The Joy of Packaging Documentation

Release 0.2

Assorted

Feb 28, 2023

CONTENTS

Scipy 2023 Tutorial

CHAPTER

ONE

PACKAGING

Packaging from start to finish including binary extensions using modern tools.

CHAPTER

TWO

TOPICS

2.1 Schedule

0:00: Overview of packaging

- SDists vs. wheels
- Pure Python vs. compiled packages
- Wheel vs conda packages
- PyPI / anaconda.org
- Links packaging documentation such as PyPA, Packaging Native

0:15: Exercise

- Identify platforms supported for the xxx packages on PyPI and anaconda.org

0:20: Virtual environments

- Setting up a virtual environment
- Setting up a conda environment
- Using a task runner (nox)

0:30: Exercise writing a noxfile

- Take existing working package and add a simple noxfile

0:50: Break & catch up

1:00: Pyproject.toml

- Essential specifications
- Optional specifications
- Specifying requirements
- Introduce the concept of "build-backend"

1:10: Exercise

- Fill in the missing pieces in a project.toml for a sample package
- Build a source distribution for the package

1:20: Building and uploading to PyPI: tools and package types

- Core tools

    - Pipx
    - build
    - twine: the secure way to upload to PyPI
  - For consolidated experience & dependency management
    - Pdm (https://pdm.fming.dev/latest/)
    - May be Hatch (https://hatch.pypa.io) (more like a replacement for tox and nox)
  - Building a source distribution
  - Building a wheel
  - Discuss use of delocate/Auditwheel/…
  - Difference between linux & manylinux wheels (internalize dependencies, glibc compatibility, …)

1:35: Worked example/exercise: building a package and uploading to pypi

  - Continuing from the the previous exercise, build a wheel for the package
  - Register the package on the pypi testing server
  - Upload the built distributions using twine
  - Delete one of the uploaded files on pypi and try re-uploading (will fail)
  - Introduce the idea of .post releases (it will happen to everyone who uploads)

1:45: Coffee break

2:05: Binaries and dependencies: how scikit-build can make life easier

  - Scikit-build overview & motivation
  - Adding a minimal CMakeLists.txt
  - Building the extension
  - Adding options and controlling the build

2:30: Exercise: add CMake project that generates python extension.

  - Tie it into previous python project.
  - Setup build caching

2:50: Break & catch up

3:00: Automated building with cloud-based CI services

  - GitHub action
  - Pre-commit.yml
    - Ruff
  - https://cibuildwheel.readthedocs.io/en/stable/

3:15: Exercise:

  - Update previous example adding cibuildwheel support
  - Linting using pre-commit + Ruff
  - Automated PyPI release

3:30: Handling dependencies

- "In-project" compilation
- External

3:45: Exercise

- Add a dependency to the project
  - pybind11 (in-project)
  - lz4 (external)

2.2 Overview of packaging for Python

2.2.1 SDists vs. wheels

2.2.2 Pure Python vs. compiled packages

2.2.3 Wheel vs conda packages

2.2.4 PyPI / anaconda.org

2.2.5 Links packaging documentation such as PyPA, Packaging Native

2.3 Environments and task runners

You will see two *very* common recommendations when installing a package:

```
$ pip install <package>          # Use only in virtual environment!
$ pip install --user <package>   # Almost never use
```

Don't use them unless you know exactly what you are doing! The first one will try to install globally, and if you don't have permission, will install to your user site packages. In global site packages, you can get conflicting versions of libraries, you can't tell what you've installed for what, packages can update and break your system; it's a mess. And user site packages are worse, because all installs of Python on your computer share it, so you might override and break things you didn't intend to. And with pip's new smart solver, updating packages inside a global environment can take many minutes and produce unexpectedly solves that are technically "correct" but don't work because it backsolved conflicts to before issues were discovered.

There is a solution: virtual environments (libraries) or pipx (applications).

