



UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386

# Python Packaging

Liam Keegan, SSC

# Motivation

It would be reasonable to assume that packaging Python code should be a straightforward, well-defined, user-friendly process, much like writing Python code…

…unfortunately this is not really the case, and some googling will provide a multitude of conflicting advice on how to do this.

The Python Packaging User Guide (*packaging.python.org*) itself begins with this slightly discouraging note:

> ***"Building your understanding of Python packaging is a journey. Patience and continuous improvement are key to success."***

Our goal today is to present a practical and modern approach to packaging and distributing your Python code.

# Course Outline

- Packaging overview
- Hands on: create a Python package
  - source file layout
  - pyproject.toml to allow local install using pip
- Hands on: publish to PyPI
  - add dependencies
  - add metadata
  - publish package to testPyPI
- Next steps
  - automated PyPI deployment using GitHub Actions
  - conda-forge recipe
- Compiled extension modules
  - pybind11 / nanobind c++ libraries
  - scikit-build-core for CMake integration
  - cibuildwheel to automate building of binary wheels

# Course slides and code

You will find slides and code samples for this course at:

ssciwr.github.io/python-packaging

It will be helpful to keep this open in a browser tab during the course.

This page also contains

- Links to example repositories
- Links to recommended resources
- Links to cookiecutters to create your own projects


# Packaging Overview

# What is a Python module?

A Python module is something that we can import in Python, typically a single Python source file (excluding the .py filename extension)

```
# stats.py

import random


def flip_coin():
    return random.choice(["Heads", "Tails"])
```

```
Python 3.12.1
>>> import stats
>>> stats.flip_coin()
'Tails'
```

If you have many modules it is often convenient to organise them in a hierarchical way, which can be done using a Python package.

# What is a Python package?

A Python package is simply a folder that contains Python source code files, including one with the special name `__init__.py`. When you import a package, the contents of `__init__.py` (if any) are automatically imported.

calculate/

```
# stats.py

import random
def flip_coin():
    return random.choice(["Heads", "Tails"])
```

```
# __init__.py

from .stats import flip_coin
```

```
Python 3.12.1
>>> import calculate
>>> calculate.flip_coin()
'Heads'
```

# What else is required?

In addition to providing Python code, to make a package usable by others we typically need to provide a lot of additional information, such as:

- Do any other Python libraries need to be installed?
- Is there code that needs to be compiled?
- Which Python versions and operating systems are supported?
- Test code and how to run it
- Documentation
- License
- Authors / maintainers
- Entry points / scripts
- ...

# How to convey this information?

How can we provide this information to the user so they can use the package? Some (not recommended) options include:

- Don’t! Just let the user figure it out by trial and error
- Provide a README with instructions or a list of dependencies
- Supply a requirements.txt file with a list of dependencies for the user to install
- …

All of these are inconvenient and error-prone for the user, and it would be much nicer if we could instead provide this information to a tool which the user could use to take of all this automatically…

# Where do our packages come from?

We typically install our packages from a package index / distribution / repository:

- `pip install numpy`
  - Install numpy from PyPI (Python Package Index) using pip
- `conda install numpy -c conda-forge`
  - Install numpy from the conda-forge conda channel using conda
- `brew install numpy`
  - Install numpy from homebrew using brew
- `...`

In all of these cases, the package manager (pip/conda/brew) performs these steps:

- First install any libraries that the package requires
- Then build any compiled modules (or install pre-built if available)
- Finally, install the Python code for the package

# Metadata and build recipes

So we need to tell the package manager how to build and install our package. Each package manager has a different way of doing this:

- `pip`
  - pyproject.toml file
- `conda`
  - meta.yaml recipe
- `brew`
  - formula or cask
- ...

Each of these also has a different process to submit, build and publish packages.

# Packaging Overview

To package our Python code we need to provide two things

- The Python package itself (aka “Import package”)
  - The Python source code / modules / package that the user will import
- Metadata / build / install instructions
  - What are the dependencies of our package
  - How to compile / build / install it
  - Each package manager has a different way of specifying this

Combining these allows us to publish a “Distribution package” to a repository such as such as PyPI or conda-forge, to allow users to easily install and use it.


# A Minimal Python Package

# calculate

We'll now create a bare-bones Python package "calculate" from scratch.

