c
  | +-inc.h
  +-f3.c
  +-f3.o
  | +-f3.c
  | +-inc.h
  +-inc.h
  +-prog
    +-f1.o
    | +-f1.c
    | +-inc.h
    +-f2.o
    | +-f2.c
    | +-inc.h
    +-f3.o
      +-f3.c
      +-inc.h
```

The tree will also be printed when the `-n` (no execute) option is used, which allows you to examine the dependency graph for a configuration without actually rebuilding anything in the tree.

By default SCons uses "ASCII art" to draw the tree. It is possible to use line-drawing characters (Unicode calls these Box Drawing) to make a nicer display. To do this, add the `linedraw` qualifier:

```
% scons -Q --tree=all,linedraw
cc -o f1.o -c -I. f1.c
cc -o f2.o -c -I. f2.c
cc -o f3.o -c -I. f3.c
cc -o prog f1.o f2.o f3.o
###.
  ##SConstruct
  ##f1.c
  ###f1.o
  # ##f1.c
  # ##inc.h
  ##f2.c
  ###f2.o
  # ##f2.c
  # ##inc.h
  ##f3.c
  ###f3.o
  # ##f3.c
  # ##inc.h
  ##inc.h
  ###prog
    ###f1.o
    # ##f1.c
    # ##inc.h
    ###f2.o
    # ##f2.c
    # ##inc.h
    ###f3.o
      ##f3.c
      ##inc.h
```

The `--tree` option only prints the dependency graph for the specified targets (or the default target(s) if none are specified on the command line). So if you specify a target like `f2.o` on the command line, the `--tree` option will only print the dependency graph for that file:

```
% scons -Q --tree=all f2.o
cc -o f2.o -c -I. f2.c
+-f2.o
  +-f2.c
  +-inc.h
```

This is, of course, useful for restricting the output from a very large build configuration to just a portion in which you're interested. Multiple targets are fine, in which case a tree will be printed for each specified target:

```
% scons -Q --tree=all f1.o f3.o
cc -o f1.o -c -I. f1.c
+-f1.o
  +-f1.c
  +-inc.h
cc -o f3.o -c -I. f3.c
+-f3.o
  +-f3.c
```

```
  +-inc.h
```

The *`status`* argument may be used to tell SCons to print status information about each file in the dependency graph:

```
% scons -Q --tree=status
cc -o f1.o -c -I. f1.c
cc -o f2.o -c -I. f2.c
cc -o f3.o -c -I. f3.c
cc -o prog f1.o f2.o f3.o
 E         = exists
  R        = exists in repository only
   b       = implicit builder
   B       = explicit builder
    S      = side effect
     P     = precious
      A    = always build
       C   = current
        N  = no clean
         H = no cache

[E b      ]+-.
[E     C  ]  +-SConstruct
[E     C  ]  +-f1.c
[E B   C  ]  +-f1.o
[E     C  ]  | +-f1.c
[E     C  ]  | +-inc.h
[E     C  ]  +-f2.c
[E B   C  ]  +-f2.o
[E     C  ]  | +-f2.c
[E     C  ]  | +-inc.h
[E     C  ]  +-f3.c
[E B   C  ]  +-f3.o
[E     C  ]  | +-f3.c
[E     C  ]  | +-inc.h
[E     C  ]  +-inc.h
[E B   C  ]  +-prog
[E B   C  ]    +-f1.o
[E     C  ]    | +-f1.c
[E     C  ]    | +-inc.h
[E B   C  ]    +-f2.o
[E     C  ]    | +-f2.c
[E     C  ]    | +-inc.h
[E B   C  ]    +-f3.o
[E     C  ]      +-f3.c
[E     C  ]      +-inc.h
```

Note that `--tree=all,status` is equivalent; the *`all`* is assumed if only *`status`* is present. As an alternative to *`all`*, you can specify `--tree=derived` to have SCons only print derived targets in the tree output, skipping source files (like `.c` and `.h` files):

```
% scons -Q --tree=derived
cc -o f1.o -c -I. f1.c
cc -o f2.o -c -I. f2.c
cc -o f3.o -c -I. f3.c
cc -o prog f1.o f2.o f3.o
```

```
+-.
  +-f1.o
  +-f2.o
  +-f3.o
  +-prog
    +-f1.o
    +-f2.o
    +-f3.o
```

You can use the *`status`* modifier with *`derived`* as well:

