scikit-build Documentation

Release 0.10.0+31.gb2d4bc2.dirty

scikit-build team

Sep 30, 2019

User guide

scikit-build is an improved build system generator for CPython C/C++/Fortran/Cython extensions. It provides better support for additional compilers, build systems, cross compilation, and locating dependencies and their associated build requirements.

The **scikit-build** package is fundamentally just glue between the *setuptools* Python module and CMake.

To get started, see *this example*. For more examples, see scikit-build-sample-projects.

CHAPTER 1

Installation

1.1 Install package with pip

To install with pip:

```
$ pip install scikit-build
```

1.2 Install from source

To install scikit-build from the latest source, first obtain the source code:

```
$ git clone https://github.com/scikit-build/scikit-build
$ cd scikit-build
```

then install with:

```
$ pip install .
```

or:

```
$ pip install -e .
```

for development.

1.3 Dependencies

1.3.1 Python Packages

The project has a few common Python package dependencies. The runtime dependencies are:

```
wheel>=0.29.0
setuptools>=28.0.0
packaging
```

The build time dependencies (also required for development) are:

```
codecov>=2.0.5
coverage>=4.2
cython>=0.25.1
flake8>=3.0.4
pathlib2; python_version < '3.0'
path.py>=11.5.0
pytest>=4.5.0
pytest-cov>=2.7.1
pytest-mock>=1.10.4
pytest-runner>=5.1
pytest-virtualenv>=1.2.5
requests
six>=1.10.0
virtualenv
```

1.3.2 Compiler Toolchain

The same compiler toolchain used to build the CPython interpreter should also be available. Refer to the CPython Developer's Guide for details about the compiler toolchain for your operating system.

For example, on *Ubuntu Linux*, install with:

```
$ sudo apt-get install build-essential
```

On *Mac OSX*, install XCode to build packages for the system Python.

On Windows, install the version of Visual Studio used to create the target version of CPython

1.3.3 CMake

The easiest way to get CMake is *to add it to the pyproject.toml file*. With pip 10 or later, this will cause the CMake Python package to be downloaded and installed when your project is built.

To manually install the *cmake* package from PyPI:

```
$ pip install cmake
```

To install the *cmake* package in conda:

```
$ conda install -c conda-forge cmake
```

You can also download the standard CMake binaries for your platform.

Alternatively, build CMake from source with a C++ compiler if binaries are not available for your operating system.

CHAPTER 2

Why should I use scikit-build ?

Scikit-build is a replacement for distutils.core.Extension with the following advantages:

- provide better support for *additional compilers and build systems*
- first-class *cross-compilation* support
- location of dependencies and their associated build requirements

CHAPTER 3

Basic Usage

3.1 Example of setup.py, CMakeLists.txt and pyproject.toml

To use scikit-build in a project, place the following in your project's *setup.py* file:

```
from skbuild import setup  # This line replaces 'from setuptools import setup'
```

Your project now uses scikit-build instead of setuptools.

Next, add a `CMakeLists.txt` to describe how to build your extension. In the following example, a C++ extension named `_hello` is built:

```
cmake_minimum_required(VERSION 3.11.0)
project(hello)
find_package(PythonExtensions REQUIRED)

add_library(_hello MODULE hello/_hello.cxx)
python_extension_module(_hello)
install(TARGETS _hello LIBRARY DESTINATION hello)
```

Then, add a `pyproject.toml` to list the build system requirements:

```
[build-system]
requires = ["setuptools", "wheel", "scikit-build", "cmake", "ninja"]
```

Note: By default, scikit-build looks in the project top-level directory for a file named `CMakeLists.txt`. It will then invoke `cmake` executable specifying a *generator* matching the python being used.

3.2 Setup options

3.2.1 setuptools options

The section below documents some of the options accepted by the `setup()` function.

- `packages`: Explicitly list of all packages to include in the distribution. Setuptools will not recursively scan the source tree looking for any directory with an `__init__.py` file. To automatically generate the list of packages, see Using find_package().
- `package_dir`: A mapping of package to directory names
- `include_package_data`: If set to `True`, this tells setuptools to automatically include any data files it finds inside your package directories that are specified by your `MANIFEST.in` file. For more information, see the setuptools documentation section on Including Data Files.
- `package_data`: A dictionary mapping package names to lists of glob patterns. For a complete description and examples, see the setuptools documentation section on Including Data Files. You do not need to use this option if you are using include_package_data, unless you need to add e.g. files that are generated by your setup script and build process. (And are therefore not in source control or are files that you don't want to include in your source distribution.)
- `exclude_package_data`: Dictionary mapping package names to lists of glob patterns that should be excluded from the package directories. You can use this to trim back any excess files included by include_package_data. For a complete description and examples, see the setuptools documentation section on Including Data Files.
- `py_modules`: List all modules rather than listing packages. More details in the Listing individual modules section of the distutils documentation.
- `data_files`: Sequence of *(directory, files)* pairs. Each *(directory, files)* pair in the sequence specifies the installation directory and the files to install there. More details in the Installing Additional Files section of the setuptools documentation.
- `entry_points`: A dictionary mapping entry point group names to strings or lists of strings defining the entry points. Entry points are used to support dynamic discovery of services or plugins provided by a project. See Dynamic Discovery of Services and Plugins for details and examples of the format of this argument. In addition, this keyword is used to support Automatic Script Creation.
- `scripts`: List of python script relative paths. If the first line of the script starts with `#!` and contains the word *python*, the Distutils will adjust the first line to refer to the current interpreter location. More details in the Installing Scripts section of the distutils documentation.

New in version 0.8.0.

- `zip_safe`: A boolean indicating if the Python packages may be run directly from a zip file. If not already set, scikit-build sets this option to `False`. See Setting the zip_safe flag section of the setuptools documentation.

Note: As specified in the Wheel documentation, the `universal` and `python-tag` options have no effect.

3.2.2 scikit-build options

Scikit-build augments the `setup()` function with the following options:

- `cmake_args`: List of CMake options.

For example:

```
setup(
  [...]
  cmake_args=['-DSOME_FEATURE:BOOL=OFF']
  [...]
  )
```

- `cmake_install_dir`: relative directory where the CMake artifacts are installed. By default, it is set to an empty string.
- `cmake_source_dir`: Relative directory containing the project `CMakeLists.txt`. By default, it is set to the top-level directory where `setup.py` is found.

New in version 0.5.0.

- `cmake_with_sdist`: Boolean indicating if CMake should be executed when running *sdist* command. Setting this option to `True` is useful when part of the sources specified in `MANIFEST.in` are downloaded by CMake. By default, this option is `False`.

New in version 0.7.0.

- `cmake_languages`: Tuple of languages that the project use, by default *('C', 'CXX',)*. This option ensures that a generator is chosen that supports all languages for the project.
- `cmake_minimum_required_version`: String identifying the minimum version of CMake required to configure the project.

Scikit-build changes the following options:

New in version 0.7.0.

- `setup_requires`: If `cmake` is found in the list, it is explicitly installed first by scikit-build.

3.3 Command line options

```
usage: setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...] [skbuild_opts]
↪[cmake_configure_opts] [-- [cmake_opts] [-- [build_tool_opts]]]
or: setup.py --help [cmd1 cmd2 ...]
or: setup.py --help-commands
or: setup.py cmd --help
```

There are few types of options:

- *setuptools options*:
  - `[global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]`
  - `--help [cmd1 cmd2 ...]`
  - `cmd --help`
- *scikit-build options*: `[skbuild_opts]`
- *CMake configure options*: `[cmake_configure_opts]`
- *CMake options*: `[cmake_opts]`
- *build tool options*: `[build_tool_opts]`

setuptools, scikit-build and CMake configure options can be passed normally, the cmake and build_tool set of options needs to be separated by `--`:

```
Arguments following a "--" are passed directly to CMake (e.g. -DSOME_FEATURE:BOOL=ON).
Arguments following a second "--" are passed directly to  the build tool.
```

3.3.1 setuptools options

For more details, see the official documentation.

scikit-build extends the global set of setuptools options with:

New in version 0.4.0.

```
Global options:
  [...]
  --hide-listing      do not display list of files being included in the
                      distribution
```

New in version 0.5.0.

```
Global options:
  [...]
  --force-cmake       always run CMake
  --skip-cmake        do not run CMake
```

Note: As specified in the Wheel documentation, the `--universal` and `--python-tag` options have no effect.

3.3.2 scikit-build options

```
scikit-build options:
  --build-type       specify the CMake build type (e.g. Debug or Release)
  -G , --generator   specify the CMake build system generator
  -j N               allow N build jobs at once
  [...]
```

New in version 0.7.0.

```
scikit-build options:
  [...]
  --cmake-executable specify the path to the cmake executable
```

New in version 0.8.0.

```
scikit-build options:
  [...]
  --skip-generator-test  skip generator test when a generator is explicitly selected␣
↪using --generator
```

3.3.3 CMake Configure options

New in version 0.10.1.

These options are relevant when configuring a project and can be passed as global options using *setup.py* or *pip install*.

The CMake options accepted as global options are any of the following:

```
-C<initial-cache>            = Pre-load a script to populate the cache.
-D<var>[:<type>]=<value>     = Create or update a cmake cache entry.
```

Warning: The CMake configure option should be passed without spaces. For example, use *-DSOME_FEATURE:BOOL=ON* instead of *-D SOME_FEATURE:BOOL=ON*.

3.3.4 CMake options

These are any specific to CMake. See list of CMake options.

For example:

```
-DSOME_FEATURE:BOOL=OFF
```

3.3.5 build tool options

These are specific to the underlying build tool (e.g msbuild.exe, make, ninja).

CHAPTER 4

Advanced Usage

4.1 How to test if scikit-build is driving the compilation ?

To support the case of code base being built as both a standalone project and a python wheel, it is possible to test for the variable `SKBUILD`:

```
if(SKBUILD)
  message(STATUS "The project is built using scikit-build")
endif()
```

4.2 Adding cmake as building requirement only if not installed or too low a version

If systematically installing cmake wheel is not desired, the `setup_requires` list can be set using the following approach:

```
from packaging.version import LegacyVersion
from skbuild.exceptions import SKBuildError
from skbuild.cmaker import get_cmake_version

# Add CMake as a build requirement if cmake is not installed or is too low a version
setup_requires = []
try:
    if LegacyVersion(get_cmake_version()) < LegacyVersion("3.4"):
        setup_requires.append('cmake')
except SKBuildError:
    setup_requires.append('cmake')
```

4.3 Enabling parallel build

4.3.1 Ninja

If *Ninja* generator is used, the associated build tool (called `ninja`) will automatically parallelize the build based on the number of available CPUs.

To limit the number of parallel jobs, the build tool option `-j N` can be passed to `ninja`.

For example, to limit the number of parallel jobs to *3*, the following could be done:

```
python setup.py bdist_wheel -- -- -j3
```

For complex projects where more granularity is required, it is also possible to limit the number of simultaneous link jobs, or compile jobs, or both.

Indeed, starting with CMake 3.11, it is possible to configure the project with these options:

- CMAKE_JOB_POOL_COMPILE
- CMAKE_JOB_POOL_LINK
- CMAKE_JOB_POOLS

For example, to have at most *5* compile jobs and *2* link jobs, the following could be done:

```
python setup.py bdist_wheel -- \
  -DCMAKE_JOB_POOL_COMPILE:STRING=compile \
  -DCMAKE_JOB_POOL_LINK:STRING=link \
  '-DCMAKE_JOB_POOLS:STRING=compile=5;link=2'
```

4.3.2 Unix Makefiles

If *Unix Makefiles* generator is used, the associated build tool (called `make`) will **NOT** automatically parallelize the build, the user has to explicitly pass option like `-j N`.

For example, to limit the number of parallel jobs to *3*, the following could be done:

```
python setup.py bdist_wheel -- -- -j3
```

4.3.3 Visual Studio IDE

If *Visual Studio IDE* generator is used, there are two types of parallelism:

- target level parallelism
- object level parallelism

Warning: Since finding the right combination of parallelism can be challenging, whenever possible we recommend to use the *Ninja* generator.

To adjust the object level parallelism, the compiler flag `/MP[processMax]` could be specified. To learn more, read /MP (Build with Multiple Processes).

For example:

```
set CXXFLAGS=/MP4
python setup.py bdist_wheel
```

Starting with Visual Studio 2010, the target level parallelism can be set from command line using `/maxcpucount:N`. This defines the number of simultaneous `MSBuild.exe` processes. To learn more, read Building Multiple Projects in Parallel with MSBuild.

For example:

```
python setup.py bdist_wheel -- -- /maxcpucount:4
```

4.4 Support for isolated build

New in version 0.8.0.

As specified in PEP 518, dependencies required at install time can be specified using a `pyproject.toml` file. Starting with pip 10.0, pip reads the `pyproject.toml` file and installs the associated dependencies in an isolated environment. See the pip build system interface documentation.