Recommended: create a new conda environment to work in:

```
conda env create -n packaging python
conda activate packaging
```

A minimal version of this package is available here:

https://github.com/ssciwr/python-packaging/tree/main/calculate-minimal

# Recommended project structure: src layout

```
/calculate
    /src
        /calculate
            __init__.py
            stats.py
            ...
    pyproject.toml
```

Any code that should be importable goes here - this is the Python package

Defines how to install the package

See https://packaging.python.org/en/latest/discussions/src-layout-vs-flat-layout for more information

# Python code

Our Python package consists of these two .py files:

src/calculate/

```python
# stats.py

import random


def flip_coin():
    return random.choice(["Heads", "Tails"])
```

```python
# __init__.py

from .stats import flip_coin
```

# pyproject.toml

This is a configuration file in TOML format that contains three sections:

- `[build-system]`
  - Specify the build backend to build & install your package
  - Also any required build-time dependencies
- `[project]`
  - Specify project metadata such as name, version and authors
  - Also any required run-time dependencies
- `[tool]`
  - Settings for tools such as cibuildwheel
  - We won't need this section for now

# pyproject.toml: build-system

Specifies what build system should be used to build and install your package, and any packages that are required to be installed to do this.

Recommended default for pure Python packages:

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

Alternative build-systems include setuptools, flit, PDM, ...

They all offer the same basic functionality, differ only in advanced features.

# pyproject.toml: project

Specifies project metadata and dependencies.

Minimal required contents are a project name and version number:

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

# pyproject.toml

A minimal complete pyproject.toml for our package:

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


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

# Bare-bones package

We now have a very minimal but installable Python package.

https://github.com/ssciwr/python-packaging/tree/main/calculate-minimal

From the top level directory (that contains pyproject.toml) we can now install the package using with pip with the command

```
pip install .
```

Here pip is the “build frontend” tool that calls the “build backend” tool we specified in pyproject.toml to build and install the package.

# Bare-bones package

We should now be able to import and use our package in Python

```
Python 3.12.1
>>> import calculate
>>> calculate.flip_coin()
'Heads'
>>> calculate.__file__
'/home/lkeegan/micromamba/envs/python-packaging/lib/python3.12/site-p
ackages/calculate/__init__.py'
```

# Sidenote: editable install

When we `import calculate` in Python, we now get the installed package.

While we are developing the code though, we might want to make a change to the code and then import the modified package without having to install it again with each change.

The way to do this conveniently is to do an “editable” install with -e:

```
pip install -e .
```

This installs links to your source files instead of copying them, so that any changes you make are also applied to the installed package.


# PyPI: Python Package Index

# What is PyPI?

The Python Package Index (PyPI) is a repository of Python software.

Instead of the user downloading your package files and then installing it, you can publish your package on PyPI, then the user can directly install it from there:

```
pip install calculate
```

To publish our package to PyPI, we need to extend pyproject.toml with more metadata, then build and publish our package on PyPI.

We'll use the TestPyPI index today, but the real one works in exactly the same way.

# A note on package naming

***Python*** package names must consist of lower case ASCII letters, digits, and underscores "_", for example:

- `my_package`

***PyPI*** package names are also allowed to use hyphens "-" and periods "." as separators, repeat separators, and are case-insensitive. So all of these are equivalent PyPI names:

- `my_package`
- `my-package`
- `My-Package`
- `my.PaCkage`
- `my._-.PacKaGe`
- …

# A note on package naming

It is also allowed for the ***PyPI*** package name to differ from the ***Python*** package name.

In general this is not recommended as it can lead to confusion, especially if your python package name also exists as a (different) pypi package!

Notable exception: forks of unmaintained projects that want to be a drop-in replacement for the original project, such as pillow (PIL fork):

```
pip install pillow
```

```
import PIL
```

# Recommended naming convention

Recommendation (most commonly used convention):

- Use lower case letters, digits and underscores for your package name
- Use the same name on PyPI but replace the underscores with hyphens

Example:

- Python package name: **`my_package`**
- PyPI package name: **`my-package`**

Tip: check PyPI (and conda-forge if relevant) before choosing a name for your package - many are already taken!

