Scikit-build-core

A modern build-backend for CPython C/C++/Fortran/Cython extensions

Henry Schreiner[1] ⓘ ✉, **Jean-Christophe Fillion-Robin**[2] ⓘ ✉, and **Matt McCormick**[2] ⓘ ✉

[1]Princeton University, [2]Kitware Inc.

Abstract

Discover how scikit-build-core revolutionizes Python extension building with its seamless integration of CMake and Python packaging standards. Learn about its enhanced features for cross-compilation, multi-platform support, and simplified configuration, which enable writing binary extensions with pybind11, Nanobind, Fortran, Cython, C++, and more. Dive into the transition from the classic scikit-build to the robust scikit-build-core and explore its potential to streamline package distribution across various environments.

Keywords   Build system, CMake

Correspondence to
Henry Schreiner
henryfs@princeton.edu

Open Access   ⓒ ⓐ

1. Introduction

Python packaging has evolved significantly in the last few years. Standards have been written to allow the development of new build backends not constrained by setuptool's complex legacy. Build time dependencies, once nearly impossible to rely on, are now the standard, and required for any build system, including the original setuptools. And this new system is controlled by one central location, the `pyproject.toml` file.

We present scikit-build-core: a new build system based on these standards that brings together Python packaging and the CMake build system, the popular general purpose build system for languages like C, C++, Fortran, Cython, and CUDA. Together, this new system provides a simple entry point to building extensions that still scales to major projects. This allows everyone to take advantage of existing libraries and harness the performance available in these languages.

We will look at creating a package with scikit-build-core. Then we will cover some of it's most interesting and innovative features. Then we will peer into the design and internal workings. Finally, we will look at some of the packages that adopted scikit-build-core. A lot of the ecosystem was improved, even beyond scikit-build-core, as part of this project, so we will also highlight some of that work.

2. History of packaging

Python has a long history compared to modern languages with first-party packaging solutions; packaging wasn't something that was considered important for Python for quite a while. Python gained a standard library module to help with packaging called `distutils` in Python 1.6 and 2.0, in the year 2000, nearly ten years after the initial release. Distribution was difficult, leading to packages containing large numbers of distinct modules (such as SciPy [1]) to reduce the number of packages one had to figure out how to install, and "distributions" of Python to be created, such as the Enthought distribution. This eventually

led to the creation of Conda, a modular binary package manager that was particularly good at (and written in) Python.

Several developments greatly improved Python's packaging story, such the addition of a new binary format, the "wheel" made distributing binaries easier. This, along with new hardware and a changing installation landscape highlighted an issue with putting a packaging tool in the standard library: you can't freely update the standard library. Third party "extensions" to distutils appeared; the one that became the main tool for many years is setuptools. It wasn't long before setuptools became required; building directly with distutils was too buggy, and setuptools was injecting patches into distutils when it loaded.

Package installers, originally `easy_install` and then the more full-featured `pip` came along. `pip` was tightly coupled to setuptools, and even helped it out by making sure it's injections to distutils were done, even if packages didn't import setuptools first or at all. Pip was directly and deeply tied to setuptools; if you wanted to build a package, setuptools was the only answer. Even if you make the SDist (Python's source distribution format) and wheel(s) (Python's binary distribution format) yourself, which was actually pretty easy, as those were standardized, you couldn't make sure that `pip` wouldn't try to use setuptools to build a wheel from an SDist.

Faults in the organically grown distutils/setuptools quickly became apparent. You couldn't tell the installer to update setuptools from `setup.py`, because it was just a Python file that was being run. It was hard to declare built-time dependencies (though they really did try by running the Python file twice, the first time with stubs). You couldn't parse metadata without running the setup.py. It was hard to extend the setuptools commands, and the entire API was public, which meant it was really hard to fix something without breaking everyone else.

Third party tools for building packages started showing up, like Flit and Poetry. These worked by making a compatibility `setup.py` and injecting it into the SDist, just in case pip needed to build the wheel. This is when standardization efforts, in the form of PEPs, began to change the packaging landscape forever.

PEP 517 [2] defined an API for build-frontends (like pip) to talk to build-backends (like setuptools). PEP 518 [3] defined an isolated build environment, which would allow builders to download build dependencies, like a specific version of setuptools or a plugin. Later, PEP 621 [4] would add a standard way to define basic package metadata, and PEP 660 [5] would add an API for editable installs.