An isolated environment will be created when using pip to install packages directly from source or to create an editable installation.

scikit-build supports these use cases as well as the case where the isolated environment support is explicitly disabled using the pip option `--no-build-isolation` available with the *install*, *download* and *wheel* commands.

4.5 Optimized incremental build

To optimize the developer workflow, scikit-build reconfigures the CMake project only when needed. It caches the environment associated with the generator as well as the CMake execution properties.

The CMake properties are saved in a *`CMake spec file`* responsible to store the CMake executable path, the CMake configuration arguments, the CMake version as well as the environment variables `PYTHONNOUSERSITE` and `PYTHONPATH`.

If there are no `CMakeCache.txt` file or if any of the CMake properties changes, scikit-build will explicitly reconfigure the project calling *`skbuild.cmaker.CMaker.configure()`*.

If a file is added to the CMake build system by updating one of the `CMakeLists.txt` file, scikit-build will not explicitly reconfigure the project. Instead, the generated build-system will automatically detect the change and reconfigure the project after *`skbuild.cmaker.CMaker.make()`* is called.

4.6 Cross-compilation

See CMake Toolchains.

4.6.1 Introduction to dockross

Note: *To be documented.* See #227.

4.6.2 Using dockcross-manylinux to generate Linux wheels

Note: *To be documented.* See #227.

4.6.3 Using dockcross-mingwpy to generate Windows wheels

Note: *To be documented.* See #227.

4.7 Examples for scikit-build developers

Note: *To be documented.* See #227.

Provide small, self-contained setup function calls for (at least) two use cases:

- when a *CMakeLists.txt* file already exists
- when a user wants scikit-build to create a *CMakeLists.txt* file based on the user specifying some input files.

CHAPTER 5

C Runtime, Compiler and Build System Generator

scikit-build uses sensible defaults allowing to select the C runtime matching the official CPython recommendations. It also ensures developers remain productive by selecting an alternative environment if recommended one is not available.

The table below lists the different C runtime implementations, compilers and their usual distribution mechanisms for each operating systems.

	Linux	MacOSX	Windows
C runtime	GNU C Library (glibc)	libSystem library	Microsoft C run-time library
Compiler	GNU compiler (gcc)	clang	Microsoft C/C++ Compiler (cl.exe)
Provenance	Package manager	OSX SDK within XCode	• Microsoft Visual Studio • Microsoft Windows SDK

5.1 Build system generator

Since scikit-build simply provides glue between *setuptools* and *CMake*, it needs to choose a CMake generator to configure the build system allowing to build of CPython C extensions.

The table below lists the generator supported by scikit-build:

Operating System	Linux	MacOSX	Windows
CMake Generator	1. *Ninja* 2. *Unix Makefiles*		1. *Ninja* 2. *Visual Studio* 3. *NMake Makefiles* 4. *NMake Makefiles JOM*

When building a project, scikit-build iteratively tries each generator (in the order listed in the table) until it finds a working one.

For more details about CMake generators, see CMake documentation.

5.1.1 Ninja

- Supported platform(s): Linux, MacOSX and Windows
- If ninja executable is in the `PATH`, the associated generator is used to setup the project build system based on `ninja` files.
- In a given python environment, installing the ninja python package with `pip install ninja` will ensure that ninja is in the `PATH`.

Note: Automatic parallelism

An advantage of ninja is that it automatically parallelizes the build based on the number of CPUs. See *Enabling parallel build*.

Note: Ninja on Windows

When *Ninja* generator is used on Windows, scikit-build will make sure the project is configured and built with the appropriate[3] environment (equivalent of calling `vcvarsall.bat x86` or `vcvarsall.bat amd64`).

When Visual Studio >= 2017 is used, ninja is available by default thanks to the Microsoft CMake extension:

```
C:/Program Files (x86)/Microsoft Visual Studio/2017/Professional/Common7/IDE/
↪CommonExtensions/Microsoft/CMake/Ninja/ninja.exe
```

5.1.2 Unix Makefiles

- Supported platform(s): Linux, MacOSX
- scikit-build uses this generator to generate a traditional `Makefile` based build system.

[3] Implementation details: This is made possible by internally using the function `query_vcvarsall` from the `distutils.msvc9compiler` (or `distutils._msvccompiler` when visual studio >= 2015 is used). To ensure, the environment associated with the latest compiler is properly detected, the `distutils` modules are systematically patched using `setuptools.monkey.patch_for_msvc_specialized_compiler()`.

5.1.3 Visual Studio IDE

- Supported platform(s): Windows
- scikit-build uses the generator corresponding to selected version of Visual Studio and generate a `solution file` based build system.

	Architecture	
CPython Version	x86 (32-bit)	x64 (64-bit)
3.5 and above	Visual Studio 15 2017 Visual Studio 14 2015	Visual Studio 15 2017 Win64 Visual Studio 14 2015 Win64
3.3 to 3.4	Visual Studio 10 2010	Visual Studio 10 2010 Win64
2.7 to 3.2	Visual Studio 9 2008	Visual Studio 9 2008 Win64

Note: The Visual Studio generators can not be used when only *alternative environments* are installed, in that case *Ninja* or *NMake Makefiles* are used.

5.1.4 NMake Makefiles

- Supported platform(s): Windows
- scikit-build will make sure the project is configured and built with the appropriate[3] environment (equivalent of calling `vcvarsall.bat x86` or `vcvarsall.bat amd64`).

Note: NMake Makefiles JOM

The *NMake Makefiles JOM* generator is supported **but** it is not automatically used by scikit-build (even if jom executable is in the `PATH`), it always needs to be explicitly specified. For example:

```
python setup.py build -G "NMake Makefiles JOM"
```

For more details, see *scikit-build options*.

5.2 Linux

scikit-build uses the toolchain set using `CC` (and `CXX`) environment variables. If no environment variable is set, it defaults to `gcc`.

To build compliant Linux wheels, scikit-build also supports the `manylinux` platform described in PEP-0513. We recommend the use of dockcross/manylinux-x64 and dockcross/manylinux-x86. These images are optimized for building Linux wheels using scikit-build.

5.3 MacOSX

scikit-build uses the toolchain set using `CC` (and `CXX`) environment variables. If no environment variable is set, it defaults to the Apple compiler installed with XCode.

5.3.1 Default Deployment Target and Architecture

New in version 0.7.0.

The default deployment target and architecture selected by scikit-build are hard-coded for MacOSX and are respectively `10.6` and `x86_64`.

This means that the platform name associated with the *bdist_wheel* command is:

```
macosx-10.6-x86_64
```

and is equivalent to building the wheel using:

```
python setup.py bdist_wheel --plat-name macosx-10.6-x86_64
```

Respectively, the values associated with the corresponding CMAKE_OSX_DEPLOYMENT_TARGET and CMAKE_OSX_ARCHITECTURES CMake options that are automatically used to configure the project are the following:

```
CMAKE_OSX_DEPLOYMENT_TARGET:STRING=10.6
CMAKE_OSX_ARCHITECTURES:STRING=x86_64
```

As illustrated in the table below, choosing `10.6` as deployment target to build MacOSX wheels will allow them to work on *System CPython*, the *Official CPython*, *Macports* and also *Homebrew* installations of CPython.

Table 1: List of platform names for each CPython distributions, CPython and OSX versions.

CPython Distribution	CPython Version	OSX Version	`get_platform()` [1]
Official CPython	3.6, 3.5, 3.4, 2.7	10.12	macosx-**10.6**-intel
	3.4, 2.7	10.9	
	2.7	10.7	
System CPython	2.7	10.12	macosx-10.12-intel
		10.9	macosx-10.9-intel
		10.7	macosx-10.7-intel
Macports CPython	2.7	10.9	macosx-10.9-x86_64
Homebrew CPython	2.7	10.9	

The information above have been adapted from the excellent Spinning wheels article written by Matthew Brett.

5.3.2 Default SDK and customization

New in version 0.7.0.

By default, scikit-build lets CMake discover the most recent SDK available on the system during the configuration of the project. CMake internally uses the logic implemented in the Platform/Darwin-Initialize.cmake CMake module.

5.3.3 Customizing SDK

New in version 0.7.0.

If needed, this can be overridden by explicitly passing the CMake option CMAKE_OSX_SYSROOT. For example:

[1] `from distutils.util import get_platform; print(get_platform())`

```
python setup.py bdist_wheel -- -DCMAKE_OSX_SYSROOT:PATH=/Applications/Xcode.app/
↪Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.12.sdk
```

5.3.4 Customizing Deployment Target and Architecture

New in version 0.11.0.

Deployment target can be customized by setting the *MACOSX_DEPLOYMENT_TARGET* environment variable.

New in version 0.7.0.

Deployment target and architecture can be customized by associating the `--plat-name macosx-<deployment_target>-<arch>` option with the *bdist_wheel* command.

For example:

```
python setup.py bdist_wheel --plat-name macosx-10.9-x86_64
```

scikit-build also sets the value of CMAKE_OSX_DEPLOYMENT_TARGET and CMAKE_OSX_ARCHITECTURES option based on the provided platform name. Based on the example above, the options used to configure the associated CMake project are:

```
-DCMAKE_OSX_DEPLOYMENT_TARGET:STRING=10.9
-DCMAKE_OSX_ARCHITECTURES:STRING=x86_64
```

5.3.5 libstdc++ vs libc++

Before OSX 10.9, the default was `libstdc++`.

With OSX 10.9 and above, the default is `libc++`.

Forcing the use of `libstdc++` on newer version of OSX is still possible using the flag `-stdlib=libstdc++`. That said, doing so will report the following warning:

```
clang: warning: libstdc++ is deprecated; move to libc++
```

- libstdc++:

  This is the GNU Standard C++ Library v3 aiming to implement the ISO 14882 Standard C++ library.

- libc++:

  This is a new implementation of the C++ standard library, targeting C++11.

5.4 Windows

5.4.1 Microsoft C run-time and Visual Studio version

On windows, scikit-build looks for the version of Visual Studio matching the version of CPython being used. The selected Visual Studio version also defines which Microsoft C run-time and compiler are used:

Python version	2.7 to 3.2	3.3 to 3.4	3.5 and above
Microsoft C run-time	msvcr90.dll	msvcr100.dll	ucrtbase.dll
Compiler version	MSVC++ 9.0	MSVC++ 10.0	MSVC++ 14.0
Visual Studio version	2008	2010	2015

5.4.2 Installing compiler and Microsoft C run-time

As outlined above, installing a given version of Visual Studio will automatically install the corresponding compiler along with the Microsoft C run-time libraries.

This means that if you already have the corresponding version of Visual Studio installed, your environment is ready.

Nevertheless, since older version of Visual Studio are not available anymore, this next table references links for installing alternative environments:

Table 2: Download links for Windows SDK and Visual Studio.

CPython version	Download links for Windows SDK or Visual Studio
3.5 and above	• Visual C++ Build Tools 2015 or • Visual Studio 2015
3.3 to 3.4	Windows SDK for Windows 7 and .NET 4.0
2.7 to 3.2	Microsoft Visual C++ Compiler for Python 2.7

These links have been copied from the great article[2] of Steve Dower, engineer at Microsoft.

[2] How to deal with the pain of "unable to find vcvarsall.bat"

CHAPTER 6

CMake modules

To facilitate the writing of `CMakeLists.txt` used to build CPython C/C++/Cython extensions, **scikit-build** provides the following CMake modules:

6.1 Cython

Find `cython` executable.

This module will set the following variables in your project:

`CYTHON_EXECUTABLE` path to the `cython` program

`CYTHON_VERSION` version of `cython`

`CYTHON_FOUND` true if the program was found

For more information on the Cython project, see http://cython.org/.

Cython is a language that makes writing C extensions for the Python language as easy as Python itself.

The following functions are defined:

`add_cython_target`

Create a custom rule to generate the source code for a Python extension module using cython.

add_cython_target(<Name> [<CythonInput>] [EMBED_MAIN] [C | CXX] [PY2 | PY3] [OUTPUT_VAR <OutputVar>])

`<Name>` is the name of the new target, and `<CythonInput>` is the path to a cython source file. Note that, despite the name, no new targets are created by this function. Instead, see `OUTPUT_VAR` for retrieving the path to the generated source for subsequent targets.

If only `<Name>` is provided, and it ends in the ".pyx" extension, then it is assumed to be the `<CythonInput>`. The name of the input without the extension is used as the target name. If only `<Name>` is provided, and it does not end in the ".pyx" extension, then the `<CythonInput>` is assumed to be `<Name>.pyx`.

The Cython include search path is amended with any entries found in the `INCLUDE_DIRECTORIES` property of the directory containing the `<CythonInput>` file. Use `include_directories` to add to the Cython include search path.

Options:

EMBED_MAIN Embed a main() function in the generated output (for stand-alone applications that initialize their own Python runtime).

C | CXX Force the generation of either a C or C++ file. By default, a C file is generated, unless the C language is not enabled for the project; in this case, a C++ file is generated by default.

PY2 | PY3 Force compilation using either Python-2 or Python-3 syntax and code semantics. By default, Python-2 syntax and semantics are used if the major version of Python found is 2. Otherwise, Python-3 syntax and sematics are used.