# Choose a name for our package

- We need a name that isn't already in use
- Suggestion: append your name, e.g.
  - PyPI name: `calculate-name`
  - Python package name: `calculate_liam`
- Check it is not taken on test.pypi.org
- Update your folder names, code and pyproject.toml accordingly
- Check that installing and importing the package still works

# Adding a dependency

Let's add some new functionality that uses numpy

```python
# stats.py

import random
import numpy as np

def roll_dice(n_dice: int, n_sides: int):
    return np.random.randint(1, n_sides, n_dice)

def flip_coin():
    return random.choice(["Heads", "Tails"])
```

```python
# __init__.py

from .stats import flip_coin
from .stats import roll_dice
```

# Dependencies in pyproject.toml

Any packages listed in dependencies will automatically be installed when your package is installed:

```
[project]
...
dependencies = ["numpy"]
...
```

You can also add more specific version/os/etc requirements here, e.g.

```
dependencies = ["numpy >= 1.16, < 2.0.0"]
```

Much more information about this here:

packaging.python.org/en/latest/specifications/dependency-specifiers/#dependency-specifiers

# Installation dependencies

If you did an editable install of your package, you can already try to use the new `roll_dice` function without installing the package again, but you may get a ModuleNotFoundError:

```
ModuleNotFoundError: No module named 'numpy'
```

This is because numpy was not a dependency when you installed your package. Whenever you add a dependency to your pyproject.toml, you need to install your package again to install the new dependencies (even for an editable install):

```
pip install -e .
```

# Recommended project structure: src layout

```
/calculate-liam
    /src
        /calculate_liam
            __init__.py
            stats.py
            ...
    pyproject.toml
```

Any code that should be importable goes here - this is the Python package

Defines how to install the package

See https://packaging.python.org/en/latest/discussions/src-layout-vs-flat-layout for more information

# Recommended project structure: src layout

```
/calculate-liam
    /src
        /calculate_liam
            __init__.py
            stats.py
            ...
    /tests
        __init__.py
        test_stats.py
        ...
    /docs
        ...
    pyproject.toml
    README.md
    LICENSE
```

Any code that should be importable goes here - this is the Python package

The test suite and documentation are part of the project, but not part of the Python package

Defines how to install the package

All projects should have a readme

All projects should have a license

See https://packaging.python.org/en/latest/discussions/src-layout-vs-flat-layout for more information

# Adding an optional dependency

Let’s add a unit test that uses the pytest library

```
# ../tests/test_stats.py

def test_flip_coin():
    coin = stats.flip_coin()
    assert coin in ["Heads", "Tails"]
```

We don’t want to make this a dependency of our package, as most users presumably won’t want to run our tests. But it would be good to specify what additional dependencies are required to run the tests.

# Optional dependencies in pyproject.toml

Optional dependencies have a name and a list of dependencies

```
[project.optional-dependencies]

test = ["pytest"]
```

At install time, optional dependencies can be added using square brackets

```
pip install -e .[test]
```

This does an editable install using the pyproject.toml from the current directory, and also installs the optional “test” dependencies

# Adding a command line interface

Let's add a CLI using the click library, so the user can flip a coin from the command line:

```
# cli.py

import click
from .stats import flip_coin

@click.command()
def flip_coin_cli():
    click.echo(flip_coin())
```

# Adding a script to pyproject.toml

Script entry points can be specified as “package.module:function”, e.g.

```
[project.scripts]
flip-coin = "calculate_liam.cli:flip_coin_cli"
```

This will install a `flip-coin` command which will call the `flip_coin_cli` function:

```
$ flip-coin
Heads
```

Note that we also need to add the “click” library to our list of dependencies.

# Metadata in pyproject.toml

Finally we add some project metadata that will be displayed by PyPI

```
[project]
...
authors = [
  { name="Liam Keegan", email="liam@keegan.ch" },
]
description = "A simple package to flip a coin or roll dice"
readme = "README.md"
license = { text = "MIT"}

classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: MIT License",
    "Operating System :: OS Independent",
]
...
```

# Complete package

We now have a more complete package which is ready to publish:

https://github.com/ssciwr/python-packaging/tree/main/calculate-liam

- src/calculate_liam/
  - __init__.py
  - cli.py
  - stats.py
- test/
  - __init__.py
  - test_stats.py
- LICENSE.md
- README.md
- pyproject.toml

# build

First make sure we have up-to-date versions of pip, build and twine:

```
pip install --upgrade pip build twine
```

Now run build in the directory that contains pyproject.toml

```
python -m build
```

build is a frontend build tool, like pip, but instead of installing your package it should generate a compressed “source distribution” as well as one or more “build distribution” wheel files in a directory named `dist/`.