Python build backends started to appear. Most of the initial build backends were were designed for Python-only packages. `flit-core`, `poetry-core`, `pdm-backend`, and `hatchling` are some popular build backends that support some or all of these PEPs. Setuptools also ended up gaining support for all these PEPs, as well.

Compiled backends were a bit slower, but today we have several great choices. `scikit-build-core` for CMake, `meson-python` for Meson, and `maturin` for Cargo (Rust) are the most popular. `enscons` for SCons should get a special mention as the first binary build backend, even though it is mostly a historical curiosity at this point.

A side note on Conda: like most general package managers, it runs arbitrary commands from a recipe to build and install the package, then it captures the installed files. Many Python packages use the `pip install .` command, but any commands can be placed here, including native CMake calls; as long files are placed in the right place, the result can be used from conda installed Python. Though packages doing this may forget to generate a `<package>.dist-info` directory, which is used by things like `importlib.metadata`.

3. Scikit-build (classic)

The original scikit-build [6] was released as PyCMake at SciPy 2014, and renamed two years later, at SciPy 2016, following the "scikit" convention introduced by SciPy. Being developed well before the packaging PEPs, it was designed as a wrapper around distutils and/or setuptools. This design had flaws, but was the only way this could be done at the time.

Because it was deeply tied to setuptools internals, updates to setuptools or wheel often would break scikit-build. There were a lot of limitations of setuptools that scikit-build couldn't alleviate properly.

However, it did allow users to use a real build system (CMake) with their Python projects. A notable example are two packages produced by the scikit-build team: ninja and cmake redistributions on PyPI. Users could pip install `cmake` and `ninja` anywhere that wheels worked.

In 2021, a proposal was written and accepted by the NSF to fund development on a new package, scikit-build-core, built on top of the packaging standards and free from setuptools and other historical cruft. Work started in 2022, and the first major package to switch was Awkward Array, at the end of that year.

4. Using scikit-build-core

4.1. Binary Python package generation overview

Figure 1 provides an overview of modern binary Python package generation. The package developer begins by defining the package structure, incorporating package source files, a CMakeLists.txt CMake configuration, and a pyproject.toml file adhering to PEP 517 specifications. To initiate the package build process, the developer employs a Build Frontend tool, such as `pip` or `build`, and may supply optional configuration parameters. The chosen Build Frontend then triggers the PEP 517 Build Backend, exemplified by scikit-build-core, which in turn employs CMake, managing configuration options appropriately. Utilizing a blend of build system introspection and configured options alongside the *CMakeLists.txt* logic, CMake generates a native build system configuration. This native build system, whether Ninja configuration files, Unix Makefiles, Visual Studio Project Files, etc., defines dependencies for the artifacts produced by the native toolchain, which is responsible for compiling C/C++ source code into native binaries.

Once configured, scikit-build-core initiates the native build system, managing the bundling of resulting build artifacts and metadata into a wheel. These wheels may undergo additional processing with post-processing tools to enhance platform compatibility. End-users of the package have the option to directly utilize the wheel via a package manager or generate their own package by invoking the build system from a source distribution (SDist). Typically, both the binary wheels and the source distributions are uploaded to the Python Package Index (PyPI) for broader accessibility.