There are likely a *few* libraries (ideally just `pipx`) that you just have to install globally. Go ahead, but be careful (and always use your system package manager instead if you can, like `brew` on macOS or the Windows ones – Linux package managers tend to be too old to use for Python libraries).

2.3.1 Virtual Environments

Note: The following uses the standard library `venv` module. The `virtualenv` module can be installed from PyPI, and works identically, though is a bit faster and provides newer pip by default.

Python 3 comes with the `venv` module built-in, which supports making virtual environments. To make one, you call the module with

```
$ python3 -m venv .venv
```

This creates links to Python and pip in `.venv/bin`, and creates a site-packages directory at `.venv/lib`. You can just use `.venv/bin/python` if you want, but many users prefer to source the activation script:

```
$ . .venv/bin/activate
```

(Shell specific, but there are activation scripts for all common shells here). Now `.venv/bin` has been added to your PATH, and usually your shell's prompt will be modified to indicate you are "in" a virtual environment. You can now use `python`, `pip`, and anything you install into the virtualenv without having to prefix it with `.venv/bin/`.

Attention: Check the version of pip installed! If it's old, you might want to run `pip install -U pip` or, for modern versions of Python, you can add `--upgrade-deps` to the venv creation line.

To "leave" the virtual environment, you undo those changes by running the deactivate function the activation added to your shell:

```
deactivate
```

What about conda?

The same concerns apply to Conda. You should avoid installing things to the `base` environment, and instead make environments and use those above. Quick tips:

```
$ conda config --set auto_activate_base false  # turn off the default environment
$ conda env create -n some_name  # or use paths with `-p`
$ conda activate some_name
$ conda deactivate
```

2.3.2 Pipx

There are many Python packages that provide a command line interface and are not really intended to be imported (`pip`, for example, should not be imported). It is really inconvenient to have to set up venvs for every command line tool you want to install, however. `pipx`, from the makers of `pip`, solves this problem for you. If you `pipx install` a package, it will be created inside a new virtual environment, and just the executable scripts will be exposed in your regular shell.

Pipx also has a `pipx run <package>` command, which will download a package and run a script of the same name, and will cache the temporary environment for a week. This means you have all of PyPI at your fingertips in one line on any computer that has pipx installed!

2.3.3 Task runner (nox)

A task runner, like make (fully general), rake (Ruby general), invoke (Python general), tox (Python packages), or nox (Python simi-general), is a tool that lets you specify a set of tasks via a common interface. These can be a crutch, allowing poor packaging practices to be employed behind a custom script, and they can hide what is actually happening.

Nox has two strong points that help with this concern. First, it is very explicit, and even prints what it is doing as it operates. Unlike the older tox, it does not have any implicit assumptions built-in. Second, it has very elegant built-in support for both virtual and Conda environments. This can greatly reduce new contributor friction with your codebase.

A daily developer is not expected to use nox for simple tasks, like running tests or linting. You should not rely on nox to make a task that should be made simple and standard (like building a package) complicated. You are not expected to use nox for linting on CI, or sometimes even for testing on CI, even if those tasks are provided for users. Nox is a few seconds slower than running directly in a custom environment - but for new users and rarely run tasks, it is *much* faster than explaining how to get setup or manually messing with virtual environments. It is also highly reproducible, creating and destroying the temporary environment each time by default.

You should use nox to make it easy and simple for new contributors to run things. You should use nox to make specialized developer tasks easy. You should use nox to avoid making single-use virtual environments for docs and other rarely run tasks.

Since nox is an application, you should install it with `pipx`. If you use Homebrew, you can install `nox` with that (Homebrew isolates Python apps it distributes too, just like pipx).