# Wheel and source distributions

The `.tar.gz` file is a “source distribution” - if you decompress it you will find all the files from your project, and you can then do `pip install .` to install your package from these files.

The `.whl` file is a “built distribution” or Wheel - essentially a zip file containing only the files that need to be installed to use your package. In general these wheels can also contain pre-compiled extension modules, and in this case there may be a separate wheel for each Python version and operating system.

Both these distributions get uploaded to PyPI, and pip install will always prefer to install from a wheel, only falling back to the source distribution if no suitable wheel exists.

# PyPI account and API key

Make an account on testPyPI

- https://test.pypi.org/account/register

Generate an API key

- https://test.pypi.org/manage/account/#api-tokens
- Scope: Entire Account
- Copy / save the token somewhere - it won't be displayed again!

# Upload to (test)PyPI with twine

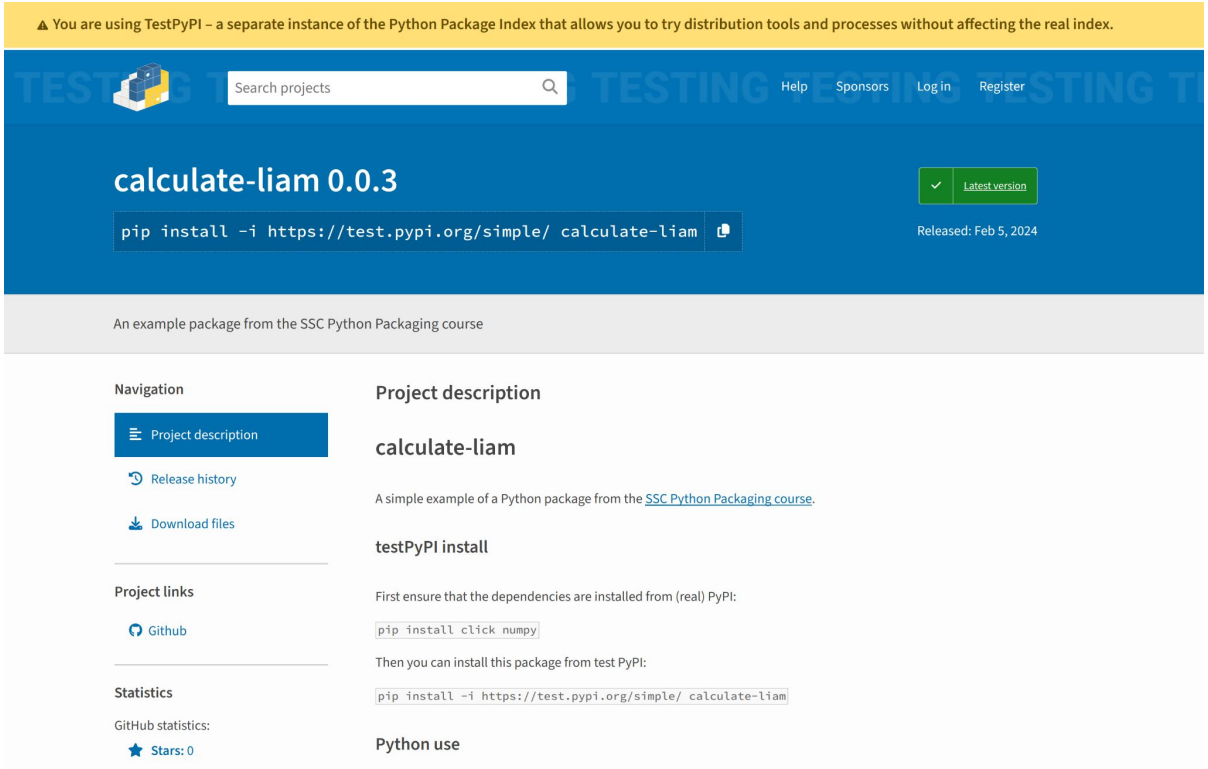
To upload the source and built distributions that were generated in the dist folder to the testPyPI repository:

```
python -m twine upload --repository testpypi dist/*
```

When prompted enter `__token__` for the user and your API key (which should start with “pypi-”) for the password:

- Username: `__token__`
- Password: `pypi-abc123etc`

# Your package on (test)PyPI




test.pypi.org/project/calculate-liam

# Install your package from (test)PyPI

To install the package from testPyPI, use pip and specify the index url:

```
pip install -i https://test.pypi.org/simple/ calculate-liam
```

Note that any dependencies your package has will also be installed from testPyPI, which will likely fail.