OUTPUT_VAR <OutputVar> Set the variable `<OutputVar>` in the parent scope to the path to the generated source file. By default, `<Name>` is used as the output variable name.

Defined variables:

<OutputVar> The path of the generated source file.

Cache variables that effect the behavior include:

CYTHON_ANNOTATE whether to create an annotated .html file when compiling

CYTHON_FLAGS additional flags to pass to the Cython compiler

6.1.1 Example usage

```
find_package(Cython)

# Note: In this case, either one of these arguments may be omitted; their
# value would have been inferred from that of the other.
add_cython_target(cy_code cy_code.pyx)

add_library(cy_code MODULE ${cy_code})
target_link_libraries(cy_code ...)
```

6.2 NumPy

Find the include directory for `numpy/arrayobject.h` as well as other NumPy tools like `conv-template` and `from-template`.

This module sets the following variables:

NumPy_FOUND True if NumPy was found.

NumPy_INCLUDE_DIRS The include directories needed to use NumpPy.

NumPy_VERSION The version of NumPy found.

NumPy_CONV_TEMPLATE_EXECUTABLE Path to conv-template executable.

NumPy_FROM_TEMPLATE_EXECUTABLE Path to from-template executable.

The module will also explicitly define one cache variable:

`NumPy_INCLUDE_DIR`

Note: To support NumPy < v0.15.0 where `from-template` and `conv-template` are not declared as entry points, the module emulates the behavior of standalone executables by setting the corresponding variables with the path the the python interpreter and the path to the associated script. For example:

```
set(NumPy_CONV_TEMPLATE_EXECUTABLE /path/to/python /path/to/site-packages/numpy/
↪distutils/conv_template.py CACHE STRING "Command executing conv-template program"␣
↪FORCE)

set(NumPy_FROM_TEMPLATE_EXECUTABLE /path/to/python /path/to/site-packages/numpy/
↪distutils/from_template.py CACHE STRING "Command executing from-template program"␣
↪FORCE)
```

6.3 PythonExtensions

This module defines CMake functions to build Python extension modules and stand-alone executables.

The following variables are defined:

```
PYTHON_PREFIX                     - absolute path to the current Python
                                    distribution's prefix
PYTHON_SITE_PACKAGES_DIR          - absolute path to the current Python
                                    distribution's site-packages directory
PYTHON_RELATIVE_SITE_PACKAGES_DIR - path to the current Python
                                    distribution's site-packages directory
                                    relative to its prefix
PYTHON_SEPARATOR                  - separator string for file path
                                    components.  Equivalent to ``os.sep`` in
                                    Python.
PYTHON_PATH_SEPARATOR             - separator string for PATH-style
                                    environment variables.  Equivalent to
                                    ``os.pathsep`` in Python.
PYTHON_EXTENSION_MODULE_SUFFIX    - suffix of the compiled module. For example, on
                                    Linux, based on environment, it could be ``.
↪cpython-35m-x86_64-linux-gnu.so``.
```

The following functions are defined:

`python_extension_module`

For libraries meant to be used as Python extension modules, either dynamically loaded or directly linked. Amend the configuration of the library target (created using `add_library`) with additional options needed to build and use the referenced library as a Python extension module.

python_extension_module(<Target> [LINKED_MODULES_VAR <LinkedModVar>] [FORWARD_DECL_MODULES_VAR <ForwardDeclModVar>] [MODULE_SUFFIX <ModuleSuffix>])

Only extension modules that are configured to be built as MODULE libraries can be runtime-loaded through the standard Python import mechanism. All other modules can only be included in standalone applications that are written to expect their presence. In addition to being linked against the libraries for these modules, such applications must forward declare their entry points and initialize them prior to use. To generate these forward declarations and initializations, see `python_modules_header`.

If `<Target>` does not refer to a target, then it is assumed to refer to an extension module that is not linked at all, but compiled along with other source files directly into an executable. Adding these modules does not cause any library configuration modifications, and they are not added to the list of linked modules. They still must be forward declared and initialized, however, and so are added to the forward declared modules list.

If the associated target is of type `MODULE_LIBRARY`, the LINK_FLAGS target property is used to set symbol visibility and export only the module init function. This applies to GNU and MSVC compilers.

Options:

`LINKED_MODULES_VAR <LinkedModVar>` Name of the variable referencing a list of extension modules whose libraries must be linked into the executables of any stand-alone applications that use them. By default, the global property `PY_LINKED_MODULES_LIST` is used.

`FORWARD_DECL_MODULES_VAR <ForwardDeclModVar>` Name of the variable referencing a list of extension modules whose entry points must be forward declared and called by any stand-alone applications that use them. By default, the global property `PY_FORWARD_DECL_MODULES_LIST` is used.

`MODULE_SUFFIX <ModuleSuffix>` Suffix appended to the python extension module file. The default suffix is retrieved using `sysconfig.get_config_var("SO")"`, if not available, the default is then `.so` on unix and `.pyd` on windows. Setting the variable `PYTHON_EXTENSION_MODULE_SUFFIX` in the caller scope defines the value used for all extensions not having a suffix explicitly specified using `MODULE_SUFFIX` parameter.

`python_standalone_executable`
python_standalone_executable(<Target>)

For standalone executables that initialize their own Python runtime (such as when building source files that include one generated by Cython with the –embed option). Amend the configuration of the executable target (created using `add_executable`) with additional options needed to properly build the referenced executable.

`python_modules_header`

Generate a header file that contains the forward declarations and initialization routines for the given list of Python extension modules. `<Name>` is the logical name for the header file (no file extensions). `<HeaderFilename>` is the actual destination filename for the header file (e.g.: decl_modules.h).

python_modules_header(<Name> [HeaderFilename] [FORWARD_DECL_MODULES_LIST <ForwardDeclModList>] [HEADER_OUTPUT_VAR <HeaderOutputVar>] [INCLUDE_DIR_OUTPUT_VAR <IncludeDirOutputVar>])

without the extension is used as the logical name. If only `<Name>` is

If only `<Name>` is provided, and it ends in the ".h" extension, then it is assumed to be the `<HeaderFilename>`. The filename of the header file provided, and it does not end in the ".h" extension, then the `<HeaderFilename>` is assumed to `<Name>.h`.

The exact contents of the generated header file depend on the logical `<Name>`. It should be set to a value that corresponds to the target application, or for the case of multiple applications, some identifier that conveyes its purpose. It is featured in the generated multiple inclusion guard as well as the names of the generated initialization routines.

The generated header file includes forward declarations for all listed modules, as well as implementations for the following class of routines:

`int <Name>_<Module>(void)` Initializes the python extension module, `<Module>`. Returns an integer handle to the module.

`void <Name>_LoadAllPythonModules(void)` Initializes all listed python extension modules.

`void CMakeLoadAllPythonModules(void);` Alias for `<Name>_LoadAllPythonModules` whose name does not depend on `<Name>`. This function is excluded during preprocessing if the preprocessing macro `EXCLUDE_LOAD_ALL_FUNCTION` is defined.

`void Py_Initialize_Wrapper();` Wrapper arpund `Py_Initialize()` that initializes all listed python extension modules. This function is excluded during preprocessing if the preprocessing macro

EXCLUDE_PY_INIT_WRAPPER is defined. If this function is generated, then Py_Initialize() is redefined to a macro that calls this function.

Options:

FORWARD_DECL_MODULES_LIST <ForwardDeclModList> List of extension modules for which to generate forward declarations of their entry points and their initializations. By default, the global property PY_FORWARD_DECL_MODULES_LIST is used.

HEADER_OUTPUT_VAR <HeaderOutputVar> Name of the variable to set to the path to the generated header file. By default, <Name> is used.

INCLUDE_DIR_OUTPUT_VAR <IncludeDirOutputVar> Name of the variable to set to the path to the directory containing the generated header file. By default, <Name>_INCLUDE_DIRS is used.

Defined variables:

<HeaderOutputVar> The path to the generated header file

<IncludeDirOutputVar> Directory containing the generated header file

6.3.1 Example usage

```
find_package(PythonInterp)
find_package(PythonLibs)
find_package(PythonExtensions)
find_package(Cython)
find_package(Boost COMPONENTS python)

# Simple Cython Module -- no executables
add_cython_target(_module.pyx)
add_library(_module MODULE ${_module})
python_extension_module(_module)

# Mix of Cython-generated code and C++ code using Boost Python
# Stand-alone executable -- no modules
include_directories(${Boost_INCLUDE_DIRS})
add_cython_target(main.pyx CXX EMBED_MAIN)
add_executable(main boost_python_module.cxx ${main})
target_link_libraries(main ${Boost_LIBRARIES})
python_standalone_executable(main)

# stand-alone executable with three extension modules:
# one statically linked, one dynamically linked, and one loaded at runtime
#
# Freely mixes Cython-generated code, code using Boost-Python, and
# hand-written code using the CPython API.

# module1 -- statically linked
add_cython_target(module1.pyx)
add_library(module1 STATIC ${module1})
python_extension_module(module1
                        LINKED_MODULES_VAR linked_module_list
                        FORWARD_DECL_MODULES_VAR fdecl_module_list)

# module2 -- dynamically linked
include_directories({Boost_INCLUDE_DIRS})
add_library(module2 SHARED boost_module2.cxx)
```

(continues on next page)

(continued from previous page)

```
target_link_libraries(module2 ${Boost_LIBRARIES})
python_extension_module(module2
                        LINKED_MODULES_VAR linked_module_list
                        FORWARD_DECL_MODULES_VAR fdecl_module_list)

# module3 -- loaded at runtime
add_cython_target(module3a.pyx)
add_library(module3 MODULE ${module3a} module3b.cxx)
target_link_libraries(module3 ${Boost_LIBRARIES})
python_extension_module(module3
                        LINKED_MODULES_VAR linked_module_list
                        FORWARD_DECL_MODULES_VAR fdecl_module_list)

# application executable -- generated header file + other source files
python_modules_header(modules
                      FORWARD_DECL_MODULES_LIST ${fdecl_module_list})
include_directories(${modules_INCLUDE_DIRS})

add_cython_target(mainA)
add_cython_target(mainC)
add_executable(main ${mainA} mainB.cxx ${mainC} mainD.c)

target_link_libraries(main ${linked_module_list} ${Boost_LIBRARIES})
python_standalone_executable(main)
```

The following functions are defined:

add_python_library

Add a library that contains a mix of C, C++, Fortran, Cython, F2PY, Template, and Tempita sources. The required targets are automatically generated to "lower" source files from their high-level representation to a file that the compiler can accept.

> **add_python_library(<Name>** SOURCES [source1 [source2 . . .]] [INCLUDE_DIRECTORIES [dir1 [dir2 . . .]] [LINK_LIBRARIES [lib1 [lib2 . . .]] [DEPENDS [source1 [source2 . . .]]])

6.3.2 Example usage

```
find_package(PythonExtensions)

file(GLOB arpack_sources ARPACK/SRC/*.f ARPACK/UTIL/*.f)

 add_python_library(arpack_scipy
   SOURCES ${arpack_sources}
           ${g77_wrapper_sources}
   INCLUDE_DIRECTORIES ARPACK/SRC
 )
```

add_python_extension

Add a extension that contains a mix of C, C++, Fortran, Cython, F2PY, Template, and Tempita sources. The required targets are automatically generated to "lower" source files from their high-level representation to a file that the compiler can accept.

> **add_python_extension(<Name>** SOURCES [source1 [source2 . . .]] [INCLUDE_DIRECTORIES [dir1 [dir2 . . .]] [LINK_LIBRARIES [lib1 [lib2 . . .]] [DEPENDS [source1 [source2 . . .]]])

6.3.3 Example usage

```
find_package(PythonExtensions)

file(GLOB arpack_sources ARPACK/SRC/*.f ARPACK/UTIL/*.f)

 add_python_extension(arpack_scipy
   SOURCES ${arpack_sources}
           ${g77_wrapper_sources}
   INCLUDE_DIRECTORIES ARPACK/SRC
 )
```

6.4 F2PY

The purpose of the F2PY –Fortran to Python interface generator– project is to provide a connection between Python and Fortran languages.

F2PY is a Python package (with a command line tool f2py and a module f2py2e) that facilitates creating/building Python C/API extension modules that make it possible to call Fortran 77/90/95 external subroutines and Fortran 90/95 module subroutines as well as C functions; to access Fortran 77 COMMON blocks and Fortran 90/95 module data, including allocatable arrays from Python.

For more information on the F2PY project, see http://www.f2py.com/.

The following variables are defined:

```
F2PY_EXECUTABLE      - absolute path to the F2PY executable
```

```
F2PY_VERSION_STRING  - the version of F2PY found
F2PY_VERSION_MAJOR   - the F2PY major version
F2PY_VERSION_MINOR   - the F2PY minor version
F2PY_VERSION_PATCH   - the F2PY patch version
```

Note: By default, the module finds the F2PY program associated with the installed NumPy package.