```
% scons -Q --tree=derived,status
cc -o f1.o -c -I. f1.c
cc -o f2.o -c -I. f2.c
cc -o f3.o -c -I. f3.c
cc -o prog f1.o f2.o f3.o
 E         = exists
  R        = exists in repository only
   b       = implicit builder
   B       = explicit builder
    S      = side effect
     P     = precious
      A    = always build
       C   = current
        N  = no clean
         H = no cache

[E b      ]+-.
[E B   C  ]  +-f1.o
[E B   C  ]  +-f2.o
[E B   C  ]  +-f3.o
[E B   C  ]  +-prog
[E B   C  ]    +-f1.o
[E B   C  ]    +-f2.o
[E B   C  ]    +-f3.o
```

Note that the order of the `--tree=` arguments doesn't matter; `--tree=status,derived` is completely equivalent.

The default behavior of the `--tree` option is to repeat all of the dependencies each time the library dependency (or any other dependency file) is encountered in the tree. If certain target files share other target files, such as two programs that use the same library:

```
env = Environment(CPPPATH = ['.'],
                  LIBS = ['foo'],
                  LIBPATH = ['.'])
env.Library('foo', ['f1.c', 'f2.c', 'f3.c'])
env.Program('prog1.c')
env.Program('prog2.c')
```

Then there can be a *lot* of repetition in the `--tree=` output:

```
% scons -Q --tree=all
```

```
cc -o f1.o -c -I. f1.c
cc -o f2.o -c -I. f2.c
cc -o f3.o -c -I. f3.c
ar rc libfoo.a f1.o f2.o f3.o
ranlib libfoo.a
cc -o prog1.o -c -I. prog1.c
cc -o prog1 prog1.o -L. -lfoo
cc -o prog2.o -c -I. prog2.c
cc -o prog2 prog2.o -L. -lfoo
+-.
  +-SConstruct
  +-f1.c
  +-f1.o
  | +-f1.c
  | +-inc.h
  +-f2.c
  +-f2.o
  | +-f2.c
  | +-inc.h
  +-f3.c
  +-f3.o
  | +-f3.c
  | +-inc.h
  +-inc.h
  +-libfoo.a
  | +-f1.o
  | | +-f1.c
  | | +-inc.h
  | +-f2.o
  | | +-f2.c
  | | +-inc.h
  | +-f3.o
  |   +-f3.c
  |   +-inc.h
  +-prog1
  | +-prog1.o
  | | +-prog1.c
  | | +-inc.h
  | +-libfoo.a
  |   +-f1.o
  |   | +-f1.c
  |   | +-inc.h
  |   +-f2.o
  |   | +-f2.c
  |   | +-inc.h
  |   +-f3.o
  |     +-f3.c
  |     +-inc.h
  +-prog1.c
  +-prog1.o
  | +-prog1.c
  | +-inc.h
  +-prog2
  | +-prog2.o
```

```
  | | +-prog2.c
  | | +-inc.h
  | +-libfoo.a
  |   +-f1.o
  |   | +-f1.c
  |   | +-inc.h
  |   +-f2.o
  |   | +-f2.c
  |   | +-inc.h
  |   +-f3.o
  |     +-f3.c
  |     +-inc.h
  +-prog2.c
  +-prog2.o
    +-prog2.c
    +-inc.h
```

In a large configuration with many internal libraries and include files, this can very quickly lead to huge output trees. To help make this more manageable, a *`prune`* modifier may be added to the option list, in which case SCons will print the name of a target that has already been visited during the tree-printing in square brackets (`[]`) as an indication that the dependencies of the target file may be found by looking farther up the tree:

```
% scons -Q --tree=prune
cc -o f1.o -c -I. f1.c
cc -o f2.o -c -I. f2.c
cc -o f3.o -c -I. f3.c
ar rc libfoo.a f1.o f2.o f3.o
ranlib libfoo.a
cc -o prog1.o -c -I. prog1.c
cc -o prog1 prog1.o -L. -lfoo
cc -o prog2.o -c -I. prog2.c
cc -o prog2 prog2.o -L. -lfoo
+-.
  +-SConstruct
  +-f1.c
  +-f1.o
  | +-f1.c
  | +-inc.h
  +-f2.c
  +-f2.o
  | +-f2.c
  | +-inc.h
  +-f3.c
  +-f3.o
  | +-f3.c
  | +-inc.h
  +-inc.h
  +-libfoo.a
  | +-[f1.o]
  | +-[f2.o]
  | +-[f3.o]
  +-prog1
  | +-prog1.o
  | | +-prog1.c
```