The workaround is to first install them manually from the real PyPI using pip, before installing your package from testPyPI:

```
pip install click numpy pytest
```

# Real PyPI publishing

Publishing to the real PyPI index works in exactly the same way, the only difference is to not specify the test.pypi.org repository, and use an API key from your PyPI account:

```
python -m build
python -m twine upload dist/*
```

Pip uses the PyPI index by default, so installing a package that you publish on PyPI works just like installing any other package on PyPI:

```
pip install calculate-liam
```


# Automated PyPI publishing

# GitHub actions

Instead of manually uploading the built files to PyPI, many projects use CI (continuous integration) to do this automatically, typically whenever a commit is tagged with a version number

Automatic publishing using CI in one slide:

- Actions are scripts that run automatically when code is pushed to github
- A yaml file specifies when and which actions should run
- They are mostly used for running automated tests, code linting, etc
- They can also be used for code publishing / deployment
- There is a specific action for publishing to PyPI

# GitHub action to publish to PyPI

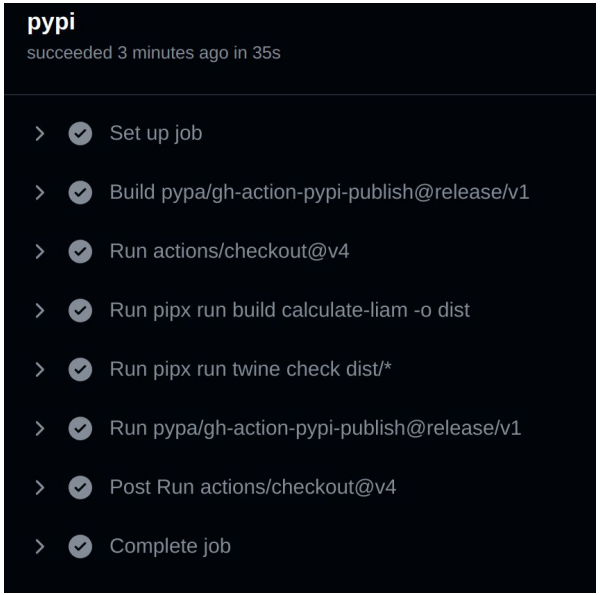
```yaml
name: PyPI publishing
on: push
jobs:
  pypi:
    runs-on: ubuntu-latest
    environment: release
    permissions:
      id-token: write
    steps:
      - uses: actions/checkout@v4
      - run: pipx run build calculate-liam -o dist
      - run: pipx run twine check dist/*
      - if: github.event_name == 'push' && startsWith(github.event.ref, 'refs/tags/')
        uses: pypa/gh-action-pypi-publish@release/v1
        with:
          repository-url: https://test.pypi.org/legacy/
          verbose: true
```

