

Everything* you didn't know you needed

*blatant marketing nonsense

Kilian Lieret and Henry Schreiner

Princeton University


CoDaS-HEP school 2022




Slides available as  open source, contributions welcome.

8/5/2022

Pre-commit hooks

Run small checks *before* you commit

- **!?** **Problem:** How can I stop myself from committing low-quality code?
-  **Solution:**
 - *git hooks* allow you to run scripts that are triggered by certain actions
 - a pre-commit hook is triggered every time you run `git commit`
 - in principle you can set them up yourself by placing scripts into `.git/hooks`

-  **Making it practical:**
 - The pre-commit framework is a python package that makes configuring pre-commit hooks easy!
 - All hooks are configured with a single `.pre-commit-config.yaml` file
 - Few-clicks GitHub integration available: pre-commit.ci
-  **Setting it up:**
 1. `pipx install pre-commit`
 2. `cd <your repo>`
 3. `touch .pre-commit-config.yaml`
 4. `pre-commit install`
 5. Profit 

Pre-commit hooks 📌

A config that will always be useful. Optional pre-commit.ci CI service.

```
1  repos:
2  - repo: https://github.com/pre-commit/pre-commit-hooks
3    rev: 'v4.3.0'
4    hooks:
5      - id: check-added-large-files
6      - id: check-case-conflict
7      - id: check-merge-conflict
8      - id: detect-private-key
9      - id: end-of-file-fixer
10     - id: trailing-whitespace
11
12  - repo: https://github.com/codespell-project/codespell # the spell checker with ~0 false positives
13    rev: 'v2.1.0'
14    hooks:
15      - id: codespell
16        # args: ["-I", "codespell.txt"] # Optiona, one of several ways to add exceptions
17
18  ci:
19    autoupdate_schedule: monthly # default is weekly
```

See <https://scikit-hep.org/developer/style> for many more, updated weekly!

¹or any IPython system

Pre-commit hooks for python!

```
1 - repo: https://github.com/psf/black # Reformat code without compromises!
2   rev: '22.6.0'
3   hooks:
4     - id: black
5     - id: black-jupyter
6 - repo: https://github.com/PyCQA/flake8 # Simple static checks
7   rev: '5.0.1'
8   hooks:
9     - id: flake8
10      additional_dependencies: ['flake8-bugbear']
11 - repo: https://github.com/pre-commit/mirrors-mypy # Check typing (slightly more advanced)
12   rev: 'v0.971'
13   hooks:
14     - id: mypy
15 - repo: https://github.com/asottile/pyupgrade # Automatically upgrade old Python syntax
16   rev: 'v2.37.2'
17   hooks:
18     - id: pyupgrade
19       args: [--py37-plus]
```

- **Try it out:** Go [here](#) for a quick step-by-step tutorial

Hot code reloading

- **!?** **Problem:**

1. I have some code in a notebook and some code in a python file.
2. I update my python file.
3. Do I have to restart the kernel and rerun to see the changes?

-  **Solution:** No! Python supports a number of ways to "reload" imported code.

- **Easiest example:** Add the following to your Jupyter notebook¹ to reload all (!) modules every time you execute code




```
1 %load_ext autoreload
2 %autoreload 2
```

- **More granular:**

```
1 import mymodule
2 import imp
3
4 # change mymodule
5
6 imp.reload(mymodule)
```


- **Warning:** These tricks don't *always* work and there's some additional tricks (e.g., you might need to re-run `from mymodule import X` lines)`
- **Try it out!** Follow our instructions [here](#).

Cookiecutter

- **!?** **Problem:** Setting up e.g., a python package with unit testing/CI/CD, pre-commits, license, packaging information, etc., is a lot of "scaffolding" to be added.
 -  **Solution:** Creating templates
 -  **Making it practical:** cookiecutter is a command line utility for project templates
- **Examples:**
 - scikit-hep project template: All the features, all the best-practices
 - my personal python template: Fewer options, easier to read (I think ;))
 -  **Pro-tip:** cruft is a cookiecutter extension that allows to propagate updates to the template back to the projects that use it
 - **Trying it out:**

```
1 pipx install cookiecutter
2 # alternative: cruft https://...
3 cookiecutter https://github.com/scikit-hep/cookie/
4 # e.g., select project type = setuptools
5 # for the "traditional" way to set up your python
6 # package
```

SSH Config

- **!?** **Problem:** Typing long servernames and potentially tunnelling can be tiresome
-  **Solution:** Create configuration in `~/ .ssh/config`. You can even add pattern matching!

```
1  # Server I want to connect to
2  Host tiger*
3      Hostname tiger.princeton.edu
4      User kl5675
5
6  # Tunnel that I might use sometimes
7  Host tigressgateway
8      Hostname tigressgateway.princeton.edu
9      User kl5675
10
11 Host *-t
12     ProxyCommand ssh tigressgateway -W %h:%p
```

Now you can use `ssh tiger` or `ssh tiger-t` depending on whether to tunnel or not.

