SCons Cookbook

If you want some help building a project with the the `SCons` build system, this Cookbook presents some recipes for getting various tasks done. The audience is programmers who already know a bit about the build system - for learning `SCons`, the User Guide is a good starting point.

The snippets are are generally presented with little or no explanation, rather they're expected to be "oh, that's how you could do that". While they are usually trimmed down to the simplest possible form, it is hopeful they will show the path to more complex scenarios.

If you would like to contribute to the Cookbook, it lives in its own project on GitHub, where you can submit Issues and Bug Reports and Pull Requests. All suggestions will be carefully considered.

Contents

Basic Compilation From a Single Source File

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

Note: Build the file by specifying the target as an argument (`scons foo` or `scons foo.exe`) or by specifying the current directory as the target (`scons .`).

Basic Compilation From Multiple Source Files

```
env = Environment()
env.Program(target='foo', source=Split('f1.c f2.c f3.c'))
```

Setting a Compilation Flag for an Environment

```
env = Environment(CCFLAGS='-g')
env.Program(target='foo', source='foo.c')
```

Setting a Compilation Flag while calling a Builder

You can add any variables to your builder call and it will override them - only for for that call. There will be no `--verbose` on the C Compiler command line here:

```
env = Environment(CCFLAGS='--verbose')
env.Program(target='foo', source='foo.c', CCFLAGS='-g')
```

You can also use variable substitution to append to values, incuding in an override. There *will* be a `--verbose` on the command line here:

```
env = Environment(CCFLAGS='--verbose')
env.Program(target='foo_2',source='foo2.c', CCFLAGS='$CCFLAGS -g')
```

Replacing Contents of a Construction Variable

The Replace method of a construction environmnent replaces the entire existing value. It can be convenient to use a Python dictionary and unpacking to set many values at once:

```
values = {"AFLAGS": "ONE", "BFLAGS": "TWO", "EXTRAFLAGS": ["THREE", "FOUR"]}
env.Replace(**values)
```

The method replaces the entire value for each variable, which is the same thing that happens when you do a direct assignment.

```
env = Environment(AFLAGS="ONE", BFLAGS="TWO")
env['BFLAGS'] = "THREE"
```

The advantages of *Replace* here is the ability to set several variables in one call, and that a “method” of a construction environment to modify it feels more object-oriented.

Sometimes, a construction variable is a container type and you need to replace only some element of it - a common case is preprocessor macros which take values (as for `CPPDEFINES`).
Neither *Replace* nor assignment will directly help with that, nor will using AppendUnique since it deals in exact matches - that is, you can move an element to the end, but not cause the value to change.

In this case, it may be useful to use a loop or comprehension to build a new list and use *Replace* or assignment to set the new list as the value:

```
env = Environment(CPPDEFINES=["BUILD", "DEBUG=0", "LINUX=1"])
...
clone = env.Clone()
cflags = ["DEBUG=1" if "DEBUG" in val else val for val in clone['CPPDEFINES']]
clone.Replace(CPPDEFINES=cflags)
```

Search The Local Directory For .h Files

Note: You do *not* need to set `CCFLAGS` to specify `-I` options by hand. `scons` will construct the right `-I` options from the contents of `CPPPATH`.

```
env = Environment(CPPPATH=['.'])
env.Program(target='foo', source='foo.c')
```

or

```
env = Environment()
env.AppendENVPath(CPPPATH=['.'])
env.Program(target='foo', source='foo.c')
```

Search Multiple Directories For .h Files

```
env = Environment(CPPPATH=['include1', 'include2'])
env.Program(target='foo', source='foo.c')
```

or

```
env = Environment()
env.AppendENVPath(CPPPATH=['include1', 'include2'])
env.Program(target='foo', source='foo.c')
```

Building a Static Library

```
env = Environment()
env.StaticLibrary(target='foo', source=Split('l1.c l2.c'))
env.StaticLibrary(target='bar', source=['l3.c', 'l4.c'])
```

Building a Shared Library

```
env = Environment()
env.SharedLibrary(target='foo', source=['l5.c', 'l6.c'])
env.SharedLibrary(target='bar', source=Split('l7.c l8.c'))
```

Linking a Local Library Into a Program

```
env = Environment(LIBS='mylib', LIBPATH=['.'])
env.Library(target='mylib', source=Split('l1.c l2.c'))
env.Program(target='prog', source=['p1.c', 'p2.c'])
```

Defining Your Own Builder Object

Notice that when you invoke the Builder, you can leave off the target file suffix, and `scons` will add it automatically.

```
bld = Builder(
    action='pdftex < $SOURCES > $TARGET',
    suffix='.pdf',
    src_suffix='.tex'
)
env = Environment(BUILDERS={'PDFBuilder': bld})
env.PDFBuilder(target='foo.pdf', source='foo.tex')

# The following creates "bar.pdf" from "bar.tex"
env.PDFBuilder(target='bar', source='bar')
```

Note that the above initialization replaces the default dictionary of Builders, so this construction environment can not be used call Builders like `Program`, `Object`, `StaticLibrary` etc. See the next example for an alternative.