2.3.4 Running nox

If you see a `noxfile.py` in a repository, that means nox is supported. You can start by checking to see what the different tasks (called `sessions` in nox) are provided by the noxfile author. For example, if we do this on `packaging.python.org`'s repository:

```
$ nox -l  # or --list-sessions
Sessions defined in /github/pypa/packaging.python.org/noxfile.py:

- translation -> Build the gettext .pot files.
- build -> Make the website.
- preview -> Make and preview the website.
- linkcheck -> Check for broken links.

sessions marked with * are selected, sessions marked with - are skipped.
```

You can see that there are several different sessions. You can run them with `-s`:

```
$ nox -s preview
```

Will build and start up a preview of the site.

If you need to pass options to the session, you can separate nox options with and the session options with `--`.

2.3.5 Writing a Noxfile

For this example, we'll need a minimal test file for pytest to run. Let's make this file in a local directory:

```
# test_nox.py

def test_runs():
    assert True
```

Let's write our own noxfile. If you are familiar with pytest, this should look familiar as well; it's intentionally rather close to pytest. We'll make a minimal session that runs pytest:

```
# noxfile.py
import nox

@nox.session()
def tests(session):
    session.install("pytest")
    session.run("pytest")
```

A noxfile is valid Python, so we import nox. The session decorator tells nox that this function is going to be a session. By default, the name will be the function name, the description will be the function docstring, it will run on the current version of Python (the one nox is using), and it will make a virtual environment each time the session runs, though all of this is changeable via keyword arguments to session.

The session function will be given a `nox.Session` object that has various useful methods. `.install` will install things with pip, and `.run` will run a command in a sesson. The `.run` command will print a warning if you use an executable outside the virtual environment unless `external=True` is passed. Errors will exit the session.

Let's expand this a little:

```
# noxfile.py
import nox

@nox.session()
def tests(session: nox.Session) -> None:
    """
    Run our tests.
    """
    session.install("pytest")
    session.run("pytest", *session.posargs)
```

This adds a type annotation to the session object, so that IDE's and type checkers can help you write the code in the function. There's a docstring, which will print out nice help text when a user lists the sessions. And we pass through to pytest anything the user passes in via `session.posargs`

Let's try running it:

```
$ nox -s tests
nox > Running session tests
nox > Creating virtual environment (virtualenv) using python3.10 in .nox/tests
nox > python -m pip install pytest
nox > pytest
==================================== test session starts␣
↪====================================
```

(continues on next page)

(continued from previous page)

```
platform darwin -- Python 3.10.5, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/henryschreiner/git/teaching/packaging
collected 1 item

test_nox.py .                                                            ␣
↪[100%]

==================================== 1 passed in 0.05s␣
↪====================================
nox > Session tests was successful.
```

Nox is really just doing the same thing we would do manually (and printing all the steps except the exact details of creating the virtual environment). You can see the virtual environment in `.nox/tests`!

Passing arguments through

Try passing `-v` to pytest.

```
$ nox -s tests -- -v
```

Virtual environments

How would you activate this environment?

```
$ source .nox/tests/bin/activate
```

In general, packages you work on daily are worth fully setting up with virtual environments, but if you are new to development or just occasionally contributing to a package, nox is a huge help.

2.4 The pyproject.toml file

Much research software is initially developed by hacking away in an interactive setting, such as in a Jupyter Notebook or a Python shell. However, at some point when you have a more-complicated workflow that you want to repeat, and/or make available to others, it makes sense to package your functions into modules and ultimately a software package that can be installed. This lesson will walk you through that process.

Consider the `rescale()` function written as an exercise in the Software Carpentry Programming with Python lesson.

First, as needed, create your virtual environment and install NumPy with

```
$ virtualenv .venv
$ source .venv/bin/activate
$ pip install numpy
```

Then, in a Python shell or Jupyter Notebook, declare the function:

```
import numpy as np

def rescale(input_array):
```

(continues on next page)

(continued from previous page)

```
    """Rescales an array from 0 to 1.

    Takes an array as input, and returns a corresponding array scaled so that 0
    corresponds to the minimum and 1 to the maximum value of the input array.
    """
    L = np.min(input_array)
    H = np.max(input_array)
    output_array = (input_array - L) / (H - L)
    return output_array
```

and call the function:

```
>>> rescale(np.linspace(0, 100, 5))
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
```

2.4.1 Creating our package in six lines

Let's create a Python package that contains this function.