4.2. The simplest example

Scikit-build-core was designed to be easy to use as possible, with reasonable defaults and simple configuration. A minimal, working example is possible with just three files. You need a file to build, for example, this is a simple pybind11 module in a `main.cpp`:
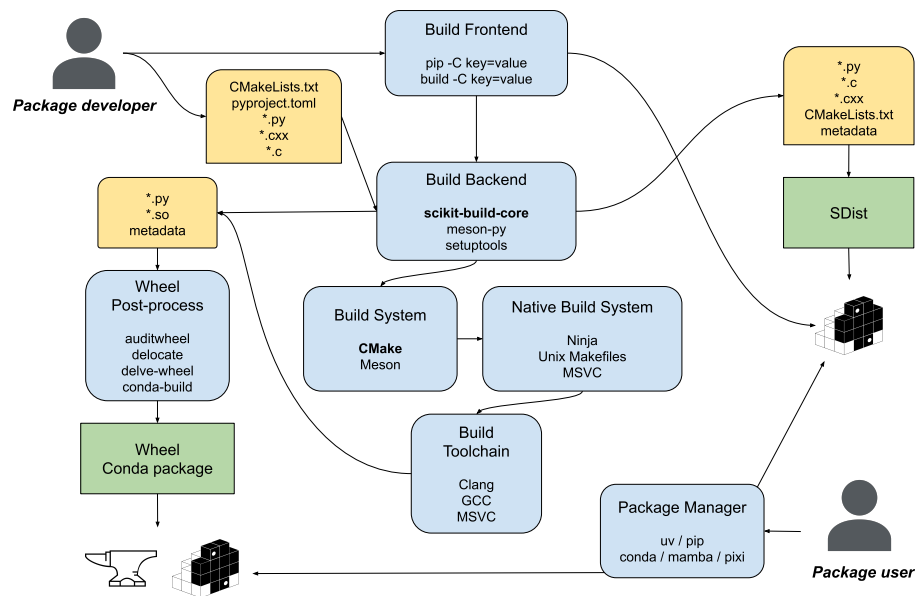
Figure 1. *Binary Python packaging dataflow.*

```cpp
#include <pybind11/pybind11.h>

PYBIND11_MODULE(example, m) {
    m.def("square", [](double x) { return x*x; });
}
```

You then need a `pyproject.toml`, and it can be as little as six lines long, just like for pure Python:

```toml
[build-system]
requires = ["scikit-build-core", "pybind11"]
build-backend = "scikit_build_core.build"

[project]
name = "example"
version = "0.0.1"
```

And finally, you need a `CMakeLists.txt` for CMake, which is also as little as six lines:

```cmake
cmake_minimum_required(VERSION 3.15...3.26)
project(example LANGUAGES CXX)

set(PYBIND11_NEWPYTHON ON)
find_package(pybind11 CONFIG REQUIRED)

pybind11_add_module(example example.cpp)
install(TARGETS example LIBRARY DESTINATION .)
```

And that all that is needed to get started.

4.3. Common needs as simple configuration

Customizing scikit-build-core is done through the `[tool.scikit-build]` table in the `pyproject.toml`. An example configuration is shown below:

```
[tool.scikit-build]
minimum-version = "0.10"

build.verbose = true
logging.level = "INFO"

wheel.expand-macos-universal-tags = true
```

On line 2, you can see the `minimum-version` setting. This is a special setting that allows scikit-build-core to make changes to default values in the future while ensuring your configuration continues to build with the defaults present the version you specify. This is very similar to CMake's `cmake_minimum_requires` feature.

On line 4-5, you can see an example of increasing the verbosity, both to print out build commands as well as see scikit-build-core's logs.

On line 7, you can see that scikit-build-core can expand the macOS universal tags to include both ARM and Intel variants for maximum compatibility with older Pip versions; this is impossible to do in setuptools unless you use API declared private and unstable in wheel.

5. Innovative features of scikit-build-core

Scikit-build-core has some interesting and powerful innovations. This section is by no means exhaustive; it doesn't cover scikit-build-core's over forty different configuration options, for example. It is instead just meant to highlight some of the most interesting features of scikit-build-core.

5.1. Dynamic requirement on CMake/Ninja

Scikit-build-core needs CMake to build, and often ninja as well. Both are available as Python wheels, but simply adding them to your `build-system.requires` list is not a good idea, because some platforms aren't supported by wheels on PyPI, such as WebAssembly, the various BSD's, ClearLinux, or Android. But these systems can get CMake other ways, such as from a package manager, so users shouldn't have a build fail because they can't get a cmake wheel if they have a sufficient version of CMake already. As it turns out, PEP 517 is flexible enough to handle this very elegantly.

PEP 517 has an API for a build tool to declare it's dependencies. This was initially used for setuptools to only depend on wheel when making wheels, but it works perfectly for optional binary dependencies as well. Scikit-build-core will check to see if `cmake` is present on the system. If it is, and it reports a version sufficient for the package, it will not be added to the requested dependencies. This is also done for ninja. This same system was added to `meson-python`, as well, since Meson also requires ninja.