6.4.1 Example usage

Assuming that a package named `method` is declared in `setup.py` and that the corresponding directory containing `__init__.py` also exists, the following CMake code can be added to `method/CMakeLists.txt` to ensure the C sources associated with `cylinder_methods.f90` are generated and the corresponding module is compiled:

```
find_package(F2PY REQUIRED)

set(f2py_module_name "_cylinder_methods")
set(fortran_src_file "${CMAKE_CURRENT_SOURCE_DIR}/cylinder_methods.f90")

set(generated_module_file ${CMAKE_CURRENT_BINARY_DIR}/${f2py_module_name}${PYTHON_
↪EXTENSION_MODULE_SUFFIX})

add_custom_target(${f2py_module_name} ALL
  DEPENDS ${generated_module_file}
```

(continues on next page)

(continued from previous page)

```
  )

add_custom_command(
  OUTPUT ${generated_module_file}
  COMMAND ${F2PY_EXECUTABLE}
    -m ${f2py_module_name}
    -c
    ${fortran_src_file}
  WORKING_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}
  )

install(FILES ${generated_module_file} DESTINATION methods)
```

Warning: Using `f2py` with `-c` argument means that f2py is also responsible to build the module. In that case, CMake is not used to find the compiler and configure the associated build system.

The following functions are defined:

add_f2py_target

Create a custom rule to generate the source code for a Python extension module using f2py.

add_f2py_target(<Name> [<F2PYInput>] [OUTPUT_VAR <OutputVar>])

`<Name>` is the name of the new target, and `<F2PYInput>` is the path to a pyf source file. Note that, despite the name, no new targets are created by this function. Instead, see `OUTPUT_VAR` for retrieving the path to the generated source for subsequent targets.

If only `<Name>` is provided, and it ends in the ".pyf" extension, then it is assumed to be the `<F2PYInput>`. The name of the input without the extension is used as the target name. If only `<Name>` is provided, and it does not end in the ".pyf" extension, then the `<F2PYInput>` is assumed to be `<Name>.pyf`.

Options:

OUTPUT_VAR <OutputVar> Set the variable `<OutputVar>` in the parent scope to the path to the generated source file. By default, `<Name>` is used as the output variable name.

DEPENDS [source [source2...]] Sources that must be generated before the F2PY command is run.

Defined variables:

<OutputVar> The path of the generated source file.

6.4.2 Example usage

```
find_package(F2PY)

# Note: In this case, either one of these arguments may be omitted; their
# value would have been inferred from that of the other.
add_f2py_target(f2py_code f2py_code.pyf)

add_library(f2py_code MODULE ${f2py_code})
target_link_libraries(f2py_code ...)
```

They can be included using `find_package`:

```
find_package(Cython REQUIRED)
find_package(NumPy REQUIRED)
find_package(PythonExtensions REQUIRED)
find_package(F2PY REQUIRED)
```

For more details, see the respective documentation of each modules.

CHAPTER 7

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

7.1 Types of Contributions

You can contribute in many ways:

7.1.1 Report Bugs

Report bugs at https://github.com/scikit-build/scikit-build/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

7.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” is open to whoever wants to implement it.

7.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with “feature” is open to whoever wants to implement it.

7.1.4 Write Documentation

The scikit-build project could always use more documentation. We welcome help with the official scikit-build docs, in docstrings, or even on blog posts and articles for the web.

7.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/scikit-build/scikit-build/issues.

If you are proposing a new feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

7.2 Get Started

Ready to contribute? Here's how to set up *scikit-build* for local development.

1. Fork the *scikit-build* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/scikit-build.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed (*pip install virtualenvwrapper*), this is how you set up your cloned fork for local development:

```
$ mkvirtualenv scikit-build
$ cd scikit-build/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8
$ python setup.py test
$ tox
```

   If needed, you can get flake8 and tox by using *pip install* to install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

7.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in *README.rst*.
3. The pull request should work for Python 2.7, and 3.3, 3.4, 3.5 and PyPy. Check https://travis-ci.org/scikit-build/scikit-build/pull_requests and make sure that the tests pass for all supported Python versions.

7.4 Tips

To run a subset of tests:

```
$ pytest tests/test_skbuild.py
```

CHAPTER 8

Hacking

8.1 Controlling CMake using scikit-build

You can drive CMake directly using scikit-build:

```
""" Use scikit-build's `cmaker` to control CMake configuration and build.

1. Use `cmaker` to define an object that provides convenient access to
   CMake's configure and build functionality.

2. Use defined object, `maker`, to call `configure()` to read the
   `CMakeLists.txt` file in the current directory and generate a Makefile,
   Visual Studio solution, or whatever is appropriate for your platform.

3. Call `make()` on the object to execute the build with the
   appropriate build tool and perform installation to the local directory.
"""
from skbuild import cmaker
maker = cmaker.CMaker()

maker.configure()

maker.make()
```

See `skbuild.cmaker.CMaker` for more details.

8.2 Internal API

8.2.1 skbuild

skbuild package

scikit-build is an improved build system generator for CPython C extensions.

This module provides the *glue* between the setuptools Python module and CMake.

`skbuild.setup(*args, **kw)`
This function wraps setup() so that we can run cmake, make, CMake build, then proceed as usual with setuptools, appending the CMake-generated output as necessary.

The CMake project is re-configured only if needed. This is achieved by (1) retrieving the environment mapping associated with the generator set in the `CMakeCache.txt` file, (2) saving the CMake configure arguments and version in `skbuild.constants.CMAKE_SPEC_FILE()`: and (3) re-configuring only if either the generator or the CMake specs change.

Subpackages

skbuild.command package

Collection of objects allowing to customize behavior of standard distutils and setuptools commands.

class `skbuild.command.set_build_base_mixin`
Bases: `object`

Mixin allowing to override distutils and setuptools commands.

finalize_options(**args, **kwargs*)
Override built-in function and set a new *build_base*.

Submodules

skbuild.command.bdist module

This module defines custom implementation of `bdist` setuptools command.

class `skbuild.command.bdist.bdist(dist, **kw)`
Bases: `skbuild.command.set_build_base_mixin`, `skbuild.utils.NewStyleClass`

Custom implementation of `bdist` setuptools command.

skbuild.command.bdist_wheel module

This module defines custom implementation of `bdist_wheel` setuptools command.

class `skbuild.command.bdist_wheel.bdist_wheel(dist, **kw)`
Bases: `skbuild.command.set_build_base_mixin`, `skbuild.utils.NewStyleClass`

Custom implementation of `bdist_wheel` setuptools command.

finalize_options(**args, **kwargs*)
Ensure MacOSX wheels include the expected platform information.

run(**args, **kwargs*)
Handle –hide-listing option.

write_wheelfile(*wheelfile_base, _=None*)
Write `skbuild <version>` as a wheel generator. See PEP-0427 for more details.

skbuild.command.build module

This module defines custom implementation of `build` setuptools command.

class `skbuild.command.build.`**build**(*dist*, ***kw*)

Bases: *skbuild.command.set_build_base_mixin*, *skbuild.utils.NewStyleClass*

Custom implementation of `build` setuptools command.

skbuild.command.build_py module

This module defines custom implementation of `build_py` setuptools command.

class `skbuild.command.build_py.`**build_py**(*dist*, ***kw*)

Bases: *skbuild.command.set_build_base_mixin*, *skbuild.utils.NewStyleClass*

Custom implementation of `build_py` setuptools command.

build_module(*module*, *module_file*, *package*)

Handle –hide-listing option.

Increments `outfiles_count`.

find_modules()

Finds individually-specified Python modules, ie. those listed by module name in 'self.py_modules'. Returns a list of tuples (package, module_base, filename): 'package' is a tuple of the path through package-space to the module; 'module_base' is the bare (no packages, no dots) module name, and 'filename' is the path to the ".py" file (relative to the distribution root) that implements the module.

initialize_options()

Handle –hide-listing option.

Initializes `outfiles_count`.

run(**args*, ***kwargs*)

Handle –hide-listing option.

Display number of copied files. It corresponds to the value of `outfiles_count`.

skbuild.command.clean module

This module defines custom implementation of `clean` setuptools command.

class `skbuild.command.clean.`**clean**(*dist*, ***kw*)

Bases: *skbuild.command.set_build_base_mixin*, *skbuild.utils.NewStyleClass*

Custom implementation of `clean` setuptools command.

run()

After calling the super class implementation, this function removes the directories specific to scikit-build.

skbuild.command.egg_info module

This module defines custom implementation of `egg_info` setuptools command.

class `skbuild.command.egg_info.`**egg_info**(*dist*, ***kw*)

Bases: *skbuild.command.set_build_base_mixin*, *skbuild.utils.NewStyleClass*

Custom implementation of `egg_info` setuptools command.

finalize_options(**args*, ***kwargs*)
Override built-in function and set a new *build_base*.

skbuild.command.generate_source_manifest module

This module defines custom `generate_source_manifest` setuptools command.

class `skbuild.command.generate_source_manifest.`**generate_source_manifest**(*dist*)
Bases: *skbuild.command.set_build_base_mixin*, `skbuild.utils.NewStyleClass`

Custom setuptools command generating a *MANIFEST* file if not already provided.

description = 'generate source MANIFEST'

finalize_options(**args*, ***kwargs*)
Set final values for all the options that this command supports.

initialize_options()
Set default values for all the options that this command supports.

run()
If neither a *MANIFEST*, nor a *MANIFEST.in* file is provided, and we are in a git repo, try to create a *MANIFEST* file from the output of *git ls-tree –name-only -r HEAD*.

We need a reliable way to tell if an existing *MANIFEST* file is one we've generated. distutils already uses a first-line comment to tell if the *MANIFEST* file was generated from *MANIFEST.in*, so we use a dummy file, *_skbuild_MANIFEST*, to avoid confusing distutils.

skbuild.command.install module

This module defines custom implementation of `install` setuptools command.

class `skbuild.command.install.`**install**(*dist*, ***kw*)
Bases: *skbuild.command.set_build_base_mixin*, *skbuild.utils.NewStyleClass*

Custom implementation of `install` setuptools command.

finalize_options(**args*, ***kwargs*)
Ensure that if the distribution is non-pure, all modules are installed in `self.install_platlib`.

Note: *setuptools.dist.Distribution.has_ext_modules()* is overridden in `setuptools_wrap.setup()`.

skbuild.command.install_lib module

This module defines custom implementation of `install_lib` setuptools command.

class `skbuild.command.install_lib.`**install_lib**(*dist*, ***kw*)
Bases: *skbuild.command.set_build_base_mixin*, *skbuild.utils.NewStyleClass*

Custom implementation of `install_lib` setuptools command.

install()
Handle –hide-listing option.

skbuild.command.install_scripts module

This module defines custom implementation of `install_scripts` setuptools command.

class `skbuild.command.install_scripts.`**`install_scripts`**(*dist*, ***kw*)
Bases: *skbuild.command.set_build_base_mixin*, *skbuild.utils.NewStyleClass*

Custom implementation of `install_scripts` setuptools command.

run(**args*, ***kwargs*)
Handle –hide-listing option.

skbuild.command.sdist module

This module defines custom implementation of `sdist` setuptools command.

class `skbuild.command.sdist.`**`sdist`**(*dist*, ***kw*)
Bases: *skbuild.command.set_build_base_mixin*, *skbuild.utils.NewStyleClass*

Custom implementation of `sdist` setuptools command.

make_archive(*base_name*, *_format*, *root_dir=None*, *base_dir=None*, *owner=None*, *group=None*)
Handle –hide-listing option.

make_distribution()
This function was originally re-implemented in setuptools to workaround https://github.com/pypa/setuptools/issues/516 and later ported to scikit-build to ensure symlinks are maintained.

make_release_tree(*base_dir*, *files*)
Handle –hide-listing option.

run(**args*, ***kwargs*)
Force *egg_info.egg_info* command to run.

skbuild.platform_specifics package

This package provides *get_platform()* allowing to get an instance of *abstract.CMakePlatform* matching the current platform.

class `skbuild.platform_specifics.`**`CMakeGenerator`**(*name*, *env=None*, *toolset=None*)
Bases: `object`

Represents a CMake generator.

__init__(*name*, *env=None*, *toolset=None*)
Instantiate a generator object with the given `name`.

By default, `os.environ` is associated with the generator. Dictionary passed as `env` parameter will be merged with `os.environ`. If an environment variable is set in both `os.environ` and `env`, the variable in `env` is used.

Some CMake generators support a `toolset` specification to tell the native build system how to choose a compiler.

description
Name of CMake generator with properties describing the environment (e.g toolset)

name
Name of CMake generator.

toolset
Toolset specification associated with the CMake generator.

skbuild.platform_specifics.**get_platform**()
Return an instance of *abstract.CMakePlatform* corresponding to the current platform.

Submodules

skbuild.platform_specifics.abstract module

This module defines objects useful to discover which CMake generator is supported on the current platform.

class skbuild.platform_specifics.abstract.**CMakeGenerator**(*name*, *env=None*, *toolset=None*)
Bases: object

Represents a CMake generator.