First, create a new directory for your software package, called `package`, and move into that:

```
$ mkdir package
$ cd package
```

You should immediately initialize an empty Git repository in this directory; if you need a refresher on using Git for version control, check out the Software Carpentry Version Control with Git lesson. (This lesson will not explicitly remind you to commit your work after this point.)

```
$ git init
```

Next, we want to create the necessary directory structure for your package. This includes:

- a `src` directory, which will contain another directory called `rescale` for the source files of your package itself;
- a `tests` directory, which will hold tests for your package and its modules/functions (this can also go inside the `rescale` directory, but we recommend keeping it at the top level so that your test suite is not installed along with the package itself);
- a `docs` directory, which will hold the files necessary for documenting your software package.

```
$ mkdir -p src/rescale tests docs
```

(The `-p` flag tells `mkdir` to create the `src` parent directory for `rescale`.)

Putting the package directory and source code inside the `src` directory is not actually *required*; instead, if you put the `<package_name>` directory at the same level as `tests` and `docs` then you could actually import or call the package directory from that location. However, this can cause several issues, such as running tests with the local version instead of the installed version. In addition, this package structure matches that of compiled languages, and lets your package easily contain non-Python compiled code, if necessary.

Inside `src/rescale`, create the files `__init__.py` and `rescale.py`:

```
$ touch src/rescale/__init__.py src/rescale/rescale.py
```

`__init__.py` is required to import this directory as a package, and should remain empty (for now). `rescale.py` is the module inside this package that will contain the `rescale()` function; copy the contents of that function into this file. (Don't forget the NumPy import!)

The last element your package needs is a `pyproject.toml` file. Create this with

```
$ touch pyproject.toml
```

and then provide the minimally required metadata, which include information about the build system (hatchling) and the package itself (`name` and `version`):

```
# contents of pyproject.toml
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "package"
version = "0.1.0"
```

The package name given here, "package," matches the directory `package` that contains our project's code. We've chosen 0.1.0 as the starting version for this package; you'll see more in a later episode about versioning, and how to specify this without manually writing it here.

The only elements of your package truly **required** to install and import it are the `pyproject.toml`, `__init__.py`, and `rescale.py` files. At this point, your package's file structure should look like this:

```
.
├── docs
├── pyproject.toml
├── src
│   └── package
│       ├── __init__.py
│       └── rescale.py
└── tests
```

2.4.2 Installing and using your package

Now that your package has the necessary elements, you can install it into your virtual environment (which should already be active). From the top level of your project's directory, enter

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

The `-e` flag tells `pip` to install in editable mode, meaning that you can continue developing your package on your computer as you test it.

Then, in a Python shell or Jupyter Notebook, import your package and call the (single) function:

```
>>> import numpy as np
>>> from package.rescale import rescale
>>> rescale(np.linspace(0, 100, 5))
```

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

{: .output}

This matches the output we expected based on our interactive testing above!

2.4.3 Your first test

Now that we have installed our package and we have manually tested that it works, let's set up this situation as a test that can be automatically run using `nox` and `pytest`.

In the `tests` directory, create the `test_rescale.py` file:

```
touch tests/test_rescale.py
```

In this file, we need to import the package, and check that a call to the `rescale` function with our known input returns the expected output:

```
# contents of tests/test_rescale.py
import numpy as np
from package.rescale import rescale

def test_rescale():
    np.testing.assert_allclose(
        rescale(np.linspace(0, 100, 5)),
        np.array([0., 0.25, 0.5, 0.75, 1.0 ]),
        )
```

Next, take the `noxfile.py` you created in an earlier episode, and modify it to

- install `numpy`, necessary to run the package;
- install `pytest`, necessary to automatically find and run the test(s);
- install the package itself; and
- run the test(s)

with:

```
# contents of noxfile.py
import nox

@nox.session
def tests(session):
    session.install('numpy', 'pytest')
    session.install('.')
    session.run('pytest')
```

Now, with the added test file and `noxfile.py`, your package's directory structure should look like:

```
.
├── docs
├── noxfile.py
├── pyproject.toml
├── src
│   └── package
│       ├── __init__.py
│       └── rescale.py
```

(continues on next page)

(continued from previous page)

```
└── tests
    └── test_rescale.py
```

(You may also see some `__pycache__` directories, which contain compiled Python bytecode that was generated when calling your package.)