Adding Your Own Builder Object to an Environment

```
bld = Builder(
    action='pdftex < $SOURCES > $TARGET'
    suffix='.pdf',
    src_suffix='.tex'
)
env = Environment()
env.Append(BUILDERS={'PDFBuilder': bld})
env.PDFBuilder(target='foo.pdf', source='foo.tex')
env.Program(target='bar', source='bar.c')
```

You also can use other Pythonic techniques to add to the BUILDERS construction variable, such as:

```
env = Environment()
env['BUILDERS']['PDFBuilder'] = bld
```

Defining Your Own Scanner Object

The following example shows adding an extremely simple scanner (`kfile_scan`) that doesn’t use a search path at all and simply returns the file names present on any `include` lines in the scanned file. This would implicitly assume that all included files live in the top-level directory:

```
import re

include_re = re.compile(r'^include\s+(\S+)$', re.M)

def kfile_scan(node, env, path, arg):
    contents = node.get_text_contents()
    includes = include_re.findall(contents)
    return env.File(includes)

kscan = Scanner(
    name='kfile',
    function=kfile_scan,
    argument=None,
    skeys=['.k'],
)

scanners = DefaultEnvironment()['SCANNERS']
scanners.append(kscan)
env = Environment(SCANNERS=scanners)

env.Command('foo', 'foo.k', 'kprocess < $SOURCES > $TARGET')

bar_in = File('bar.in')
env.Command('bar', bar_in, 'kprocess $SOURCES > $TARGET')
bar_in.target_scanner = kscan
```

It is important to note that you have to return a list of File nodes from the scan function, simple strings for the file names won't do. As in the examples shown here, you can use the `env.File` function of your current construction environment in order to create nodes on the fly from a sequence of file names with relative paths.

Here is a similar but more complete example that adds a scanner which searches a path of directories (specified as the MYPATH construction variable) for files that actually exist:

```
import re
import os

include_re = re.compile(r'^include\s+(\S+)$', re.M)

def my_scan(node, env, path, arg):
    contents = node.get_text_contents()
    includes = include_re.findall(contents)
    if not includes:
        return []
    results = []
    for inc in includes:
        for dir in path:
            file = str(dir) + os.sep + inc
            if os.path.exists(file):
                results.append(file)
                break
    return env.File(results)

scanner = Scanner(
    name='myscanner',
    function=my_scan,
    argument=None,
    skeys=['.x'],
    path_function=FindPathDirs('MYPATH'),
)

scanners = DefaultEnvironment()['SCANNERS']
scanners.append(scanner)
env = Environment(SCANNERS=scanners, MYPATH=['incs'])

env.Command('foo', 'foo.x', 'xprocess < $SOURCES > $TARGET')
```

The `FindPathDirs` function used in the previous example returns a function (actually a callable Python object) that will return a list of directories specified in the `MYPATH` construction variable. It lets `scons` detect the file `incs/foo.inc`, even if `foo.x` contains the line `include foo.inc` only. If you need to customize how the search path is derived, you would provide your own `path_function` argument when creating the Scanner object, as follows:

```
# MYPATH is a list of directories to search for files in
def pf(env, dir, target, source, arg):
    top_dir = Dir('#').abspath
    results = []
    if 'MYPATH' in env:
        for p in env['MYPATH']:
            results.append(top_dir + os.sep + p)
    return results


scanner = Scanner(
    name='myscanner',
    function=my_scan,
    argument=None,
    skeys=['.x'],
    path_function=pf
)
```

Selecting an Alternate Scanner

Sometimes you want to replace the existing scanner for certain types of files with an alternative. In the case where the scanner's *function* attribute is a dictionary, that dictionary serves as a dispatcher to other scanner functions, and you can simply update the dictionary. This example depends on the use of the global definition of `SourceFileScanner`, which exposes an *add_scanner* method, normally intended to add *additional* scanner mappings, but just as usable to do replacements.