```
  | | +-inc.h
  | +-[libfoo.a]
  +-prog1.c
  +-[prog1.o]
  +-prog2
  | +-prog2.o
  | | +-prog2.c
  | | +-inc.h
  | +-[libfoo.a]
  +-prog2.c
  +-[prog2.o]
```

Like the *status* keyword, the *prune* argument by itself is equivalent to `--tree=all,prune`.

28.4. How is SCons Constructing the Command Lines It Executes? the --debug=presub Option

Sometimes the command lines that SCons executes don't come out looking as you expect. In this case it may be useful to look at the strings before SCons performs substitution on them. This can be done with the `--debug=presub` option:

```
% scons -Q --debug=presub
Building prog.o with action:
  $CC -o $TARGET -c $CFLAGS $CCFLAGS $_CCOMCOM $SOURCES
cc -o prog.o -c -I. prog.c
Building prog with action:
  $SMART_LINKCOM
cc -o prog prog.o
```

28.5. Where is SCons Searching for Libraries? the --debug=findlibs Option

To get some insight into what library names SCons is searching for, and in which directories it is searching, Use the `--debug=findlibs` option. Given the following input `SConstruct` file:

```
env = Environment(LIBPATH = ['libs1', 'libs2'])
env.Program('prog.c', LIBS=['foo', 'bar'])
```

And the libraries `libfoo.a` and `libbar.a` in `libs1` and `libs2`, respectively, use of the `--debug=findlibs` option yields:

```
% scons -Q --debug=findlibs
  findlibs: looking for 'libfoo.a' in 'libs1' ...
  findlibs: ... FOUND 'libfoo.a' in 'libs1'
  findlibs: looking for 'libfoo.so' in 'libs1' ...
  findlibs: looking for 'libfoo.so' in 'libs2' ...
```

```
  findlibs: looking for 'libbar.a' in 'libs1' ...
  findlibs: looking for 'libbar.a' in 'libs2' ...
  findlibs: ... FOUND 'libbar.a' in 'libs2'
  findlibs: looking for 'libbar.so' in 'libs1' ...
  findlibs: looking for 'libbar.so' in 'libs2' ...
cc -o prog.o -c prog.c
cc -o prog prog.o -Llibs1 -Llibs2 -lfoo -lbar
```

28.6. Where is SCons Blowing Up? the --debug=stacktrace Option

In general, SCons tries to keep its error messages short and informative. That means we usually try to avoid showing the stack traces that are familiar to experienced Python programmers, since they usually contain much more information than is useful to most people.

For example, the following SConstruct file:

```
Program('prog.c')
```

Generates the following error if the prog.c file does not exist:

```
% scons -Q
scons: *** [prog.o] Source `prog.c' not found, needed by target `prog.o'.
```

In this case, the error is pretty obvious. But if it weren't, and you wanted to try to get more information about the error, the --debug=stacktrace option would show you exactly where in the SCons source code the problem occurs:

```
% scons -Q --debug=stacktrace
scons: *** [prog.o] Source `prog.c' not found, needed by target `prog.o'.
scons: internal stack trace:
  File "SCons/Job.py", line 203, in start
    task.prepare()
  File "SCons/Script/Main.py", line 178, in prepare
    return SCons.Taskmaster.OutOfDateTask.prepare(self)
  File "SCons/Taskmaster.py", line 186, in prepare
    executor.prepare()
  File "SCons/Executor.py", line 418, in prepare
    raise SCons.Errors.StopError(msg % (s, self.batches[0].targets[0]))
```

Of course, if you do need to dive into the SCons source code, we'd like to know if, or how, the error messages or troubleshooting options could have been improved to avoid that. Not everyone has the necessary time or Python skill to dive into the source code, and we'd like to improve SCons for those people as well...

28.7. How is SCons Making Its Decisions? the --taskmastertrace Option

The internal SCons subsystem that handles walking the dependency graph and controls the decision-making about what to rebuild is the *Taskmaster*. SCons supports a --taskmastertrace option that tells the Taskmaster to print information about the children (dependencies) of the various Nodes on its walk down the graph, which specific dependent Nodes are being evaluated, and in what order.

The `--taskmastertrace` option takes as an argument the name of a file in which to put the trace output, with `-` (a single hyphen) indicating that the trace messages should be printed to the standard output:

```
env = Environment(CPPPATH = ['.'])
env.Program('prog.c')
```

```
% scons -Q --taskmastertrace=- prog

Taskmaster: Looking for a node to evaluate
Taskmaster:     Considering node <no_state   0   'prog'> and its children:
Taskmaster:        <no_state   0   'prog.o'>
Taskmaster:      adjusted ref count: <pending    1   'prog'>, child 'prog.o'
Taskmaster:     Considering node <no_state   0   'prog.o'> and its children:
Taskmaster:        <no_state   0   'prog.c'>
Taskmaster:        <no_state   0   'inc.h'>
Taskmaster:      adjusted ref count: <pending    1   'prog.o'>, child 'prog.c'
Taskmaster:      adjusted ref count: <pending    2   'prog.o'>, child 'inc.h'
Taskmaster:     Considering node <no_state   0   'prog.c'> and its children:
Taskmaster: Evaluating <pending    0   'prog.c'>

Task.make_ready_current(): node <pending    0   'prog.c'>
Task.prepare():      node <up_to_date 0   'prog.c'>
Task.executed_with_callbacks(): node <up_to_date 0   'prog.c'>
Task.postprocess():  node <up_to_date 0   'prog.c'>
Task.postprocess():  removing <up_to_date 0   'prog.c'>
Task.postprocess():  adjusted parent ref count <pending    1   'prog.o'>

Taskmaster: Looking for a node to evaluate
Taskmaster:     Considering node <no_state   0   'inc.h'> and its children:
Taskmaster: Evaluating <pending    0   'inc.h'>

Task.make_ready_current(): node <pending    0   'inc.h'>
Task.prepare():      node <up_to_date 0   'inc.h'>
Task.executed_with_callbacks(): node <up_to_date 0   'inc.h'>
Task.postprocess():  node <up_to_date 0   'inc.h'>
Task.postprocess():  removing <up_to_date 0   'inc.h'>
Task.postprocess():  adjusted parent ref count <pending    0   'prog.o'>

Taskmaster: Looking for a node to evaluate
Taskmaster:     Considering node <pending    0   'prog.o'> and its children:
Taskmaster:        <up_to_date 0   'prog.c'>
Taskmaster:        <up_to_date 0   'inc.h'>
Taskmaster: Evaluating <pending    0   'prog.o'>

Task.make_ready_current(): node <pending    0   'prog.o'>
Task.prepare():      node <executing  0   'prog.o'>
Task.execute():      node <executing  0   'prog.o'>
cc -o prog.o -c -I. prog.c
Task.executed_with_callbacks(): node <executing  0   'prog.o'>
Task.postprocess():  node <executed   0   'prog.o'>
Task.postprocess():  removing <executed   0   'prog.o'>
Task.postprocess():  adjusted parent ref count <pending    0   'prog'>
```

```
Taskmaster: Looking for a node to evaluate
Taskmaster:     Considering node <pending    0   'prog'> and its children:
Taskmaster:        <executed   0   'prog.o'>
Taskmaster: Evaluating <pending    0   'prog'>

Task.make_ready_current(): node <pending    0   'prog'>
Task.prepare():      node <executing  0   'prog'>
Task.execute():      node <executing  0   'prog'>
cc -o prog prog.o
Task.executed_with_callbacks(): node <executing  0   'prog'>
Task.postprocess():  node <executed   0   'prog'>