5.2. Integration with external packages

Scikit-build-core has three mechanisms to allow packages on PyPI to provide CMake configuration and tools:

- The site-packages directory is added to the CMake search path by scikit-build-core. Due to the way CMake config file discovery works, this allows a package to provide a `<PkgName>Config.cmake` file in any GNU-like standard location in the package (like `pybind11/share/cmake/pybind11/pybind11Config.cmake`, for example).
- An entry point `cmake.prefix`, which adds prefix dirs, useful for adding module files that don't have to match the installed package name or are in arbitrary locations.
- An entry point `cmake.module`, which can be used to add any CMake helper files.

5.3. Dual editable modes with automatic recompile

Editable installs are supported in two modes. The default mode installs a custom finder that combines the Python files in your development directory (as specified by packages) with CMake installed files in site-packages. It is important to rely on importlib.resources instead of file manipulation in this mode. This mode supports installs into multiple environments.

This mode also supports automatic rebuilds. If you enable it, then importing your package will rebuild on any changes to the source files, something that could not be done with setuptools.

There is also an opt-in "inplace" mode that uses CMake's inplace build to build your files in the source directory. This is very similar to setuptools `build_ext --inplace`, and works with tools that can't run Python finders, like type checkers. This mode works with a single environment, because the build dir *is* the source dir. There is no dynamic finder in this mode, so automatic rebuilds are not supported.

5.4. Dynamic metadata

A system for allowing custom metadata plugins was contributed and has been one of the most successful parts of scikit-build-core. Any metadata field other than the `name` can be dynamically specified in Python packaging; the dynamic-metadata mechanism provides a way for plugins to be written by anyone, including "in-tree" plugins written inside a specific package. This gives authors quite a bit of freedom to do things like customize the package description or read the version from a file.

This system is being worked on as a standard for other build backends to use and a stand-alone namespace. Changes are expected, but the current implementation in scikit-build-core uses the following API:

```python
def dynamic_metadata(
    field: str,
    settings: Mapping[str, Any],
) -> str:
    ...
```

Plugins provide this function. Three built-in plugins are provided in scikit-build-core: one for regex, one that wraps `setuptools_scm`, and one that wraps `hatch-fancy-pypi-readme`. Using one of these looks like this:

```toml
name = "mypackage"
dynamic = ["version"]

[tool.scikit-build.metadata.version]
provider = "scikit_build_core.metadata.regex"
input = "src/mypackage/__init__.py"
```

The `provider` key tells scikit-build-core where to look for the plugin. There is also a provider-path for local plugins. All other keys are implemented by the plugin; the regex plugin implements `input` and `regex`, for example.

An extra feature that is not being proposed for broader adoption is a generation mechanism. Scikit-build-core can generate a file for you with metadata provided via templating. It looks like this:

```
[[tool.scikit-build.generate]]
path = "package/_version.py"
template = '''
version = "${version}"
'''
```

This is extremely useful due to the `location` key, which defaults to `install`, which will put the file only in the built wheel, `build`, which puts the file in the build directory, and `source`, which writes it out to the source directory and SDist, much like `setuptools_scm` normally does. In the default mode, though, no generated files are placed in your source tree.

5.5. Overrides

Static configuration has many benefits, but it has a significant drawback; you often want to configure different situations differently. For example, you might want a higher minimum CMake version on Windows, or a pure Python version for unreleased Python versions. These sorts of things can be expressed with overrides, which was designed after overrides in cibuildwheel, which in turn were based on mypy's overrides. Scikit-build-core has the most powerful version of the system, however, with an `.if` table that can do version comparisons and regexs, and supports any/all merging of conditions, and inheritance from a previous override (also added to cibuildwheel). It includes conditions for the state of the build (wheel, SDist, editable, or metadata builds) and environment variables. An example is shown below:

```
[[tool.scikit-build.overrides]]
if.platform-system = "darwin"
cmake.version = ">=3.18"
```

This will change the minimum CMake version to `>=3.18` on macOS.

6. SCIKIT-BUILD-CORE'S DESIGN

This section is devoted to the internals of scikit-build-core.

6.1. The configuration system

Scikit-build-core has over forty options, and each of these options can be specified in the `pyproject.toml`, or via config-settings (`-C` in build or pip), or via environment variables starting with SKBUILD. We also provide a JSON Schema for the TOML configuration, and a list of all options in the README. All of these are powered by a single dataclass-based configuration system developed for scikit-build-core.