__init__(*name*, *env=None*, *toolset=None*)
Instantiate a generator object with the given name.

By default, os.environ is associated with the generator. Dictionary passed as env parameter will be merged with os.environ. If an environment variable is set in both os.environ and env, the variable in env is used.

Some CMake generators support a toolset specification to tell the native build system how to choose a compiler.

description
Name of CMake generator with properties describing the environment (e.g toolset)

name
Name of CMake generator.

toolset
Toolset specification associated with the CMake generator.

class skbuild.platform_specifics.abstract.**CMakePlatform**
Bases: object

This class encapsulates the logic allowing to get the identifier of a working CMake generator.

Derived class should at least set *default_generators*.

static cleanup_test()
Delete test project directory.

static compile_test_cmakelist(**args*, ***kwds*)
Attempt to configure the test project with each *CMakeGenerator* from candidate_generators.

Only cmake arguments starting with -DCMAKE_ are used to configure the test project.

The function returns the first generator allowing to successfully configure the test project using cmake_exe_path.

default_generators
List of generators considered by *get_best_generator()*.

generator_installation_help
Return message guiding the user for installing a valid toolchain.

get_best_generator (*generator_name=None, skip_generator_test=False, languages=('CXX', 'C'), cleanup=True, cmake_executable='cmake', cmake_args=()*)

Loop over generators to find one that works by configuring and compiling a test project.

Parameters

- **generator_name** (*string or None*) – If provided, uses only provided generator, instead of trying *default_generators*.
- **skip_generator_test** (*bool*) – If set to True and if a generator name is specified, the generator test is skipped. If no generator_name is specified and the option is set to True, the first available generator is used.
- **languages** (*tuple*) – The languages you'll need for your project, in terms that CMake recognizes.
- **cleanup** (*bool*) – If True, cleans up temporary folder used to test generators. Set to False for debugging to see CMake's output files.
- **cmake_executable** (*string*) – Path to CMake executable used to configure and build the test project used to evaluate if a generator is working.
- **cmake_args** (*tuple*) – List of CMake arguments to use when configuring the test project. Only arguments starting with `-DCMAKE_` are used.

Returns CMake Generator object

Return type *CMakeGenerator* or None

Raises ***skbuild.exceptions.SKBuildGeneratorNotFoundError*** –

get_generator (*generator_name*)

Loop over generators and return the first that matches the given name.

static write_test_cmakelist (*languages*)

Write a minimal `CMakeLists.txt` useful to check if the requested `languages` are supported.

skbuild.platform_specifics.bsd module

This module defines object specific to BSD platform.

class skbuild.platform_specifics.bsd.**BSDPlatform**

Bases: *skbuild.platform_specifics.unix.UnixPlatform*

BSD implementation of *abstract.CMakePlatform*.

skbuild.platform_specifics.linux module

This module defines object specific to Linux platform.

class skbuild.platform_specifics.linux.**LinuxPlatform**

Bases: *skbuild.platform_specifics.unix.UnixPlatform*

Linux implementation of *abstract.CMakePlatform*

static build_essential_install_cmd ()

Return a tuple of the form `(distribution_name, cmd)`.

`cmd` is the command allowing to install the build tools in the current Linux distribution. It set to an empty string if the command is not known.

`distribution_name` is the name of the current distribution. It is set to an empty string if the distribution could not be determined.

generator_installation_help
Return message guiding the user for installing a valid toolchain.

skbuild.platform_specifics.osx module

This module defines object specific to OSX platform.

class skbuild.platform_specifics.osx.**OSXPlatform**
Bases: *skbuild.platform_specifics.unix.UnixPlatform*

OSX implementation of *abstract.CMakePlatform*.

generator_installation_help
Return message guiding the user for installing a valid toolchain.

skbuild.platform_specifics.platform_factory module

This modules implements the logic allowing to instantiate the expected *abstract.CMakePlatform*.

skbuild.platform_specifics.platform_factory.**get_platform**()
Return an instance of *abstract.CMakePlatform* corresponding to the current platform.

skbuild.platform_specifics.unix module

This module defines object specific to Unix platform.

class skbuild.platform_specifics.unix.**UnixPlatform**
Bases: *skbuild.platform_specifics.abstract.CMakePlatform*

Unix implementation of *abstract.CMakePlatform*.

skbuild.platform_specifics.windows module

This module defines object specific to Windows platform.

class skbuild.platform_specifics.windows.**CMakeVisualStudioCommandLineGenerator**(*name*, *year*, *toolset=None*)
Bases: *skbuild.platform_specifics.abstract.CMakeGenerator*

Represents a command-line CMake generator initialized with a specific *Visual Studio* environment.

__init__(*name*, *year*, *toolset=None*)
Instantiate CMake command-line generator.

The generator `name` can be values like *Ninja*, *NMake Makefiles* or *NMake Makefiles JOM*.

The `year` defines the *Visual Studio* environment associated with the generator. See *VS_YEAR_TO_VERSION*.

If set, the `toolset` defines the *Visual Studio Toolset* to select.

The platform (32-bit or 64-bit) is automatically selected based on the value of `platform.architecture()[0]`.

class skbuild.platform_specifics.windows.**CMakeVisualStudioIDEGenerator**(*year*, *toolset=None*)

Bases: *skbuild.platform_specifics.abstract.CMakeGenerator*

Represents a Visual Studio CMake generator.

__init__(*year*, *toolset=None*)

Instantiate a generator object with its name set to the *Visual Studio* generator associated with the given `year` (see *VS_YEAR_TO_VERSION*), the current platform (32-bit or 64-bit) and the selected `toolset` (if applicable).

skbuild.platform_specifics.windows.**VS_YEAR_TO_VERSION = {'2008': 9, '2010': 10, '2012':**

Describes the version of *Visual Studio* supported by *CMakeVisualStudioIDEGenerator* and *CMakeVisualStudioCommandLineGenerator*.

The different version are identified by their year.

class skbuild.platform_specifics.windows.**WindowsPlatform**

Bases: *skbuild.platform_specifics.abstract.CMakePlatform*

Windows implementation of *abstract.CMakePlatform*.

generator_installation_help

Return message guiding the user for installing a valid toolchain.

skbuild.platform_specifics.windows.**find_visual_studio**(*vs_version*)

Return Visual Studio installation path associated with `vs_version` or an empty string if any.

The `vs_version` corresponds to the *Visual Studio* version to lookup. See *VS_YEAR_TO_VERSION*.

Note:

- For VS 2017 and newer, returns *path* based on the result of invoking `vswhere.exe`.
- For VS 2010 to VS 2015, returns *path* by looking up all key/value pairs associated with the `Software\Microsoft\VisualStudio\SxS\VC7` registry key. Each key/value pair is the visual studio version (e.g *14.0*) and the installation path (e.g *C:/Program Files (x86)/Microsoft Visual Studio 14.0/VC/*).

---

skbuild.utils package

This module defines functions generally useful in scikit-build.

class skbuild.utils.**ContextDecorator**(***kwargs*)

Bases: `object`

A base class or mixin that enables context managers to work as decorators.

class skbuild.utils.**PythonModuleFinder**(*packages*, *package_dir*, *py_modules*, *alternative_build_base=None*)

Bases: `skbuild.utils.NewStyleClass`

Convenience class to search for python modules.

This class is based on `distutils.command.build_py.build_by` and provides a specialized version of `find_all_modules()`.

check_module(*module*, *module_file*)

Return True if `module_file` belongs to `module`.

find_all_modules (*project_dir=None*)
Compute the list of all modules that would be built by project located in current directory, whether they are specified one-module-at-a-time `py_modules` or by whole packages `packages`.

By default, the function will search for modules in the current directory. Specifying `project_dir` parameter allow to change this.

Return a list of tuples `(package, module, module_file)`.

find_package_modules (*package, package_dir*)
Temporally prepend the `alternative_build_base` to `module_file`. Doing so will ensure modules can also be found in other location (e.g `skbuild.constants.CMAKE_INSTALL_DIR`).

`skbuild.utils.`**distribution_hide_listing** (**args, **kwds*)
Given a `distribution`, this context manager temporarily sets distutils threshold to WARN if `--hide-listing` argument was provided.

It yields True if `--hide-listing` argument was provided.

`skbuild.utils.`**mkdir_p** (*path*)
Ensure directory `path` exists. If needed, parent directories are created.

Adapted from http://stackoverflow.com/a/600612/1539918

`skbuild.utils.`**new_style** (*klass*)
distutils/setuptools command classes are old-style classes, which won't work with mixins.

To work around this limitation, we dynamically convert them to new style classes by creating a new class that inherits from them and also <object>. This ensures that <object> is always at the end of the MRO, even after being mixed in with other classes.

`skbuild.utils.`**parse_manifestin** (*template*)
This function parses template file (usually MANIFEST.in)

class `skbuild.utils.`**push_dir** (*directory=None, make_directory=False*)
Bases: *skbuild.utils.ContextDecorator*

Context manager to change current directory.

`skbuild.utils.`**to_platform_path** (*path*)
Return a version of `path` where all separator are `os.sep`

`skbuild.utils.`**to_unix_path** (*path*)
Return a version of `path` where all separator are `/`

Submodules

skbuild.cmaker module

This module provides an interface for invoking CMake executable.

class `skbuild.cmaker.`**CMaker** (*cmake_executable='cmake'*)
Bases: `object`

Interface to CMake executable.

static check_for_bad_installs ()
This function tries to catch files that are meant to be installed outside the project root before they are actually installed.

Indeed, we can not wait for the manifest, so we try to extract the information (install destination) from the CMake build files `*.cmake` found in *skbuild.constants.CMAKE_BUILD_DIR()*.

It raises `skbuild.exceptions.SKBuildError` if it found install destination outside of `skbuild.constants.CMAKE_INSTALL_DIR()`.

configure (*clargs=()*, *generator_name=None*, *skip_generator_test=False*, *cmake_source_dir='.'*, *cmake_install_dir="*, *languages=('C', 'CXX')*, *cleanup=True*)
Calls cmake to generate the Makefile/VS Solution/XCode project.

clargs: tuple List of command line arguments to pass to cmake executable.

generator_name: string The string representing the CMake generator to use. If None, uses defaults for your platform.

skip_generator_test: bool If set to True and if a generator name is specified (either as a keyword argument or as *clargs* using *-G <generator_name>*), the generator test is skipped.

cmake_source_dir: string Path to source tree containing a `CMakeLists.txt`

cmake_install_dir: string Relative directory to append to `skbuild.constants.CMAKE_INSTALL_DIR()`.

languages: tuple List of languages required to configure the project and expected to be supported by the compiler. The language identifier that can be specified in the list corresponds to the one recognized by CMake.

cleanup: bool If True, cleans up temporary folder used to test generators. Set to False for debugging to see CMake's output files.

Return a mapping of the environment associated with the selected `skbuild.platform_specifics.abstract.CMakeGenerator`.

Mapping of the environment can also be later retrieved using `get_cached_generator_env()`.

get_cached_generator_env ()
If any, return a mapping of environment associated with the cached generator.

get_cached_generator_name ()
Reads and returns the cached generator from the `skbuild.constants.CMAKE_BUILD_DIR()`:. Returns None if not found.

static get_python_include_dir (*python_version*)
Get include directory associated with the current python interpreter.

static get_python_library (*python_version*)
Get path to the python library associated with the current python interpreter.

static get_python_version ()
Get version associated with the current python interpreter.

install ()
Returns a list of file paths to install via setuptools that is compatible with the data_files keyword argument.

make (*clargs=()*, *config='Release'*, *source_dir='.'*, *env=None*)
Calls the system-specific make program to compile code.

skbuild.cmaker.**get_cmake_version** (*cmake_executable='cmake'*)
Runs CMake and extracts associated version information. Raises `skbuild.exceptions.SKBuildError` if it failed to execute CMake.

skbuild.cmaker.**has_cmake_cache_arg** (*cmake_args*, *arg_name*, *arg_value=None*)
Return True if `-D<arg_name>:TYPE=<arg_value>` is found in `cmake_args`. If `arg_value` is None, return True only if `-D<arg_name>:` is found in the list.

`skbuild.cmaker.`**`pop_arg`**`(`*arg*, *args*, *default=None*`)`
Pops an argument `arg` from an argument list `args` and returns the new list and the value of the argument if present and a default otherwise.

skbuild.compat module

`skbuild.compat.`**`which`**`(`*name*, *flags=1*`)`
Analogue of unix 'which'. Borrowed from the Twisted project, see their licence here: https://twistedmatrix.com/trac/browser/trunk/LICENSE

Copied from `pytest_shutil.cmdline.which` to allow testing on conda-forge where `pytest-shutil` is not available.

skbuild.constants module

This module defines constants commonly used in scikit-build.

`skbuild.constants.`**`CMAKE_BUILD_DIR`**`()`
CMake build directory.