Taskmaster: Looking for a node to evaluate
Taskmaster: No candidate anymore.
```

The `--taskmastertrace` option doesn't provide information about the actual calculations involved in deciding if a file is up-to-date, but it does show all of the dependencies it knows about for each Node, and the order in which those dependencies are evaluated. This can be useful as an alternate way to determine whether or not your SCons configuration, or the implicit dependency scan, has actually identified all the correct dependencies you want it to.

28.8. Watch SCons prepare targets for building: the `--debug=prepare` Option

Sometimes SCons doesn't build the target you want and it's difficult to figure out why. You can use the `--debug=prepare` option to see all the targets SCons is considering, and whether they are already up-to-date or not. The message is printed before SCons decides whether to build the target.

28.9. Why is a file disappearing? the `--debug=duplicate` Option

When using the `Duplicate` option to create variant dirs, sometimes you may find files not getting linked or copied to where you expect (or not at all), or files mysteriously disappearing. These are usually because of a misconfiguration of some kind in the SConscript files, but they can be tricky to debug. The `--debug=duplicate` option shows each time a variant file is unlinked and relinked from its source (or copied, depending on settings), and also shows a message for removing "stale" variant-dir files that no longer have a corresponding source file. It also prints a line for each target that's removed just before building, since that can also be mistaken for the same thing.

28.10. Keep it simple

Over the years, many developers have chosen to dive in and make vastly complicated build systems out of SCons, which sometimes don't work quite as expected. As a general rule, make sure you *need* to reach for a complex solution before you do so. SCons is mature software and has evolved over time to meet a lot of feature requests, so there is often an easier way to do something if you can just find it. The SCons community can be helpful here - the discussion lists and chat channels can be a way to find out if something can be done an easier way before embarking on an implementation.

When something does misbehave, trying to isolate the problem to a simple test case can really help. The work to create a reproducer often helps you spot the issue yourself, and a simple example is much easier for others to look over and possibly spot logical flaws, misuse of the API, or other ways something could have been done. In addition, if it turns

out there's actually a real SCons bug (we believe it's a high quality piece of software, but all software has some bugs), it's very likely the bug filing will result in a request for a simple reproducer anyway.

Appendix A. Construction Variables

This appendix contains descriptions of all of the construction variables that are *potentially* available "out of the box" in this version of SCons. Whether or not setting a construction variable in a construction environment will actually have an effect depends on whether any of the Tools and/or Builders that use the variable have been included in the construction environment.

In this appendix, we have appended the initial $ (dollar sign) to the beginning of each variable name when it appears in the text, but left off the dollar sign in the left-hand column where the name appears for each entry.

__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.

Appendix B. Builders

This appendix contains descriptions of all of the Builders that are *potentially* available "out of the box" in this version of SCons.

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')
```

Appendix C. Tools

This appendix contains descriptions of all of the Tools modules that are available "out of the box" in this version of SCons.

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`.

Appendix D. Functions and Environment Methods

This appendix contains descriptions of all of the function and construction environment methods in this version of SCons

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*

Query name	Command-line options	Notes
`file`	`-f`, `--file`, `--makefile`, `--sconstruct`	
`hash_format`	`--hash-format`	*since 4.2*
`help`	`-h`, `--help`	
`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)*

Settable name	Command-line options	Notes
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	
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.

Appendix E. Handling Common Tasks

There is a common set of simple tasks that many build configurations rely on as they become more complex. Most build tools have special purpose constructs for performing these tasks, but since `SConscript` files are Python scripts, you can use more flexible built-in Python services to perform these tasks. This appendix lists a number of these tasks and how to implement them in Python and SCons.

Example E.1. Wildcard globbing to create a list of filenames

```
files = Glob(wildcard)
```

Example E.2. Filename extension substitution

```
import os.path
filename = os.path.splitext(filename)[0]+extension
```

Example E.3. Appending a path prefix to a list of filenames

```
import os.path
filenames = [os.path.join(prefix, x) for x in filenames]
```

Example E.4. Substituting a path prefix with another one

```
if filename.find(old_prefix) == 0:
    filename = filename.replace(old_prefix, new_prefix)
```

Example E.5. Filtering a filename list to exclude/retain only a specific set of extensions

```
import os.path
filenames = [x for x in filenames if os.path.splitext(x)[1] in extensions]
```

Example E.6. The "backtick function": run a shell command and capture the output

```
import subprocess
output = subprocess.check_output(command)
```

Example E.7. Generating source code: how code can be generated and used by SCons

The Copy builders here could be any arbitrary shell or python function that produces one or more files. This example shows how to create those files and use them in SCons.

```
#### SConstruct
env = Environment()
env.Append(CPPPATH = "#")

## Header example
env.Append(BUILDERS =
   {'Copy1' : Builder(action = 'cat < $SOURCE > $TARGET',
                      suffix='.h', src_suffix='.bar')})
env.Copy1('test.bar') # produces test.h from test.bar.
env.Program('app','main.cpp') # indirectly depends on test.bar

## Source file example
env.Append(BUILDERS =
  {'Copy2' : Builder(action = 'cat < $SOURCE > $TARGET',
                     suffix='.cpp', src_suffix='.bar2')})
foo = env.Copy2('foo.bar2') # produces foo.cpp from foo.bar2.
env.Program('app2',['main2.cpp'] + foo) # compiles main2.cpp and foo.cpp into app2.
```

Where main.cpp looks like this:

```
#include "test.h"
```

produces this:

```
% scons -Q
cat < test.bar > test.h
cc -o app main.cpp
cat < foo.bar2 > foo.cpp
cc -o app2 main2.cpp foo.cpp
```