Have `nox` run your tests. This should give you some information about what `nox` is doing, and show output along the lines of

```
$ nox
nox > Running session tests
nox > Creating virtual environment (virtualenv) using python in .nox/tests
nox > python -m pip install numpy pytest
nox > python -m pip install .
nox > pytest
============================================================================ test session␣
↪starts =============================================
platform darwin -- Python 3.9.13, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/niemeyek/Desktop/rescale
collected 1 item

tests/test_rescale.py .                                                                 ␣
↪                                                [100%]

============================================================================ 1 passed in 0.
↪07s =============================================
nox > Session tests was successful.
```

This tells us that the output of the test function matches the expected result, and therefore the test passes!

We now have a package that is installed, can be interacted with properly, and has a passing test. Next, we'll look at other files that should be included with your package.

2.4.4 Informational metadata

We left the metadata in our `project.toml` quite minimal; we just had a name and a version. There are quite a few other fields that can really help your package on PyPI, however. We'll look at them, split into categories: Informational (like author, description) and Functional (like requirements). There's also a special `dynamic` field that lets you list values that are going to come from some other source.

Name

Required. `.`, `-`, and `_` are all equivalent characters, and may be normalized to `_`. Case is unimportant. This is the only field that must exist statically in this table.

```
name = "some_project"
```

Version

Required. Many backends provide ways to read this from a file or from a version control system, so in those cases you would add `"version"` to the `dynamic` field and leave it off here.

```
version = "1.2.3"
version = "0.2.1b1"
```

Description

A string with a short description of your project.

```
description = "This is a very short summary of a very cool project."
```

Readme

The name of the readme. Most of the time this is `README.md` or `README.rst`, though there is a more complex mechanism if a user really desires to embed the readme into your `pyproject.toml` file (you don't).

```
readme = "README.md"
readme = "README.rst"
```

Authors and maintainers

This is a list of authors (or maintainers) as (usually inline) tables. A TOML table is very much like a Python dict.

```
authors = [
    {name="Me Myself", email="email@mail.com"},
    {name="You Yourself", email="email2@mail.com"},
]
maintainers = [
    {name="It Itself", email="email3@mail.com"},
]
```

Note that TOML supports two ways two write tables and two ways to write arrays, so you might see this in a different form, but it should be recognizable.

Keywords

A list of keywords for the project. This is mostly used to improve searchability.

```
keywords = ["example", "tutorial"]
```

URLs

A set of links to help users find various things for your code; some common ones are `Homepage`, `Source Code`, `Documentation`, `Bug Tracker`, `Changelog`, `Discussions`, and `Chat`. It's a free-form name, though many common names get recognized and have nice icons on PyPI.

```
# Inline form
urls.Homepage = "https://pypi.org"
urls."Source Code" = "https://pypi.org"

# Sectional form
[project.urls]
Homepage = "https://pypi.org"
"Source Code" = "https://pypi.org"
```