`skbuild.constants.`**`CMAKE_DEFAULT_EXECUTABLE = 'cmake'`**
Default path to CMake executable.

`skbuild.constants.`**`CMAKE_INSTALL_DIR`**`()`
CMake install directory.

`skbuild.constants.`**`CMAKE_SPEC_FILE`**`()`
CMake specification file storing CMake version, CMake configuration arguments and environment variables `PYTHONNOUSERSITE` and `PYTHONPATH`.

`skbuild.constants.`**`SETUPTOOLS_INSTALL_DIR`**`()`
Setuptools install directory.

`skbuild.constants.`**`SKBUILD_DIR`**`()`
Top-level directory where setuptools and CMake directories are generated.

`skbuild.constants.`**`SKBUILD_MARKER_FILE`**`()`
Marker file used by *`skbuild.command.generate_source_manifest.generate_source_manifest.run()`*.

`skbuild.constants.`**`set_skbuild_plat_name`**`(`*plat_name*`)`
Set platform name associated with scikit-build functions returning a path:

- *`SKBUILD_DIR()`*
- *`SKBUILD_MARKER_FILE()`*
- *`CMAKE_BUILD_DIR()`*
- *`CMAKE_INSTALL_DIR()`*
- *`CMAKE_SPEC_FILE()`*
- *`SETUPTOOLS_INSTALL_DIR()`*

`skbuild.constants.`**`skbuild_plat_name`**`()`
Get platform name formatted as *<operating_system>[-<operating_system_version>]-<machine_architecture>*.

Default value corresponds to `_default_skbuild_plat_name()` and can be overridden with *set_skbuild_plat_name()*.

Examples of values are *macosx-10.6-x86_64*, *linux-x86_64*, *linux-i686* or *win-am64*.

skbuild.exceptions module

This module defines exceptions commonly used in scikit-build.

exception skbuild.exceptions.**SKBuildError**
Bases: `exceptions.RuntimeError`

Exception raised when an error occurs while configuring or building a project.

exception skbuild.exceptions.**SKBuildGeneratorNotFoundError**
Bases: *skbuild.exceptions.SKBuildError*

Exception raised when no suitable generator is found for the current platform.

skbuild.setuptools_wrap module

This module provides functionality for wrapping key infrastructure components from distutils and setuptools.

skbuild.setuptools_wrap.**create_skbuild_argparser**()
Create and return a scikit-build argument parser.

skbuild.setuptools_wrap.**parse_args**()
This function parses the command-line arguments `sys.argv` and returns the tuple `(setuptools_args, cmake_executable, skip_generator_test, cmake_args, build_tool_args)` where each `*_args` element corresponds to a set of arguments separated by `--`.

skbuild.setuptools_wrap.**parse_skbuild_args**(*args*, *cmake_args*, *build_tool_args*)
Parse arguments in the scikit-build argument set. Convert specified arguments to proper format and append to cmake_args and build_tool_args. Returns the tuple `(remaining arguments, cmake executable, skip_generator_test)`.

skbuild.setuptools_wrap.**setup**(**args*, ***kw*)
This function wraps setup() so that we can run cmake, make, CMake build, then proceed as usual with setuptools, appending the CMake-generated output as necessary.

The CMake project is re-configured only if needed. This is achieved by (1) retrieving the environment mapping associated with the generator set in the `CMakeCache.txt` file, (2) saving the CMake configure arguments and version in *skbuild.constants.CMAKE_SPEC_FILE()*: and (3) re-configuring only if either the generator or the CMake specs change.

skbuild.setuptools_wrap.**strip_package**(*package_parts*, *module_file*)
Given `package_parts` (e.g. `['foo', 'bar']`) and a `module_file` (e.g. `foo/bar/jaz/rock/roll.py`), starting from the left, this function will strip the parts of the path matching the package parts and return a new string (e.g `jaz/rock/roll.py`).

The function will work as expected for either Windows or Unix-style `module_file` and this independently of the platform.

8.3 Internal CMake Modules

8.3.1 targetLinkLibrariesWithDynamicLookup

Public Functions

The following functions are defined:

target_link_libraries_with_dynamic_lookup

```
target_link_libraries_with_dynamic_lookup(<Target> [<Libraries>])
```

Useful to "weakly" link a loadable module. For example, it should be used when compiling a loadable module when the symbols should be resolve from the run-time environment where the module is loaded, and not a specific system library.

Like proper linking, except that the given `<Libraries>` are not necessarily linked. Instead, the `<Target>` is produced in a manner that allows for symbols unresolved within it to be resolved at runtime, presumably by the given `<Libraries>`. If such a target can be produced, the provided `<Libraries>` are not actually linked.

It links a library to a target such that the symbols are resolved at run-time not link-time.

The linker is checked to see if it supports undefined symbols when linking a shared library. If it does then the library is not linked when specified with this function.

On platforms that do not support weak-linking, this function works just like `target_link_libraries`.

Note: For OSX it uses `undefined dynamic_lookup`. This is similar to using `-shared` on Linux where undefined symbols are ignored.

For more details, see blog from Tim D. Smith.

---

check_dynamic_lookup

Check if the linker requires a command line flag to allow leaving symbols unresolved when producing a target of type `<TargetType>` that is weakly-linked against a dependency of type `<LibType>`.

<TargetType> can be one of "STATIC", "SHARED", "MODULE", or "EXE".

<LibType> can be one of "STATIC", "SHARED", or "MODULE".

Long signature:

```
check_dynamic_lookup(<TargetType>
                     <LibType>
                     <ResultVar>
                     [<LinkFlagsVar>])
```

Short signature:

```
check_dynamic_lookup(<ResultVar>) # <TargetType> set to "MODULE"
                                  # <LibType> set to "SHARED"
```

The result is cached between invocations and recomputed only when the value of CMake's linker flag list changes; `CMAKE_STATIC_LINKER_FLAGS` if `<TargetType>` is "STATIC", and `CMAKE_SHARED_LINKER_FLAGS` otherwise.

Defined variables:

<ResultVar> Whether the current C toolchain supports weak-linking for target binaries of type `<TargetType>` that are weakly-linked against a dependency target of type `<LibType>`.

<LinkFlagsVar> List of flags to add to the linker command to produce a working target binary of type `<TargetType>` that is weakly-linked against a dependency target of type `<LibType>`.

HAS_DYNAMIC_LOOKUP_<TargetType>_<LibType> Cached, global alias for `<ResultVar>`

DYNAMIC_LOOKUP_FLAGS_<TargetType>_<LibType> Cached, global alias for `<LinkFlagsVar>`

Private Functions

The following private functions are defined:

> **Warning:** These functions are not part of the scikit-build API. They exist purely as an implementation detail and may change from version to version without notice, or even be removed.
>
> We mean it.

_get_target_type

```
_get_target_type(<ResultVar> <Target>)
```

Shorthand for querying an abbreviated version of the target type of the given `<Target>`.

`<ResultVar>` is set to:

- "STATIC" for a STATIC_LIBRARY,
- "SHARED" for a SHARED_LIBRARY,
- "MODULE" for a MODULE_LIBRARY,
- and "EXE" for an EXECUTABLE.

Defined variables:

<ResultVar> The abbreviated version of the `<Target>`'s type.

_test_weak_link_project

```
_test_weak_link_project(<TargetType>
                        <LibType>
                        <ResultVar>
                        <LinkFlagsVar>)
```

Attempt to compile and run a test project where a target of type `<TargetType>` is weakly-linked against a dependency of type `<LibType>`:

- `<TargetType>` can be one of "STATIC", "SHARED", "MODULE", or "EXE".
- `<LibType>` can be one of "STATIC", "SHARED", or "MODULE".

Defined variables:

<ResultVar> Whether the current C toolchain can produce a working target binary of type `<TargetType>` that is weakly-linked against a dependency target of type `<LibType>`.

<LinkFlagsVar> List of flags to add to the linker command to produce a working target binary of type `<TargetType>` that is weakly-linked against a dependency target of type `<LibType>`.

CHAPTER 9

Credits

Please see the GitHub project page at https://github.com/scikit-build/scikit-build/graphs/contributors

CHAPTER 10

History

PyCMake was created at SciPy 2014 in response to general difficulties building C++ and Fortran based Python extensions across platforms. It was renamed to "scikit-build" in 2016.

CHAPTER 11

Release Notes

This is the list of changes to scikit-build between each release. For full details, see the commit logs at http://github.com/scikit-build/scikit-build

11.1 Next Release

11.1.1 New Features

- Add support for passing *CMake configure options* like *-DFOO:STRING:bar* as global *setuptools* or *pip* options.
- Add support for building project using PyPy or PyPy3. See https://pypy.org See #407.
- Add support for OS/400 (now known as IBM i). Thanks @jwoehr for the contribution. See #444.
- Display CMake command used to configure the project. Thanks @native-api for the contribution. See #443.
- CMake modules:
  - Improve CMake module *F2PY* adding *add_f2py_target()* CMake function allowing to generate **-f2pywrappers.f* and **module.c* files from **.pyf* files. Thanks @xoviat for the contribution.
  - Update CMake module *PythonExtensions* adding *add_python_library()* and *add_python_extension()*. Thanks @xoviat for the contribution.

11.1.2 Bug fixes

- Fix unclosed file resource in `skbuild.cmaker.CMaker.check_for_bad_installs()`. Thanks @Nic30 for the suggestion. See #429.
- Update CMake module *PythonExtensions*:
  - Ensure correct suffix is used for compiled python module on windows. See #383.
  - Fix warning using *EXT_SUFFIX* config variable instead of deprecated *SO* variable. See #381.

- Honor the *MACOSX_DEPLOYMENT_TARGET* environment variable if it is defined on macOS. Thanks @certik for the contribution. See #441.
- Fix CMake module *F2PY* to ensure the *f2py* executable specific to the python version being used is found. See #449. Thanks @bnavigator for the contribution.

11.1.3 Documentation

- Add *Notes* section to the *For maintainers* top-level category that includes a comparison between *sysconfig* and *distutils.sysconfig* modules.
- Remove obsolete comment in *cmaker.py*. See #439. Thanks @isuruf

11.1.4 Tests

- Continuous Integration
  - Update Azure Pipelines configuration for running tests using PyPy3 on Linux. Thanks @mattip for the contribution. See #418.

11.2 Scikit-build 0.10.0

11.2.1 New Features

- Improve message displayed when discovering a working environment for building projects. For example, instead of displaying `-- Trying "Ninja" generator`, it now displays a message like `-- Trying "Ninja (Visual Studio 15 2017 Win64 v140)" generator`.

11.2.2 Bug fixes

- Checking generator candidates can now handle handle paths and binaries with spaces, so that `setup.py --cmake-executable "C:/Program Files (x86)/cmake/cmake.exe"` works as expected. Contributed by @jokva. See #400.
- Fix sdist command to ensure symlinks in original source tree are maintained. Contributed by @anibali. See #401.
- Ensure use of *bdist_egg* or *bdist_rpm* commands trigger build using cmake.
- Fix default value returned by `skbuild.constants.skbuild_plat_name()` on macOS. See #417.

11.2.3 Internal API

- Add `skbuild.platforms.windows.find_visual_studio()`.

11.2.4 Documentation

- Fix typo in example associated with *PythonExtensions*. Thanks @eirrgang for the contribution.
- Update *Making a release* section to include *Conda: Step-by-step* release guide.

11.2.5 Tests

- Introduce `check_sdist_content()` and fix tests that are checking content of sdist to account for changes introduced in Python 3.8 and backported to python 2.7, 3.6 and 3.7. The changes introduced in python/cpython#9419 adds directory entries to ZIP files created by distutils. Thanks @anibali for the contribution. See #404.
- Fix `check_wheel_content()` to consider changes in `0.33.1 < wheel.__version__ < 0.33.4` where directory entries are included when building wheel. See pypa/wheel#294 <https://github.com/pypa/wheel/issues/294>.
- Fix reporting of `AssertionError` raised in `check_wheel_content()` function by relocating the source code into a dedicated module `tests.pytest_helpers` and by adding a `conftest.py` configuration file registering it for pytest assertion rewriting. See https://docs.pytest.org/en/latest/writing_plugins.html#assertion-rewriting and #403.
- Fix `test_generator_selection` when building with "Visual C++ for Python 2.7" installed for all users. This addresses failure associated with `win_c_compilervs2008cxx_compilervs2008python2.7` when running test in scikit-build-feedstock where "Visual C++ for Python 2.7" is installed using (vcpython27 chocolatey package.
- Continuous Integration
  - Add support for Azure Pipelines for Python 3.7 32-bit and 64-bit
  - AppVeyor: Disable test for Python 3.7 32-bit and 64-bit.
  - CircleCI: Update version of docker images from jessie to stretch. This addresses issue circleci/circleci-images#370.
  - TravisCI: Remove obsolete Python 3.4 testing. It reached end-of-life on March 18 2019.