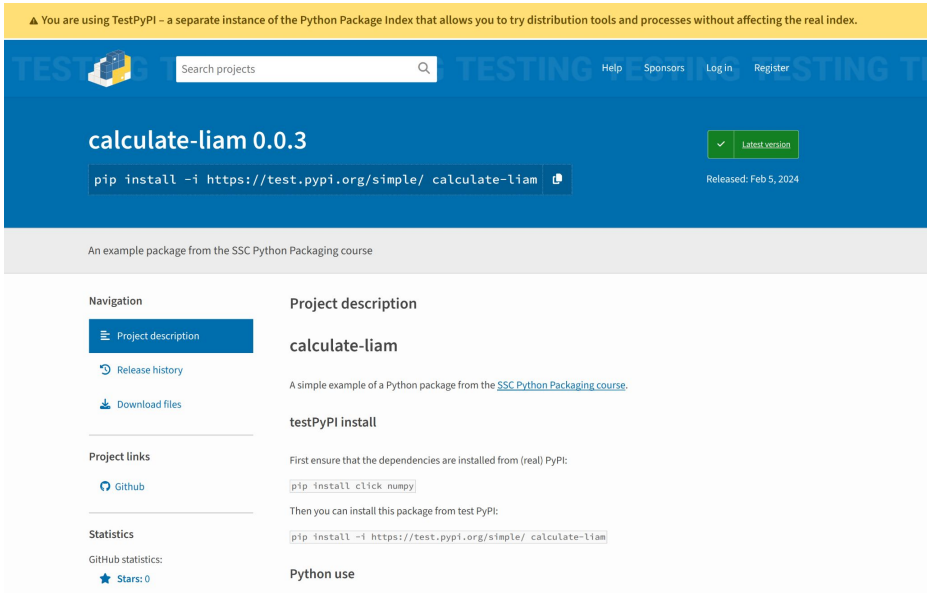
# PyPI trusted publishing

- Setup trusted publishing on PyPI
- Define a trusted github owner/repo/action
- Use pypa/gh-action-pypi-publish action
- This action can then publish to PyPI
- Avoids needing an API key

# GitHub action example

https://github.com/ssciwr/python-packaging/tree/main/calculate-liam




# Conda-forge

# Conda-forge

Conda is a package manager (like pip), but not just for Python packages.

There are different channels (i.e. collections of packages) available on conda, such as

- anaconda
- conda-forge
- bioconda
- nvidia
- etc

Conda-forge is a community effort and also the most commonly used channel

# Conda-forge dependencies

Pure Python dependencies work similarly on PyPI and on conda-forge, you provide a list of packages that your package depends on.

An important detail is that all these packages need to be available on conda-forge - it is not allowed to install a dependency from PyPI!

For most dependencies this is not an issue as conda-forge contains the vast majority of commonly used packages - but if you have a dependency that isn't available on conda-forge you will first need ensure that package is published on conda-forge (possibly by doing this yourself!) before you can publish your package.

# Conda-forge recipe 1/3

Each package has a recipe which defines all the required metadata for the package. Here is the meta.yaml recipe for our `calculate-liam` package:

```
{% set name = "calculate-liam" %}
{% set version = "0.0.3" %}

package:
  name: {{ name|lower }}
  version: {{ version }}

source:
  url: https://pypi.io/packages/source/{{ name[0] }}/{{ name }}/calculate_liam-{{ version
}}.tar.gz
  sha256: ad9fe95227ecae2e8d9c1151c5efd5367f4f3c9178670a7bd85a21e1ec300656
```

# Conda-forge recipe 2/3

```
build:
  entry_points:
    - flip-coin = calculate_liam.cli:flip_coin_cli
    - roll-dice = calculate_liam.cli:roll_dice_cli
  noarch: python
  script: {{ PYTHON }} -m pip install . -vv --no-deps --no-build-isolation
  number: 0

requirements:
  host:
    - python >=3.8
    - hatchling
    - pip
  run:
    - python >=3.8
    - click
    - numpy
```

# Conda-forge recipe 3/3

```yaml
test:
  imports:
    - calculate_liam
  commands:
    - pip check
    - flip-coin --help
    - roll-dice --help
  requires:
    - pip

about:
  summary: An example package from the SSC Python Packaging course
  license: MIT
  license_file: LICENSE.md

extra:
  recipe-maintainers:
    - lkeegan
```

# Automatically generated recipes

Most of this metadata is already in our pyproject.toml, and in fact there is a tool called `grayskull` which can automatically generate a conda-forge recipe for a package that is already published on PyPI:

```
pip install grayskull
grayskull pypi --strict-conda-forge package_name
```

For a package on testPyPI we also need to specify the testPyPI url:

```
grayskull pypi --pypi-url https://test.pypi.org/pypi calculate-liam
```

# Minimal Workflow

To submit a new package to conda-forge:

- Fork & clone github.com/conda-forge/staged-recipes and make a branch
  - `gh repo fork conda-forge/staged-recipes --clone`
  - `cd staged-recipes`
  - `git checkout -b calculate_liam`
- Run grayskull from the staged-recipes/recipes folder
  - `cd recipes`
  - `grayskull pypi --strict-conda-forge --pypi-url https://test.pypi.org/pypi calculate-liam`
- Check/modify the generated meta.yaml (e.g add tests)
- Check the build works locally
  - `cd .. && python build-locally.py`
- Make a Pull Request on GitHub


# Compiled extension modules

# Compiled extension module

We started by saying that a Python module is something you can import, typically some python source code.