Classifiers

This is a collection of classifiers as listed at https://pypi.org/classifiers/. You select the classifiers that match your projects from there. Usually, this includes a "Development Status" to tell users how stable you think your project is, and a few things like "Intended Audience" and "Topic" to help with search engines. There are some important ones though: the "License" (s) is used to indicate your license. You also can give an idea of supported Python versions, Python implementations, and "Operating System"s as well. If you have statically typed Python code, you can tell users about that, too.

```
classifiers = [
    "Development Status :: 5 - Production/Stable",
    "Intended Audience :: Developers",
    "Intended Audience :: Science/Research",
    "License :: OSI Approved :: BSD License",
    "Operating System :: OS Independent",
    "Programming Language :: Python",
    "Programming Language :: Python :: 3",
    "Programming Language :: Python :: 3 :: Only",
    "Programming Language :: Python :: 3.8",
    "Programming Language :: Python :: 3.9",
    "Programming Language :: Python :: 3.10",
    "Programming Language :: Python :: 3.11",
    "Topic :: Scientific/Engineering",
    "Topic :: Scientific/Engineering :: Information Analysis",
    "Topic :: Scientific/Engineering :: Mathematics",
    "Topic :: Scientific/Engineering :: Physics",
    "Typing :: Typed",
]
```

License (special mention)

There also is a license field, but that was rather inadequate; it didn't support multiple licenses, for example. Currently, it's best to indicate the license with a Trove Classifier, and make sure your file is called `LICENSE*` so build backends pick it up and include it in SDist and wheels. There's work on standardizing an update to the format in the future. You can manually specify a license file if you want:

```
license = {file = "LICENSE"}
```

Verify file contents

Always verify the contents of your SDist and Wheel(s) manually to make sure the license file is included.

```
tar -tvf dist/package-0.0.1.tar.gz
unzip -l dist/package-0.0.1-py3-none-any.whl
```

2.4.5 Functional metadata

The remaining fields actually change the usage of the package.

Requires-Python

This is an important and sometimes misunderstood field. It looks like this:

```
requires-python = ">=3.7"
```

Pip will see if the current version of Python it's installing for passes this expression. If it doesn't, pip will start checking older versions of the package until it finds on that passes. This is how `pip install numpy` still works on Python 3.7, even though NumPy has already dropped support for it.

You need to make sure you always have this and it stays accurate, since you can't edit metadata after releasing - you can only yank or delete release(s) and try again.

Upper caps

Upper caps are generally discouraged in the Python ecosystem, but they are (even more that usual) broken here, since this field was added to help users drop old Python versions, and the idea it would be used to restrict newer versions was not considered. The above procedures is not the right one for an upper cap! Never upper cap this and instead use Trove Classifiers to tell users what versions of Python your code was tested with.

Dependencies

Your package likely will need other packages from PyPI to run.

```
dependencies = [
  "numpy>=1.18",
]
```

You can list dependencies here without minimum versions, but if you have a lot of users, you might want minimum versions; pip will only upgrade an installed package if it's no longer viable via your requirements. You can also use a variety of markers to specify operating system specific packages.

---

project.dependencies vs. build-system.requires

What is the difference between `project.dependencies` vs. `build-system.requires`?

`build-system.requires` describes what your project needs to "build", that is, produce an SDist or wheel. Installing a built wheel will *not* install anything from `build-system.requires`, in fact, the `pyproject.toml` is not even present in the wheel! `project.dependencies`, on the other hand, is added to the wheel metadata, and pip will install anything in that field if not already present when installing your wheel.

Optional Dependencies

Sometimes you have dependencies that are only needed some of the time. These can be specified as optional dependencies. Unlike normal dependencies, these are specified in a table, with the key being the option you pass to pip to install it. For example:

```
[project.optional-dependenices]
test = ["pytest>=6"]
check = ["flake8"]
plot = ["matplotlib"]
```

Now, you can run `pip install 'package[test,check]'`, and pip will install both the required and optional dependencies `pytest` and `flake8`, but not `matplotlib`.

Entry Points

A Python package can have entry points. There are three kinds: command-line entry points (`scripts`), graphical entry points (`gui-scripts`), and general entry points (`entry-points`). As an example, let's say you have a `main()` function inside `__main__.py` that you want to run to create a command `project-cli`. You'd write:

```
[project.scripts]
project-cli = "project.__main__:main"
```

The command line name is the table key, and the form of the entry point is `package.module:function`. Now, when you install your package, you'll be able to type `project-cli` on the command line and it will run your Python function.