SSH Escape Sequences

- **!?** **Problem:** I already have an SSH session. How can I quickly forward a port?
- **💡 Solution:** SSH Escape Sequences:
 - Hit `Enter` `~` `c` (now you should see a ``ssh>`` prompt)
 - Add ``-L 8000:localhost:8000`` `Enter` to forward port 8000
 - You can add any other option (e.g., ``-X``) to modify your existing connection
 - More escape sequences available!

Autojump

- **!?** **Problem:** Changing directories in the terminal is cumbersome.
- **💡 Solution:** Autojump learns which directories you visit often. Hit ``j <some part of directory name>`` to directly jump there
- Installation instructions on [github](#)

Usage:

```
1  cd codas-hep # <-- autojump remembers this
2
3  cd ../../my-directory
4  cd some-subfolder
5
6  j codas # <-- get back to codas-hep folder
```

Terminal kung-fu

- 💡 You can quickly search through your terminal history with `Ctrl R`
- 💡 You can reference the last word of the previous command with `!$`

```
1 mkdir /path/to/some/directory/hello-world
2 cd !$
```

- 💡 Many more tricks! Read up on your shell!
- 💡 If you're using `bash`, consider switch to `zsh` (almost completely compatible) and install `oh-my-zsh` to get beautiful prompts, autocomplete on steroids and many small benefits

```
1 $ ~/D/P/x↵
2 ~/Document/Projects/xonsh/
3 $ part↵
4 this-is-part-of-a-filename
```

Tracking Jupyter notebooks with git

- **!?** **Problem:** Tracking & collaborating on Jupyter notebooks with git is a mess because of binary outputs (images) and additional metadata:
 - ``git diff`` becomes unreadable
 - merge conflicts appear often
- 💡 **Solutions:** You have several options
 1. Always strip output from notebooks before committing (easy but half-hearted)
 2. Synchronize Jupyter notebooks and python files; only track python files (slightly more advanced but best option IMO)
 3. Do not change how you *track* Jupyter notebooks; change how you *compare* them (use if you *really* want to track outputs)
 4. Avoid large amounts of code in notebooks so that the issue is less important; create python packages and use hot code reloading instead

Tracking Jupyter notebooks with git

Option 1: Track notebooks but strip outputs before committing. Add the following pre-commit hook:

```
1 - repo: https://github.com/kynan/nbstripout
2   rev: 0.5.0
3   hooks:
4     - id: nbstripout
```

Option 2: Synchronize Jupyter notebooks (untracked) to python files (tracked)

```
1 pipx install jupytext
2 echo "*.ipynb" >> ~/.gitignore # <-- tell git to ignore notebooks
3 jupytext --to py mynotebook.ipynb
4 # Now you have mynotebook.py
5 git commit mynotebook.py -m "... "
6 git push
7 # People modify the file online
8 git pull # <-- mynotebook.py got updated
9 jupytext --sync # <-- update mynotebook.ipynb
10 # Now make changes to your mynotebook.ipynb
11 jupytext --sync # <-- now mynotebook.py got updated
12 git commit ... && git push ...
```

Avoiding dependency hell

- **!?** **Problem:** Python packages depend on other packages depending on other packages causing a conflict.
- **💡 Solution:** Use conda or virtual environments (``venv``, ``virtualenv``, ``virtualenvwrapper``);

The first environment should be named ``venv``

- The Python Launcher for Unix, ``py`` picks up ``venv`` automatically!
- Visual Studio Code does too, as do a growing number of other tools.

- **!?** **Problem:** What about ``pip``-installable executables?
- **💡 Solution:** Install them with ``pipx`` instead of ``pip``! Examples:
 - ``pre-commit`` · ``black`` · ``cookiecutter`` · ``uproot-browser``

You can also use ``pipx run`` to install & execute in one step, cached for a week!

Lockfiles

- **!?** **Problem:** Upgrades *can* break things.
- **🚫** **Not a solution:** Don't add upper caps to *everything!* Only things with 50%+ chance of breaking.
- **💡** **Solution:** Use lockfiles.

Your CI and/or application (including an analysis) should have a *completely pinned environment* that works. This is not your install requirements for a library!

```
1 pip install pip-tools
2 pip-compile requirements.in # -> requirements.txt
```

Now you get a locked requirements file that can be installed:

```
1 pip install -r requirements.txt
```

Locking package managers

Locking package managers (`pdm`, `poetry`, `pipenv`) give you this with a nice all-in-one CLI:

```
1 pdm init # Setup environment using existing lockfile or general requirements
2
3 # Modify pyproject.toml as needed
4
5 pdm add numpy # Shortcut for adding to toml + install in venv
```

You'll also have a `pdm.lock` file tracking the environment it created. You can update the locks:

```
1 pdm update
```

Read up on how to use the environment that this makes to run your app.