All of the configuration lives in dataclasses that look like this:

```python
@dataclasses.dataclass
class LoggingSettings:
    level: Literal["NOTSET", "DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"] = (
        "WARNING"
    )
    """
    The logging level to display, "DEBUG", "INFO", "WARNING", and "ERROR" are
    possible options.
    """


@dataclasses.dataclass
class ScikitBuildSettings:
    cmake: CMakeSettings = dataclasses.field(default_factory=CMakeSettings)
    ninja: NinjaSettings = dataclasses.field(default_factory=NinjaSettings)
    logging: LoggingSettings = dataclasses.field(default_factory=LoggingSettings)
    ...
```

The system is a conversion system, originally based on `cattrs`, but reimplemented in pure Python to reduce dependencies and the potential for uncontrollable breakage. The system recognised a wide variety of types, including nested types and as seen above, literals. A comment in a string following the option is also recognised for the JSONSchema/README processors, because nothing has been standardized for variable docstrings, and Sphinx understands this convention.

Converters are implemented for TOML, config-settings (dict), and environment variables. The converters fill the `ScikitBuildSettings` instance with the values from the three[1] sources. Once read in, these are just a normal fully typed dataclass, so access and usage is natural and protected by mypy. Overrides are currently implemented via pre-processing the TOML. A post processing step is also added to validate the settings, and produce errors/warnings based on invalid combinations.

6.2. The File API

CMake provides a File API, which allows third party tools to read a variety of information about the build process. Scikit-build-core has a full implementation of a reader for the File API for use in plugins (currently not required for scikit-build-core itself).

The File API was implemented via dataclasses following the official schema. A recursive conversion system, similar to the settings system, was implemented to convert the written files into dataclass instances.

6.3. Plugins for other systems

Scikit-build-core 0.9.0 includes two plugins, one for setuptools, and one for hatchling. It was designed to support making custom plugins wrapping CMake builds, so these two internal plugins are examples of using scikit-build-core to make plugins. These will eventually be pulled out into separate packages; being internal allowed faster iteration on the scikit-build-core API used by plugins.

7. ADOPTION

Projects moving to scikit-build-core have seen over 800 lines of convoluted `setup.py` code turn into 20 lines of CMake and a minimal configuration file. Often this adds new platforms, like Windows or PyPy, that were not feasible to support before, and faster multithreaded compile times.

---

[1]Converters are combined using a chain converter, which allows any combination of converters; this is used by the hatchling plugin to support four converters, since it includes hatchling's TOML configuration as well as scikit-build-core's.

7.1. Statistics

Scikit-build-core is currently the most downloaded build backend designed for compiled languages other than Rust[2], with 2.7M monthly downloads[3]. It is used by a dozen packages in the top 8,000 PyPI packages, like `pyzmq`, `lightgbm`, `cmake`, `phik`, and `clang-format`. Other packages like `ninja` and `boost-histogram` have adopted it, but have not made a release with it yet.

It is possible to download every pyproject.toml from PyPI[4] and perform analyses on packages that provided an SDist. You can compute the number of packages published using scikit-build-core (255 as of June 25, up from 82 last year), and investigate the details of how projects are setting configuration.

We also monitor non-fork mentions of `scikit_build_core` in `pyproject.toml`'s using GitHub Code Search, of which there are currently (July 3) 764 results, up from 92 on July 1, 2023). This string is required as part of the build backend, so serves as a reasonable method to track this info. A search for the more precise string `scikit_build_core.build`, which avoids hatchling and setuptools plugins, provides 720 results; a similar search was not performed last year for comparison, though.

7.2. *Rapids.ai*

Rapids.ai moved their package generator to scikit-build-core, affecting all their packages, like cudf, cugraph, cuml, and rmm. They have several unusual requirements due to the need to support cuda variants. They developed a wrapper for scikit-build-core that changes the name of the package based on the current cuda version (something explicitly disallowed by PEP 621) and injects modified dependencies. Discussions on ways to handle external dependencies like CUDA without such workarounds was a major topic at the packaging summit in PyCon US 24.

7.3. *Ninja / CMake / clang-format*

One powerful use case for binaries is PyPI redistribution of CLI tools. This is used for `ninja` and `cmake`, which are first-party scikit-build-core projects, along with many third party projects like clang-format, use this to great effect. For example, clang-format provides easily pip-installable wheels that are under 2 MB and work on almost any system. We are not aware of any easier way to get a working copy of clang-format.