2.4.6 Dynamic

Any field from above that are specified by your build backend instead should be listed in the special `dynamic` field. For example, if you want `hatchling` to read `__version__.py` from `src/package/__init__.py`:

```
[project]
name = "package"
dynamic = ["version"]

[tool.hatch]
version.path = "src/package/__init__.py"
```

2.4.7 All together

Now let's take our previous example and expand it with more information. Here's an example:

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "package"
version = "0.0.1"
authors = [
  { name="Example Author", email="author@example.com" },
]
description = "A small example package"
readme = "README.md"
license = { file="LICENSE" }
requires-python = ">=3.7"
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: MIT License",
    "Operating System :: OS Independent",
]

[project.urls]
"Homepage" = "https://github.com/pypa/sampleproject"
"Bug Tracker" = "https://github.com/pypa/sampleproject/issues"
```

2.5 Tools for building and uploading

2.5.1 Core tools

Pipx

build

twine: the secure way to upload to PyPI

2.5.2 For consolidated experience & dependency management

Pdm

Hatch

2.5.3 Building a source distribution

2.5.4 Building a wheel

2.5.5 Discuss use of delocate/Auditwheel/...

2.5.6 Difference between linux & manylinux wheels

2.6 Binary extensions

2.6.1 Scikit-build overview & motivation

2.7 Continuous Integration

Continuous Integration (CI) allows you to perform tasks on a server for various events on your repository (called triggers). For example, you can use GitHub Actions (GHA) to run a test suite on every pull request.

2.7.1 GitHub Actions

GHA is made up of workflows which consist of actions. Workflows are files in the `.github/workflows` folder ending in `.yml`.

Triggers

Workflows start with triggers, which define when things run. Here are three triggers:

```
on:
  pull_request:
  push:
    branches:
      - main
```

This will run on all pull requests and pushes to main. You can also specify specific branches for pull requests instead of running on all PRs (will run on PRs targeting those branches only).

Running unit tests

Let's set up a basic test. We will define a jobs dict, with a single job named "tests". For all jobs, you need to select an image to run on - there are images for Linux, macOS, and Windows. We'll use `ubuntu-latest`.

```
jobs:
  tests:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
```

(continues on next page)

(continued from previous page)

```
      - uses: actions/setup-python@v4
        with:
          python-version: "3.10"

      - name: Install package
        run: python -m pip install -e .[test]

      - name: Test package
        run: python -m pytest
```

This has five steps:

1. Checkout the source (your repo).
2. Prepare Python 3.10 (will use a preinstalled version if possible, otherwise will download a binary).
3. Install your package with testing extras - this is just an image that will be removed at the end of the run, so "global" installs are fine. We also provide a nice name for the step.
4. Run your package's tests.

By default, if any step fails, the run immediately quits and fails.

Running in a matrix

You can parametrize values, such as Python version or operating system. Do do this, make a `strategy: matrix:` dict. Every key in that dict (except `include:` and `exclude` should be set with a list, and a job will be generated with every possible combination of values. You can access these values via the `matrix` variable; they do not "automatically" change anything.

For example:

```
example:
  strategy:
    matrix:
      onetwothree: [1, 2, 3]
  name: Job ${{ matrix.onetwothree }}
```

would produce three jobs, with names `Job 1`, `Job 2`, and `Job 3`. Elsewhere, if you refer to the `exmaple` job, it will implicitly refer to all three.

This is commonly used to set Python and operating system versions:

```
tests:
  strategy:
    fail-fast: false
    matrix:
      python-version: ["3.7", "3.11"]
      runs-on: [ubuntu-latest, windows-latest, macos-latest]
  name: Check Python ${{ matrix.python-version }} on ${{ matrix.runs-on }}
  runs-on: ${{ matrix.runs-on }}
  steps:
    - uses: actions/checkout@v3
      with:
        fetch-depth: 0 # Only needed if using setuptools-scm
```

(continues on next page)

(continued from previous page)