Task runners

- **!?** **Problem:** There are lots of way to setup environments, lots of ways to run things.
- **💡** **Solution:** A task runner (nox, tox, hatch) can create a reproducible environment with no setup.
- Nox is nice because it uses Python for configuration, and prints what it is doing.

```
1 import nox
2
3 @nox.session
4 def tests(session):
5     session.install(".[test]")
6     session.run("pytest")
```


Task runners

- **!?** **Problem:** There are lots of way to setup environments, lots of ways to run things.
- **💡** **Solution:** A task runner (nox, tox, hatch) can create a reproducible environment with no setup.
- Nox is nice because it uses Python for configuration, and prints what it is doing.

```
1 import nox
2
3 @nox.session
4 def tests(session: nox.Session) -> None:
5     """
6     Run the unit and regular tests.
7     """
8     session.install(".[test]")
9     session.run("pytest", *session.posargs)
```

Task runners

Example 1: adapted from `PyPA/manylinux``

```

1  @nox.session(python=["3.9", "3.10", "3.11"])
2  def update_python_dependencies(session):
3      session.install("pip-tools")
4      session.run(
5          "pip-compile", # Usually just need this
6          "--generate-hashes",
7          "requirements.in", # and this
8          "--upgrade",
9          "--output-file",
10         f"requirements{session.python}.txt",
11     )

```

Example 2: `python.packaging.org``

```

1  @nox.session(py="3")
2  def preview(session):
3      session.install("sphinx-autobuild")
4      build(session, autobuild=True)

```

```

1  @nox.session(py="3")
2  def build(session, autobuild=False):
3      session.install("-r", "requirements.txt")
4      shutil.rmtree(target_build_dir,
5                    ignore_errors=True)
6
7  if autobuild:
8      command = "sphinx-autobuild"
9      extra_args = "-H", "0.0.0.0"
10 else:
11     command = "sphinx-build"
12     extra_args = (
13         "--color",
14         "--keep-going",
15     )
16
17     session.run(
18         command, *extra_args,
19         "-j", "auto",
20         "-b", "html",
21         "-n", "-W",
22         *session.posargs,
23         "source", "build",
24     )

```

pytest tricks (config)

Reminder: <https://scikit-hep.org/developer/pytest> is a great place to look for tips!

And reminder: pytest looks like this:

```
1 def test_func():
2     assert 4 == 2**2
```

Let's start with the first tip: your `project.toml` file should look like this:

```
1 [tool.pytest.ini_options]
2 minversion = "6.0"
3 addopts = ["-ra", "--strict-markers", "--strict-config"]
4 xfail_strict = true
5 filterwarnings = ["error"]
6 log_cli_level = "info"
7 testpaths = ["tests"]
```

pytest tricks (running)

- `--showlocals`: Show all the local variables on failure
- `--pdb`: Drop directly into a debugger on failure
- `--trace --lf`: Run the last failure & start in a debugger
- You can also add `breakpoint()` in your code to get into a debugger

pytest tricks (running)

Approx

```
1 def test_approx():
2     0.3333333333333333 == pytest.approx(1 / 3)
```

This works natively on arrays, as well!

Test for errors

```
1 def test_raises():
2     with pytest.raises(ZeroDivisionError):
3         1 / 0
```

Marks

```
1 @pytest.mark.skipif("sys.version_info >= (3, 7)")
2 def test_only_on_37plus():
3     x = 3
4     assert f"{x = }" == "x = 3"
```

Fixtures allow reuse, setup, etc

There are quite a few built-in fixtures. And you can write more:

```
1 @pytest.fixture(
2     params=["Linux", "Darwin", "Windows"],
3     autouse=True)
4 def platform_system(request, monkeypatch):
5     monkeypatch.setattr(
6         platform, "system", lambda _: request.param)
7
8 def test_thing(platform: str):
9     assert platform in {"Linux", "Darwin", "Windows"}
```

Monkeypatching

System IO, GUIs, hardware, slow processes; there are a lot of things that are hard to test! Use monkeypatching to keep your tests fast and "unit".

Type checking

- **! Problem:** Compilers catch lots of errors in compiled languages that are runtime errors in Python! Python can't be used for lots of code!
- **💡 Solution:** Add types and run a type checker.