The core idea for most of these projects is to install the binary in the `scripts` folder of a wheel, which will be placed directly in a user's path. Scikit-build-core sets CMake variables with the various wheel folders; this one is `${SKBUILD_SCRIPTS_DIR}`.

The other common need, and one very difficult to handle in a setuptools-based wrapper, stems from the fact that these binary packages don't build against Python, so they don't need to be build per Python version. This can be easily indicated in scikit-build-core by setting:

```
[tool.scikit-build]
wheel.py-api = "py3"
```

This will build a wheel that doesn't depend on the Python API and will only depend on the system architecture. This same setting can be used for the Limited API / Stable ABI as well, by setting `cp311` or a similar minimum value. Pythons before this value will build traditional wheels, and after will build Stable ABI wheels.

---

[2]The Rust-only Maturin is much older and has 57 packages in the top 8,000.
[3]https://hugovk.github.io/top-pypi-packages
[4]https://github.com/henryiii/pystats

7.4. PyZMQ

The python wrapper for the ZeroMQ project uses scikit-buid-core. This is their configuration:

```toml
[tool.scikit-build]
wheel.packages = ["zmq"]
wheel.license-files = ["licenses/LICENSE*"]
# 3.15 is required by scikit-build-core
cmake.version = ">=3.15"
# only build/install the pyzmq component
cmake.targets = ["pyzmq"]
install.components = ["pyzmq"]
```

They explicitly list the package name (since it doesn't match the project, `pyzmq` and their licences files are in a non-standard location. The `cmake.version` setting is superfluous, since that's already scikit-build-core's minimum. The `cmake.targets` (will be `build.targets` in future versions, with back-compatibility support) and `install.components` are quite powerful, though. The target list allows building just the required target for the binding, saving time. And then installing just things marked with a specific component gives you full control over exactly what goes into the wheel.

8. RELATED WORK

As part of this project, a lot of other packages were touched on or improved. `pybind11` had some build system improvements, especially related to modern FindPython support. Nanobind improved support for the Stable ABI, and the `nanobind_example` project moved to scikit-build-core. Cibuildwheel and build were improved for all backends with extensive testing and usage in scikit-build-core. validate-pyproject was improved and validate-pyproject-schema-store was added to make it easier to use the Schema Store's schema files, including scikit-build-core's, for validation of pyproject.toml's. Scikit-build-core (along with meson-python and maturin) were added to the Scientific Python Development Guide. Fixes were made in Pyodide to ensure better CMake support for WebAssembly builds (which were also added to cibuildwheel). And scikit-build-core was one of the first backends to support free-threaded Python 3.13 builds.

9. SUMMARY

Scikit-build-core is one of the best build systems available today for Python compiled extensions, allowing access to a powerful compiled build tool (CMake) from the comfort of a statically configured `pyproject.toml` file. It supports a wide variety of innovative controls, including many things not possible with a setuptools build, like good config-settings support, wheel tag customization, and more.

REFERENCES

[1] P. Virtanen *et al.*, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020, doi: https://doi.org/10.1038/s41592-019-0686-2.

[2] N. J. Smith and T. Kluyver, "A build-system independent format for source trees," 2015. [Online]. Available: https://www.python.org/dev/peps/pep-0517

[3] B. Cannon, N. J. Smith, and D. Stufft, "Specifying Minimum Build System Requirements for Python Projects," 2015. [Online]. Available: https://www.python.org/dev/peps/pep-0518

[4] B. Cannon, D. Ingram, P. Ganssle, S. Eustace, T. Kluyver, and ping Tzu-Chung, "Storing project metadata in pyproject.toml," 2020. [Online]. Available: https://www.python.org/dev/peps/pep-0621

[5] D. Holth and S. Bidoul, "Editable installs for pyproject.toml based builds (wheel based)," 2021. [Online]. Available: https://www.python.org/dev/peps/pep-0660

[6] J.-C. Fillion-Robin, M. McCormick, O. Padron, M. Smolens, M. Grauer, and M. Sarahan, "jcfr/scipy_2018_scikit-build_talk: SciPy 2018 Talk | scikit-build: A Build System Generator for CPython C/C++/Fortran/Cython Extensions." [Online]. Available: https://doi.org/10.5281/zenodo.2565368