11.3 Scikit-build 0.9.0

11.3.1 New Features

- Add support for building distutils based extensions associated with `ext_modules` setup keyword along side skbuild based extensions. This means using `build_ext` command (and associated `--inplace` argument) is supported. Thanks @Erotemic for the contribution. See #284.

11.3.2 Bug fixes

- Fix build of wheels if path includes spaces. See issue #375. Thanks @padraic-padraic for the contribution.
- Ensure wheel platform name is correctly set when providing custom `CMAKE_OSX_DEPLOYMENT_TARGET` and `CMAKE_OSX_ARCHITECTURES` values are provided. Thanks @nonhermitian for the contribution. See #377.
- Fix testing with recent version of pytest by updating the pytest-runner requirements expression in *setup.py*. Thanks @mackelab for the contribution.

11.4 Scikit-build 0.8.1

11.4.1 Bug fixes

- Fix `bdist_wheel` command to support `wheel >= 0.32.0`. Thanks @fbudin69500 for reporting issue #360.

11.4.2 Tests

- Fix `test_distribution.py` updating use of `Path.files()` and requiring `path.py>=11.5.0`.

11.5 Scikit-build 0.8.0

11.5.1 New Features

- Introduced *`skbuild.constants.CMAKE_DEFAULT_EXECUTABLE`* to facilitate distribution of scikit-build in package manager like Nixpkgs where all paths to dependencies are hardcoded. Suggested by @FRidh.
- Setup keywords:
  - If not already set, `zip_safe` option is set to `False`. Suggested by @blowekamp.
- Add support for `--skip-generator-test` when a generator is explicitly selected using `--generator`. This allows to speed up overall build when the build environment is known.

11.5.2 Bug fixes

- Fix support for building project with CMake source directory outside of the `setup.py` directory. See #335 fixed by @massich.
- Fix reading of *.cmake* files having any character not available in CP-1252 (the default code page on windows). See #334 fixed by @bgermann.
- Fix parsing of macOS specific arguments like `--plat-name macosx-X.Y-x86_64` and `-DCMAKE_OSX_DEPLOYMENT_TARGET:STRING=X.Y` and ensure that the ones specified as command line arguments override the default values or the one hard-coded in the `cmake_args` setup keyword. Thanks @yonip for the help addressing #342.
- Support case where relative directory set in *package_dir* has an ending slash. For example, specifying `package_dir={'awesome': 'src/awesome/'}`, is now properly handled.
- Fix support for isolated build environment ensuring the CMake project is reconfigured when `pip install -e .` is called multiple times. See #352.

11.5.3 Documentation

- README: Update overall download count.
- Add logo and update sphinx configuration. Thanks @SteveJordanKW for the design work.
- Update *CMake installation* section. Thanks @thewtex.
- Add *Support for isolated build* section.

- Add *Optimized incremental build* section.
- Update *usage documentation* to specify that `--universal` and `--python-tags` have no effect. Thanks @bgermann for the suggestion. See #353.
- Simplify documentation merging *Extension Build System* section with the *Advanced Usage* section. Thanks @thewtex for the suggestion.

11.5.4 Tests

- Add `check_wheel_content` utility function.
- Skip *test_setup_requires_keyword_include_cmake* if running in conda test environment or if https://pypi.org is not reachable. Suggested by @Luthaf.
- Continuous Integration
  - TravisCI:
    * Remove testing of linux now covered by CircleCI, add testing for Python 3.5, 3.6 and 3.7 on macOS.
    * Ensure system python uses latest version of pip
  - AppVeyor, CircleCI: Add testing for Python 3.7
  - Remove uses of unneeded $<RUN_ENV> command wrapper. scikit-build should already take care of setting up the expected environment.
  - Always install up-to-date scikit-ci and scikit-ci-addons.
  - Simplify release process managing *versioning* with python-versioneer and update *Making a release* documentation.

11.6 Scikit-build 0.7.1

11.6.1 Documentation

- Fix description and classifier list in setup.py.
- Fix link in README.

11.7 Scikit-build 0.7.0

11.7.1 New Features

- Faster incremental build by re-configuring the project only if needed. This was achieved by (1) adding support to retrieve the environment mapping associated with the generator set in the `CMakeCache.txt` file, (2) introducing a *`CMake spec file`* storing the CMake version as well as the the CMake arguments and (3) re-configuring only if either the generator or the CMake specs change. Thanks @xoviat for the contribution. See #301.
- CMake modules:
  - CMake module *PythonExtensions*: Set symbol visibility to export only the module init function. This applies to GNU and MSVC compilers. Thanks @xoviat. See #299.

  - Add CMake module *F2PY* useful to find the `f2py` executable for building Python extensions with Fortran. Thanks to @xoviat for moving forward with the integration. Concept for the module comes from the work of @scopatz done in PyNE project. See #273.
  - Update CMake module *NumPy* setting variables `NumPy_CONV_TEMPLATE_EXECUTABLE` and `NumPy_FROM_TEMPLATE_EXECUTABLE`. Thanks @xoviat for the contribution. See #278.
- Setup keywords:
  - Add support for *cmake_languages* setup keyword.
  - Add support for `include_package_data` and `exclude_package_data` setup keywords as well as parsing of `MANIFEST.in`. See #315. Thanks @reiver-dev for reporting the issue.
  - Add support for `cmake_minimum_required_version` setup keyword. See #312. Suggested by @henryiii.
  - Install cmake if found in `setup_requires` list. See #313. Suggested by @henryiii.
- Add support for `--cmake-executable` scikit-build command line option. Thanks @henryborchers for the suggestion. See #317.
- Use `_skbuild/platform-X.Y` instead of `_skbuild` to build package. This allows to have a different build directory for each python version. Thanks @isuruf for the suggestion and @xoviat for contributing the feature. See #283.
- Run cmake and `develop` command when command `test` is executed.

11.7.2 Bug fixes

- Fix support of `--hide-listing` when building wheel.
- CMake module *Cython*: Fix escaping of spaces associated with `CYTHON_FLAGS` when provided as command line arguments to the cython executable through CMake cache entries. See #265 fixed by @neok-m4700.
- Ensure package data files specified in the `setup()` function using `package_data` keyword are packaged and installed.
- Support specifying a default directory for all packages not already associated with one using syntax like `package_dir={'':'src'}` in `setup.py`. Thanks @benjaminjack for reporting the issue. See #274.
- Improve `--skip-cmake` command line option support so that it can re-generate a source distribution or a python wheel without having to run cmake executable to re-configure and build. Thanks to @jonwoodring for reporting the issue on the mailing list.
- Set `skbuild <version>` as wheel generator. See PEP-0427 and #191.
- Ensure `MANIFEST.in` is considered when generating source distribution. Thanks @seanlis for reporting the problem and providing an initial patch, and thanks @henryiii for implementing the corresponding test. See #260.
- Support generation of source distribution for git repository having submodules. This works only for version of git >= 2.11 supporting the `--recurse-submodules` option with `ls-files` command.

11.7.3 Internal API

- Add `skbuild.cmaker.get_cmake_version()`.

11.7.4 Python Support

- Tests using Python 3.3.x were removed and support for this version of python is not guaranteed anymore. Support was removed following the deprecation warnings reported by version 0.31.0 of wheel package, these were causing the tests `test_source_distribution` and `test_wheel` to fail.

11.7.5 Tests

- Speedup execution of tests that do not require any CMake language enabled. This is achieved by (1) introducing the test project `hello-no-language`, (2) updating test utility functions `execute_setup_py` and `project_setup_py_test` to accept the optional parameter `disable_languages_test` allowing to skip unneeded compiler detection in test project used to verify that the selected CMake generator works as expected, and (3) updating relevant tests to use the new test project and parameters.

  Overall testing time on all continuous integration services was reduced:

  - AppVeyor:
    - from **~16 to ~7** minutes for 64 and 32-bit Python 2.7 tests done using Visual Studio Express 2008
    - from more than **2 hours to ~50 minutes** for 64 and 32-bit Python 3.5 tests done using Visual Studio 2015. Improvement specific to Python 3.x were obtained by caching the results of slow calls to `distutils.msvc9compiler.query_vcvarsall` (for Python 3.3 and 3.4) and `distutils._msvccompiler._get_vc_env` (for Python 3.5 and above). These functions were called multiple times to create the list of *`skbuild.platform_specifics.windows.CMakeVisualStudioCommandLineGenerator`* used in *`skbuild.platform_specifics.windows.WindowsPlatform`*.
  - CircleCI: from **~7 to ~5** minutes.
  - TravisCI: from **~21 to ~10** minutes.
- Update maximum line length specified in flake8 settings from 80 to 120 characters.
- Add `prepend_sys_path` utility function.
- Ensure that the project directory is prepended to `sys.path` when executing test building sample project with the help of `execute_setup_py` function.
- Add codecov config file for better defaults and prevent associated Pull Request checks from reporting failure when coverage only slightly changes.

11.7.6 Documentation

- Improve internal API documentation:
  - *`skbuild.platform_specifics.windows`*
  - *`skbuild.command`*
  - *`skbuild.command.generate_source_manifest`*
  - *`skbuild.utils`*
- Split usage documentation into a *Basic Usage* and *Advanced Usage* sections.

11.7.7 Cleanups

- Fix miscellaneous pylint warnings.

11.8 Scikit-build 0.6.1

11.8.1 Bug fixes

- Ensure CMake arguments passed to scikit-build and starting with `-DCMAKE_*` are passed to the test project allowing to determine which generator to use. For example, this ensures that arguments like `-DCMAKE_MAKE_PROGRAM:FILEPATH=/path/to/program` are passed. See #256.

11.8.2 Documentation

- Update *Making a release* section including instructions to update `README.rst` with up-to-date pypi download statistics based on Google big table.

11.9 Scikit-build 0.6.0

11.9.1 New features

- Improve `py_modules` support: Python modules generated by CMake are now properly included in binary distribution.
- Improve developer mode support for `py_modules` generated by CMake.

11.9.2 Bug fixes

- Do not implicitly install python modules when the beginning of their name match a package explicitly listed. For example, if a project has a package `foo/__init__.py` and a module `fooConfig.py`, and only package `foo` was listed in `setup.py`, `fooConfig.py` is not installed anymore.
- CMake module *targetLinkLibrariesWithDynamicLookup*: Fix the caching of *dynamic lookup* variables. See #240 fixed by @blowekamp.

11.9.3 Requirements

- wheel: As suggested by @thewtex, unpinning version of the package by requiring `>=0.29.0` instead of `==0.29.0` will avoid uninstalling a newer version of wheel package on up-to-date system.

11.9.4 Documentation

- Add a command line *CMake Options* section to *Usage*.
- Fix *table* listing *Visual Studio IDE* version and corresponding with *CPython version* in *C Runtime, Compiler and Build System Generator*.
- Improve *Making a release* section.

11.9.5 Tests

- Extend `test_hello`, `test_setup`, and `test_sdist_hide_listing` to (1) check if python modules are packaged into source and wheel distributions and (2) check if python modules are copied into the source tree when developer mode is enabled.

11.9.6 Internal API

- Fix `skbuild.setuptools_wrap.strip_package()` to handle empty package.
- Teach `skbuild.command.build_py.build_py.find_modules()` function to look for *py_module* file in `CMAKE_INSTALL_DIR`.
- Teach `skbuild.utils.PythonModuleFinder` to search for *python module* in the CMake install tree.
- Update `skbuild.setuptools_wrap._consolidate()` to copy file into the CMake tree only if it exists.
- Update `skbuild.setuptools_wrap._copy_file()` to create directory only if there is one associated with the destination file.

11.10 Scikit-build 0.5.1

11.10.1 Bug fixes

- Ensure file copied in "develop" mode have "mode bits" maintained.

11.11 Scikit-build 0.5.0

11.11.1 New features

- Improve user experience by running CMake only if needed. See #207
- Add support for *cmake_with_sdist* setup keyword argument.
- Add support for `--force-cmake` and `--skip-cmake` global *setup command-line options*.
- scikit-build conda-forge recipe added by @isuruf. See conda-forge/staged-recipes#1989
- Add support for development mode. (#187).
- Improved *C Runtime, Compiler and Build System Generator* selection:
- If available, uses *Ninja* build system generator on all platforms. An advantages is that ninja automatically parallelizes the build based on the number of CPUs.
- Automatically set the expected *Visual Studio* environment when `Ninja` or `NMake Makefiles` generators are used.
- Support Microsoft Visual C++ Compiler for Python 2.7. See #216.
- Prompt for user to install the required compiler if it is not available. See #27.
- Improve *targetLinkLibrariesWithDynamicLookup* CMake Module extending the API of `check_dynamic_lookup` function:
- Update long signature: `<LinkFlagsVar>` is now optional

- Add support for short signature: `check_dynamic_lookup(<ResultVar>)`. See SimpleITK/SimpleITK#80.

11.11.2 Bug fixes

- Fix scikit-build source distribution and add test. See #214 Thanks @isuruf for reporting the issue.
- Support building extension within a virtualenv on windows. See #119.