The other kind of module you can import is a compiled extension module, that was written in a compiled language such as C, C++ or Fortran.

This is most commonly used for performance reasons - in most situations a compiled language can offer much better performance than an interpreted language like Python.

The other main use case is to provide a Python interface to an existing c++ library.

# Challenges

Compared to writing a Python module, things are more complicated:

- Need to write compiled code that interfaces with the Python C interface
- Need to integrate CMake and pyproject.toml build systems
- Then:
  - User downloads and compiles the code for their operating system and Python version
  - Typically the logic for this was added to setup.py and ran on package installation
  - But this requires the user to have a working compiler setup - many ways for this to go wrong
- Modern alternative:
  - Pre-compile the code into binary wheels for all combinations of operating systems and Python versions
  - This used to require an impractical amount of work
  - Nowadays it’s actually pretty easy thanks to excellent tooling

# C++ bindings

There are many different ways to write a compiled extension module:

- Use C and the Python.h C interface
  - Low level approach, not recommended!
- Use Cython
  - Write Python-like code, ok if you don't have existing C++ code to interface with
- Use a tool like SWIG
  - Auto-generates bindings, good if you have a lot of code you want to provide bindings for
- Use a C++ library like pybind11 / nanobind / (boost.python)
  - User friendly and widely used, best choice for most situations
- We'll use pybind11 for our examples

# pybind11

*“pybind11 is a lightweight header-only library that exposes C++ types in Python and vice versa, mainly to create Python bindings of existing C++ code”*

- Easy to use
- Very well documented
- Actively maintained
- Widely used
- Good performance
- See **nanobind** for a newer more efficient alternative

# pybind11

```cpp
// example.cpp

#include <pybind11/pybind11.h>

namespace py = pybind11;

float square(float x) { return x * x; }

PYBIND11_MODULE(example, m) {
    m.def("square", &square);
}
```

# CMake

```
# CMakeLists.txt

cmake_minimum_required(VERSION 3.15...3.26)
project(${SKBUILD_PROJECT_NAME} LANGUAGES CXX)

set(PYBIND11_NEWPYTHON ON)
find_package(pybind11 CONFIG REQUIRED)

pybind11_add_module(example example.cpp)

install(TARGETS example LIBRARY DESTINATION .)
```

# Scikit-build-core

*“Provides a bridge between CMake and the Python build system, allowing you to make Python modules with CMake.”*

- Easy to use
- Excellent documentation
- Actively maintained
- Successor to widely used scikit-build
- Recently developed but already stable enough for production use