In this case, the alternative scanner for C/C++ type files is patched in in place of the default:

```
import SCons.Scanner.C
CConditionalScanner = SCons.Scanner.C.CConditionalScanner()
SourceFileScanner.add_scanner('.c', CConditionalScanner)
# do more of those as needed for other suffixes (perhaps see $CPPSUFFIXES)
```

Creating a Hierarchical Build

Notice that the file names specified in a subdirectory's SConscript file are relative to that subdirectory.

`SConstruct`:

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

SConscript('sub/SConscript')
```

`sub/SConscript`:

```
env = Environment()
# Builds sub/foo from sub/foo.c
env.Program(target='foo', source='foo.c')

SConscript('dir/SConscript')
```

`sub/dir/SConscript`:

```
env = Environment()
# Builds sub/dir/foo from sub/dir/foo.c
env.Program(target='foo', source='foo.c')
```

Sharing Variables Between SConscript Files

You must explicitly call Export and Import for variables that you want to share between SConscript files.

`SConstruct`

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

Export('env')
SConscript('subdirectory/SConscript')
```

`subdirectory/SConscript`

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

Alternatively, you can use the `exports` keyword argument to pass a variable to an SConscript In this case the global definition that would be affected by calling `Export` is not modified. The SConscript file still needs to do the *Import*.

`SConstruct`

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

SConscript('subdirectory/SConscript', exports='env')
```

Building Multiple Variants From the Same Source

Use the `variant_dir` keyword argument to the SConscript function to establish one or more separate variant build directory trees for a given source directory:

`SConstruct`

```
cppdefines = ['FOO']
SConscript('src/SConscript', variant_dir='foo', exports='cppdefines')

cppdefines = ['BAR']
SConscript('src/SConscript', variant_dir='bar', exports='cppdefines')
```

`src/SConscript`

```
Import('cppdefines')
env = Environment(CPPDEFINES=cppdefines)
env.Program(target='src', source='src.c')
```

Note the use of the `exports` keyword argument to pass a different value for the `cppdefines` variable value each time we call the `SConscript` function.

Hierarchical Build of Two Libraries Linked With a Program

`SConstruct`:

```
env = Environment(LIBPATH=['#libA', '#libB'])
Export('env')
SConscript('libA/SConscript')
SConscript('libB/SConscript')
SConscript('Main/SConscript')
```

`libA/SConscript`:

```
Import('env')
env.Library('a', Split('a1.c a2.c a3.c'))
```

`libB/SConscript`:

```
Import('env')
env.Library('b', Split('b1.c b2.c b3.c'))
```

`Main/SConscript`:

```
Import('env')
e = env.Clone(LIBS=['a', 'b'])
e.Program('foo', Split('m1.c m2.c m3.c'))
```

The `#` in the `LIBPATH` directories specify that they're relative to the top-level directory, so they don't turn into `Main/libA` when they're used in `Main/SConscript`

Specifying only 'a' and 'b' for the library names allows `scons` to attach the appropriate library prefix and suffix for the current platform in creating the library filename (for example, `liba.a` on POSIX systems, `a.lib` on Windows).

Customizing construction variables from the command line.

The following would allow the C compiler to be specified on the command line or in the file `custom.py`.

```
vars = Variables('custom.py')
vars.Add('CC', 'The C compiler.')
env = Environment(variables=vars)
Help(vars.GenerateHelpText(env))
```

The user could specify the C compiler on the command line:

```
scons "CC=my_cc"
```

or in the `custom.py` file:

```
CC = 'my_cc'
```

or get documentation on the options:

```
$ scons -h

CC: The C compiler.
    default: None
    actual: cc
```

Using Microsoft Visual C++ precompiled headers

Since `windows.h` includes everything and the kitchen sink, it can take quite some time to compile it over and over again for a bunch of object files, so Microsoft provides a mechanism to compile a set of headers once and then include the previously compiled headers in any object file. This technology is called precompiled headers (PCH). The general recipe is to create a file named `StdAfx.cpp` that includes a single header named `StdAfx.h`, and then include every header you want to precompile in `StdAfx.h`, and finally include `"StdAfx.h` as the first header in all the source files you are compiling to object files. For example:

`StdAfx.h`:

```
#include <windows.h>
#include <my_big_header.h>
```

`StdAfx.cpp`:

```
#include <StdAfx.h>
```

`Foo.cpp`:

```
#include <StdAfx.h>

/* do some stuff */
```

`Bar.cpp`:

```
#include <StdAfx.h>

/* do some other stuff */
```

`SConstruct`:

```
env=Environment()
env['PCHSTOP'] = 'StdAfx.h'
env['PCH'] = env.PCH('StdAfx.cpp')[0]
env.Program('MyApp', ['Foo.cpp', 'Bar.cpp'])
```

For more information see the documentation for the `PCH` builder, and the `$PCH` and `$PCHSTOP` construction variables. To learn about the details of precompiled headers consult the MSDN documentation for `/Yc`, `/Yu`, and `/Yp`.

Using Microsoft Visual C++ external debugging information

Since including debugging information in programs and shared libraries can cause their size to increase significantly, Microsoft provides a mechanism for including the debugging information in an external file called a PDB file. `scons` supports PDB files through the `$PDB` construction variable.

`SConstruct`:

```
env=Environment()
env['PDB'] = 'MyApp.pdb'
env.Program('MyApp', ['Foo.cpp', 'Bar.cpp'])
```

Setting Up a Python Virtualenv for SCons

It is often useful to set up a virtualenv when working with a project that uses SCons to build. A virtualenv is a way to create an isolated execution environment - you can install whatever you need there, and change versions as needed, without affecting anything else on your system that uses Python. Here is an example session:

Linux Windows MacOS

```
$ cd Work
$ python -m venv myvenv
$ source myvenv/bin/activate
(myvenv) $ pip list --outdated
Package    Version Latest Type
---------- ------- ------ -----
pip        19.3.1  22.0.4 wheel
setuptools 41.6.0  62.1.0 wheel
WARNING: You are using pip version 19.3.1; however, version 22.0.4 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
(myvenv) $ pip install --upgrade pip setuptools scons
Collecting pip
  Downloading pip-22.0.4-py3-none-any.whl (2.1 MB)
Collecting setuptools
  Downloading setuptools-62.1.0-py3-none-any.whl (1.1 MB)
Collecting scons
  Downloading SCons-4.3.0-py3-none-any.whl (4.2 MB)
Installing collected packages: pip, setuptools, scons
  Found existing installation: pip 19.3.1
    Uninstalling pip-19.3.1:
      Successfully uninstalled pip-19.3.1
  Found existing installation: setuptools 41.6.0
    Uninstalling setuptools-41.6.0:
      Successfully uninstalled setuptools-41.6.0
Successfully installed pip-22.0.4 scons-4.3.0 setuptools-62.1.0
(myvenv) $ scons --version
SCons by Steven Knight et al.:
        SCons: v4.3.0.559790274f66fa55251f5754de34820a29c7327a, Tue, 16 Nov 2021 19:09:21 +0000, by
bdeegan on octodog
        SCons path: ['/home/user/Work/myvenv/lib64/python3.8/site-packages/SCons']
Copyright (c) 2001 - 2021 The SCons Foundation
(myvenv) $ deactivate
$
```

To use this virtualenv for work, repeat the step to activate it, go to your project, and install any additional requirements your project may have (skip this step if there are none).

Linux Windows MacOS

```
$ cd Work/myproj
$ source ~/Work/myvenv/bin/activate
(myvenv) $ pip install -r requirements.txt
```

Setting up a Cross Compile Environment

SCons doesn't have any specific support for cross-compiling, but it's not too hard to set up. There are a few things that you need to consider:

1. Cross-compilation tools often have alternate names, for example the C++ compiler might be called `arm-none-eabi-g++` instead of `g++`. Since SCons won't auto-detect these alternate names, and doesn't have a pre-defined toolname-prefix construction variable, you have to handle pointing to the right tools yourself by setting the appropriate construction variables.
2. Cross-compilation tools often live in a filesystem path of their own, not in "standard" locations from the viewpoint of, say a Linux system (Windows systems nearly always put an installed package in a unique path so this is a bit less of a surprise there). For eaxmple, a system might put all the tools in `/opt/arm-none-eabi/bin`. Even if such a path is in your own search path, SCons doesn't use that and instead uses its own idea of "standard locations". So you have to add any special path to the tools to SCons' idea of locations.
3. Cross-compiled code normally can't be run directly on the host system. It may be able to run in an emulator, or it may need to be sent over to the target machine. The upshot of this is that any build instructions that depend executing a locally built binary need some attention if a cross-compiled build is attempted. This could affect things like code- or data- generation tools that are made as part of the build, then are called to generate something which is used later in the build. It also affects builds which want to build test binaries and then execute the tests, all as part of the build process. Both of these scenarios would have to be adapted somehow.

Here is a very basic setup example:

```
import os

env_options = {
   "CC"    : "nios2-linux-gnu-gcc",
   "CXX"   : "nios2-linux-gnu-g++",
   "LD"    : "nios2-linux-gnu-g++",
   "AR"    : "nios2-linux-gnu-ar",
   "STRIP" : "nios2-linux-gnu-strip",
}

env = Environment(**env_options)
env.AppendENVPath('PATH': '/path/to/tools')
```