```
    - name: Setup Python ${{ matrix.python-version }}
      uses: actions/setup-python@v4
      with:
        python-version: ${{ matrix.python-version }}

    - name: Install package
      run: python -m pip install -e .[test]

    - name: Test package
      run: python -m pytest
```

There are two special keys: `include:` will take a list of jobs to include one at a time. For example, you could add Python 3.9 on Linux (but not the others):

```
include:
  - python-version: 3.9
    runs-on: ubuntu-latest
```

`include` can also list more keys than were present in the original parametrization; this will add a key to an existing job.

The `exclude:` key does the opposite, and lets you remove jobs from the matrix.

Other actions

GitHub Actions has the concept of actions, which are just GitHub repositories of the form `org/name@tag`, and there are lots of useful actions to choose from (and you can write your own by composing other actions, or you can also create them with JavaScript or Dockerfiles). Here are a few:

There are some GitHub supplied ones:

- actions/checkout: Almost always the first action. v2+ does not keep Git history unless `with: fetch-depth: 0` is included (important for SCM versioning). v1 works on very old docker images.
- actions/setup-python: Do not use v1; v2+ can setup any Python, including uninstalled ones and pre-releases. v4 requires a Python version to be selected.
- actions/cache: Can store files and restore them on future runs, with a settable key.
- actions/upload-artifact: Upload a file to be accessed from the UI or from a later job.
- actions/download-artifact: Download a file that was previously uploaded, often for releasing. Match upload-artifact version.

And many other useful ones:

- ilammy/msvc-dev-cmd: Setup MSVC compilers.
- jwlawson/actions-setup-cmake: Setup any version of CMake on almost any image.
- wntrblm/nox: Setup all versions of Python and provide nox.
- pypa/gh-action-pypi-publish: Publish Python packages to PyPI.
- pre-commit/action: Run pre-commit with built-in caching.
- conda-incubator/setup-miniconda: Setup conda or mamba on GitHub Actions.
- peaceiris/actions-gh-pages: Deploy built files to to GitHub Pages

- ruby/setup-miniconda Setup Ruby if you need it for something.

2.7.2 Pre-commit

2.7.3 Building wheels with cibuildwheel

2.7.4 Exercise

Add a CI file for your package.

2.8 Handling Dependencies

2.8.1 "In-project" compilation (pybind11)

2.8.2 External

See https://github.com/pypa/cibuildwheel/issues/1251#issuecomment-1236364876 for example

2.9 Tutorial Content Updates

You will find here the list of changes integrated in the tutorial after it was first given at the SciPy 2018 conference.

Changes are grouped in sections identified using `YYYY-MM` representing the year and month when the related changes were done.

The sections are ordered from most recent to the oldest.

2.9.1 2032-02

Started rewrite for modern packaging.

2.9.2 2018-08

Better handling data file in `setup_py_exercise_small_example_package` section

- Put package data in `data` directory.
- Reflect this change in the code.
- Add `package_data` to setup function.

2.9.3 2018-07

This is the first set of changes incorporating the feedback from attendees.

Making a Python Package

- Add directory setup_example/capitalize discussed in `setup_py_exercise_small_example_package` section.

Building and Uploading to PyPI

- Update `Installing a wheel` tutorial adding `Install a package from TestPyPI <install_wheel_from_testpypi>` section.

CHAPTER

THREE

YOUR GUIDES

Henry Schreiner
Maintainer of scikit-build, scikit-hep, cibuildwheel, build, meson-python, python-metadata, and other packages.

Matt McCormick:
Maintainer of dockcross, of Python packages for the Insight Toolkit (ITK)

Jean-Christophe Fillion-Robin:
Maintainer of scikit-build, scikit-ci, scikit-ci-addons and python-cmake-buildsystem