# Scikit-build-core

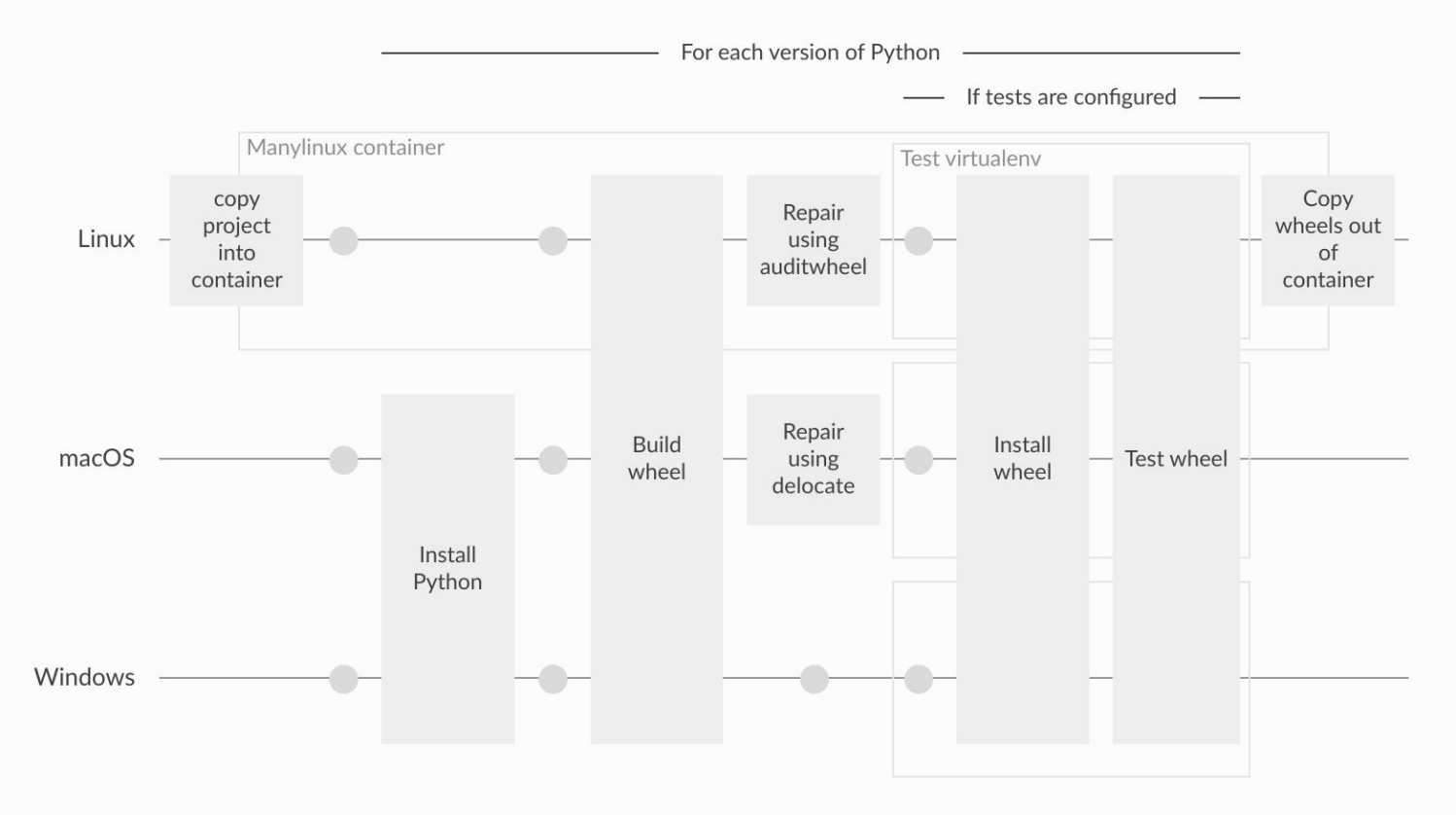
```toml
# pyproject.toml

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

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

# cibuildwheel

*“cibuildwheel runs on your CI server - currently it supports GitHub Actions, Azure Pipelines, Travis CI, AppVeyor, CircleCI, and GitLab CI - and it builds and tests your wheels across all of your platforms.”*

- Well documented
- Actively maintained
- Widely used
- Makes an impossible task trivial

# cibuildwheel



# cibuildwheel

```yaml
name: Build
on: [push, pull_request]
jobs:
  build_wheels:
    name: Build wheels on ${{ matrix.os }}
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-20.04, windows-2019, macos-11]
    steps:
      - uses: actions/checkout@v4
      - name: Build wheels
        uses: pypa/cibuildwheel@v2.16.5
      - uses: actions/upload-artifact@v4
        with:
          name: cibw-wheels-${{ matrix.os }}-${{ strategy.job-index }}
          path: ./wheelhouse/*.whl
```

# Examples

https://github.com/ssciwr/pybind11-numpy-example

- Simple example of use
- Uses pybind11 for bindings
- CMake and Scikit-build-core for the build system
- cibuildwheel and Github Actions to build wheels
- Publish wheels to PyPI

More complicated example: https://github.com/ssciwr/hammingdist


# Summary

# Summary

In this course we covered:

- Python packages
- Packaging on PyPI
- Automated PyPI deployment using GitHub Actions
- Packaging on conda-forge
- Compiled C++ extension modules

# Further resources

- Pure Python packaging
  - https://packaging.python.org/en/latest/tutorials/packaging-projects
  - https://learn.scientific-python.org/development/guides/packaging-simple
- Python packaging with compiled extensions
  - https://learn.scientific-python.org/development/guides/packaging-compiled
  - https://scikit-build-core.readthedocs.io/en/latest/getting_started.html
  - https://pypackaging-native.github.io/
- Cookiecutters to generate your own GitHub repo for a Python package
  - https://github.com/ssciwr/cookiecutter-python-package
  - https://github.com/scientific-python/cookie