11.11.3 Documentation

- add *C Runtime, Compiler and Build System Generator* section
- add *Release Notes* section
- allow github issues and users to easily be referenced using ``:issue:`XY` `` and ``:user:`username` `` markups. This functionality is enabled by the sphinx-issue sphinx extension
- make_a_release: Ensure uploaded distributions are signed
- usage:
- Add empty cross-compilation / wheels building sections
- Add *Why should I use scikit-build ?*
- Add *Setup options* section
- hacking:
- Add *Internal API* section generated using *sphinx-apidoc*.
- Add *Internal CMake Modules* to document *targetLinkLibrariesWithDynamicLookup* CMake module.

11.11.4 Requirements

- setuptools: As suggested by @mivade in #212, remove the hard requirement for `==28.8.0` and require version `>= 28.0.0`. This allows to "play" nicely with conda where it is problematic to update the version of setuptools. See pypa/pip#2751 and ContinuumIO/anaconda-issues#542.

11.11.5 Tests

- Improve "push_dir" tests to not rely on build directory name. Thanks @isuruf for reporting the issue.
- travis/install_pyenv: Improve MacOSX build time updating scikit-ci-addons
- Add `get_cmakecache_variables` utility function.

11.11.6 Internal API

- *`skbuild.cmaker.CMaker.configure()`*: Change parameter name from `generator_id` to `generator_name`. This is consistent with how generator are identified in CMake documentation. This change breaks backward compatibility.
- *`skbuild.platform_specifics.abstract.CMakePlatform.get_best_generator()`*: Change parameter name from `generator` to `generator_name`. Note that this function is also directly importable from *`skbuild.platform_specifics`*. This change breaks backward compatibility.

- *skbuild.platform_specifics.abstract.CMakeGenerator*: This class allows to handle generators as sophisticated object instead of simple string. This is done anticipating the support for CMAKE_GENERATOR_PLATFORM and CMAKE_GENERATOR_TOOLSET. Note also that the class is directly importable from *skbuild.platform_specifics* and is now returned by `skbuild.platform_specifics.get_best_generator()`. This change breaks backward compatibility.

11.11.7 Cleanups

- appveyor.yml:
- Remove unused "on_failure: event logging" and "notifications: GitHubPullRequest"
- Remove unused SKIP env variable

11.12 Scikit-build 0.4.0

11.12.1 New features

- Add support for `--hide-listing` option
- allow to build distributions without displaying files being included
- useful when building large project on Continuous Integration service limiting the amount of log produced by the build
- CMake module: `skbuild/resources/cmake/FindPythonExtensions.cmake`
- Function `python_extension_module`: add support for module suffix

11.12.2 Bug fixes

- Do not package python modules under "purelib" dir in non-pure wheel
- CMake module: `skbuild/resources/cmake/targetLinkLibrariesWithDynamicLookup.cmake`:
- Fix the logic checking for cross-compilation (the regression was introduced by #51 and #47
- It configure the text project setting CMAKE_ENABLE_EXPORTS to ON. Doing so ensure the executable compiled in the test exports symbols (if supported by the underlying platform)

11.12.3 Docs

- Add short note explaining how to include scikit-build CMake module
- Move "Controlling CMake using scikit-build" into a "hacking" section
- Add initial version of "extension_build_system" documentation

11.12.4 Tests

- tests/samples: Simplify project removing unneeded install rules and file copy
- Simplify continuous integration

- use scikit-ci and scikit-ci-addons
- speed up build setting up caching
- Makefile:
- Fix *coverage* target
- Add *docs-only* target allowing to regenerate the Sphinx documentation without opening a new page in the browser.

11.13 Scikit-build 0.3.0

11.13.1 New features

- Improve support for "pure", "CMake" and "hybrid" python package
- a "pure" package is a python package that have all files living in the project source tree
- an "hybrid" package is a python package that have some files living in the project source tree and some files installed by CMake
- a "CMake" package is a python package that is fully generated and installed by CMake without any of his files existing in the source tree
- Add support for source distribution. See #84
- Add support for setup arguments specific to scikit-build:
- `cmake_args`: additional option passed to CMake
- `cmake_install_dir`: relative directory where the CMake project being built should be installed
- `cmake_source_dir`: location of the CMake project
- Add CMake module `FindNumPy.cmake`
- Automatically set `package_dir` to reasonable defaults
- Support building project without CMakeLists.txt

11.13.2 Bug fixes

- Fix dispatch of arguments to setuptools, CMake and build tool. See #118
- Force binary wheel generation. See #106
- Fix support for `py_modules` (6716723)
- Do not raise error if calling "clean" command twice

11.13.3 Documentation

- Improvement of documentation published on http://scikit-build.readthedocs.io/en/latest/
- Add docstrings for most of the modules, classes and functions

11.13.4 Tests

- Ensure each test run in a dedicated temporary directory
- Add tests to raise coverage from 70% to 91%
- Refactor CI testing infrastructure introducing CI drivers written in python for AppVeyor, CircleCI and TravisCI
- Switch from `nose` to `py.test`
- Relocate sample projects into a dedicated home: https://github.com/scikit-build/scikit-build-sample-projects

11.13.5 Cleanups

- Refactor commands introducing `set_build_base_mixin` and `new_style`
- Remove unused code

CHAPTER 12

Making a release

A core developer should use the following steps to create a release *X.Y.Z* of **scikit-build** on PyPI and Conda.

12.1 Prerequisites

- All CI tests are passing on AppVeyor, Azure Pipelines, CircleCI and Travis CI.
- You have a GPG signing key.

12.2 Documentation conventions

The commands reported below should be evaluated in the same terminal session.

Commands to evaluate starts with a dollar sign. For example:

```
$ echo "Hello"
Hello
```

means that `echo "Hello"` should be copied and evaluated in the terminal.

12.3 Setting up environment

1. First, register for an account on PyPI.
2. If not already the case, ask to be added as a `Package Index Maintainer`.
3. Create a `~/.pypirc` file with your login credentials:

```
[distutils]
index-servers =
  pypi
  pypitest

[pypi]
username=<your-username>
password=<your-password>

[pypitest]
repository=https://test.pypi.org/legacy/
username=<your-username>
password=<your-password>
```

where `<your-username>` and `<your-password>` correspond to your PyPI account.

12.4 PyPI: Step-by-step

1. Make sure that all CI tests are passing on AppVeyor, Azure Pipelines, CircleCI and Travis CI.
2. Download the latest sources

```
$ cd /tmp && \
  git clone git@github.com:scikit-build/scikit-build && \
  cd scikit-build
```

3. List all tags sorted by version

```
$ git fetch --tags && \
  git tag -l | sort -V
```

4. Choose the next release version number

```
$ release=X.Y.Z
```

> **Warning:** To ensure the packages are uploaded on PyPI, tags must match this regular expression: `^[0-9]+(\.[0-9]+)*(\.post[0-9]+)?$`.

5. In *README.rst*, update PyPI download count after running this big table query and commit the changes.

```
$ git add README.rst && \
  git commit -m "README: Update download stats [ci skip]"
```

Note: To learn more about *pypi-stats*, see How to get PyPI download statistics.

6. In *CHANGES.rst* replace `Next Release` section header with `Scikit-build X.Y.Z` and commit the changes.

```
$ git add CHANGES.rst && \
  git commit -m "Scikit-build ${release}"
```

7. Tag the release

```
$ git tag --sign -m "Scikit-build ${release}" ${release} master
```

Warning: We recommend using a GPG signing key to sign the tag.

8. Create the source distribution and wheel

```
$ python setup.py sdist bdist_wheel
```

9. Publish the both release tag and the master branch

```
$ git push origin ${release} && \
  git push origin master
```

10. Upload the distributions on PyPI

```
twine upload dist/*
```

Note: To first upload on TestPyPI , do the following:

```
$ twine upload -r pypitest dist/*
```

11. Create a clean testing environment to test the installation

```
$ pushd $(mktemp -d) && \
  mkvirtualenv scikit-build-${release}-install-test && \
  pip install scikit-build && \
  python -c "import skbuild"
```

Note: If the `mkvirtualenv` command is not available, this means you do not have virtualenvwrapper installed, in that case, you could either install it or directly use virtualenv or venv.

To install from TestPyPI, do the following:

```
$ pip install -i https://test.pypi.org/simple scikit-build
```

12. Cleanup

```
$ popd && \
  deactivate  && \
  rm -rf dist/* && \
  rmvirtualenv scikit-build-${release}-install-test
```

13. Add a `Next Release` section back in *CHANGES.rst*, commit and push local changes.

```
$ git add CHANGES.rst && \
  git commit -m "CHANGES.rst: Add \"Next Release\" section [ci skip]" && \
  git push origin master
```

12.5 Conda: Step-by-step

Warning: Publishing on conda requires to have corresponding the corresponding Github release.

After a GitHub release is created in the scikit-build project and after the conda-forge Autoticking Bot creates a pull request on the scikit-build-feedstock , follow these steps to finalize the conda package release:

1. Review and update scikit-build-feedstock pull request to include Python 3.5 support (see here for an example)
2. Merge pull-request

In case the bot failed (e.g because of GH rate limitation) and in order to explicitly release a new version on conda-forge, follow the steps below:

1. Choose the next release version number (that matches with the PyPI version last published)

```
$ release=X.Y.Z
```

2. Fork scikit-build-feedstock

   First step is to fork scikit-build-feedstock repository. This is the recommended best practice by conda.

3. Clone forked feedstock

   Fill the YOURGITHUBUSER part.

```
$ cd /tmp && git clone https://github.com/YOURGITHUBUSER/scikit-build-feedstock.
↪git
```

4. Download corresponding source for the release version

```
$ cd /tmp && \
  wget https://github.com/NeurodataWithoutBorders/scikit-build/releases/
↪download/$release/scikit-build-$release.tar.gz
```

5. Create a new branch

```
$ cd scikit-build-feedstock && \
  git checkout -b $release
```

6. Modify `meta.yaml`

   Update the version string and sha256.

   We have to modify the sha and the version string in the `meta.yaml` file.

   For linux flavors:

```
$ sed -i "2s/.*/{% set version = \"$release\" %}/" recipe/meta.yaml
$ sha=$(openssl sha256 /tmp/scikit-build-$release.tar.gz | awk '{print $2}')
$ sed -i "3s/.*/{$ set sha256 = \"$sha\" %}/" recipe/meta.yaml
```

   For macOS:

```
$ sed -i -- "2s/.*/{% set version = \"$release\" %}/" recipe/meta.yaml
$ sha=$(openssl sha256 /tmp/scikit-build-$release.tar.gz | awk '{print $2}')
$ sed -i -- "3s/.*/{$ set sha256 = \"$sha\" %}/" recipe/meta.yaml
```

7. Push the changes

```
$ git push origin $release
```

8. Create a Pull Request

   Create a pull request against the main repository. If the tests are passed a new release will be published on Anaconda cloud.

CHAPTER 13

Notes

13.1 *sysconfig* vs *distutils.sysconfig*

After installing CPython, two sysconfig modules are available:

- *sysconfig*
- *distutils.sysconfig*

A difference is the value associated with the `EXT_SUFFIX` and `SO` configuration variables.

		Linux	macOS	Windows
		CPython 2.7		
SO	**sysconfig**	.so	.so	.pyd
	distutils.sysconfig			
EXT_SUFFIX	**sysconfig**	None	None	None
	distutils.sysconfig			
		CPython >= 3.5		
SO	**sysconfig**	.cpython-37m-x86_64-linux-gnu.so	.cpython-37m-darwin.so	.pyd
	distutils.sysconfig			.cp37-win_amd64.pyd
EXT_SUFFIX	**sysconfig**			.pyd
	distutils.sysconfig			.cp37-win_amd64.pyd

Note: The `EXT_SUFFIX` was introduced in Python 3.4 and is functionally equivalent to `SO` configuration variable. The `SO` configuration variable has been deprecated since Python 3.4.

Note: The information reported in the table have been collected execution the following python snippet.

```python
def display_ext_suffix_config_var():
    import platform
    import sys
    import sysconfig
    from distutils import sysconfig as du_sysconfig
    details = (platform.python_implementation(),) + sys.version_info[:3]
    print("%s %s.%s.%s" % details)
    print(" SO")
    print("  sysconfig           : %s" % sysconfig.get_config_var('SO'))
    print("  distutils.sysconfig : %s" % du_sysconfig.get_config_var('SO'))
    print(" EXT_SUFFIX")
    print("  sysconfig           : %s" % sysconfig.get_config_var('EXT_SUFFIX'))
    print("  distutils.sysconfig : %s" % du_sysconfig.get_config_var('EXT_SUFFIX'))

display_ext_suffix_config_var()
```

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

CHAPTER 15

Resources

- Free software: MIT license
- Documentation: http://scikit-build.readthedocs.io/en/latest/
- Source code: https://github.com/scikit-build/scikit-build
- Mailing list: https://groups.google.com/forum/#!forum/scikit-build

Python Module Index

S

Index

Symbols

A

B

C

D

E

F

G

H

I

T

U

V

W