Typed code looks like this:

```
1 def f(x: float) -> float:
2     y = x**2
3     return y
```

- Functions always have types in and out
- Variable definitions rarely have types

How do we use it?

```
1 mypy --strict tmp.py
2 Success: no issues found in 1 source file
```

Some type checkers: MyPy (Python), Pyright (Microsoft), Pytype (Google), or Pyre (Meta).¹or any IPython system

Type checking (details)

- Adds text - but adds *checked content* for the reader!
- External vs. internal typing
- Libraries need to provide typing *or* stubs can be written
- Many stubs are available, and many libraries have types (numpy, for example)
- An *active* place of development for Python & libraries!

```
1  from __future__ import annotations
2
3
4  def f(x: int) -> list[int]:
5      return list(range(x))
6
7
8  def g(x: str | int) -> None:
9      if isinstance(x, str):
10         print("string", x.lower())
11     else:
12         print("int", x)
```

Type checking (Protocol)

But Python is duck-typed! Noooooooo!

Duck typing can be formalized by a Protocol:

```
1  from typing import Protocol # or typing_extensions for < 3.8
2
3  class Duck(Protocol):
4      def quack() -> str:
5          ...
6
7  def pester_duck(a_duck: Duck) -> None:
8      print(a_duck.quack())
9
10 class MyDuck:
11     def quack() -> str:
12         return "quack"
13
14 if typing.TYPE_CHECKING:
15     _: Duck = typing.cast(MyDuck, None)
```


Type checking (pre-commit)

```
1 - repo: https://github.com/pre-commit/mirrors-mypy
2   rev: "v0.971"
3   hooks:
4     - id: mypy
5       files: src
6       args: []
7       additional_dependencies: [numpy==1.22.1]
```

- Args should be empty, or have things you add (pre-commit's default is poor)
- Additional dependencies can exactly control your environment for getting types

Benefits

- Covers all your code without writing tests
 - Including branches that you might forget to run, cases you might forget to add, etc.
- Adds vital information for your reader following your code
- All mistakes displayed at once, good error messages
- Unlike compiled languages, you can lie if you need to
- You can use `mypyc` to compile (2-5x speedup for mypy, 2x speedup for black)`

ACT (for GitHub Actions)

- **! Problem:** You use GitHub Actions for everything. But what if you want to test the run out locally?
- **💡 Solution:** Use ACT (requires Docker)!

```
1 # Install with something like brew install act
2
3 act # Runs on: push
4
5 act pull_request -j test # runs the test job as if it was a pull request
```

If you use a task runner, like nox, you should be able to avoid this most of the time. But it's handy in a pinch! <https://github.com/nektos/act>

Python libraries: Rich, Textual, Rich-cli

Textualize is one of the fastest growing library families. Recently Rich was even vendored into Pip!

progress bar demo (Using Python 3.11 TaskGroups, because why not)

```
1  from rich.progress import Progress
2  import asyncio
3
4  async def lots_of_work(n: int, progress: Progress) -> None:
5      for i in progress.track(range(n), description=f"[red]Computing {n}..."):
6          await asyncio.sleep(.1)
7
8  async def main():
9      with Progress() as progress:
10         async with asyncio.TaskGroup() as g:
11             g.create_task(lots_of_work(40, progress))
12             g.create_task(lots_of_work(30, progress))
13
14  asyncio.run(main())
```

Rich: Beautiful terminal output

Rich is not just a "color terminal" library.

- Color and styles
- Console markup
- Syntax highlighting
- Tables, panels, trees
- Progress bars and live displays
- Logging handlers
- Inspection
- Traceback formatter
- Render to SVG

Rich
Rich Features

Colors

- ✓ 4-bit color
- ✓ 8-bit color
- ✓ Truecolor (16.7 million)
- ✓ Dumb terminals
- ✓ Automatic color conversion

Styles All ansi styles: **bold**, *dim*, *italic*, underline, ~~strikethrough~~, **reverse**, and even Blink.

Text Word wrap text. Justify left, center, right or full.

Asian language support

- 该库支持中文、日文和韩文本!
- ライブラリは中国語、日本語、韓国語のテキストをサポートしています
- 이 라이브러리는 중국어, 일본어 및 한국어 텍스트를 지원합니다

Markup Rich supports a simple brcode-like markup for color, style, and emoji 🍌 🍍 🍉 🍊 🍋 🍌 🍍 🍉 🍊 🍋

Tables

Date	Title	Production Budget	Box Office
Dec 20, 2019	Star Wars: The Rise of Skywalker	\$275,000,000	\$375,126,118
May 25, 2018	Solo: A Star Wars Story	\$275,000,000	\$393,151,347
Dec 15, 2017	Star Wars Ep. VIII: The Last Jedi	\$262,000,000	\$1,332,539,889
May 19, 1999	Star Wars Ep. I: The phantom Menace	\$115,000,000	\$1,027,044,677

Syntax highlighting & pretty printing

```

1 def iter_last(values: Iterable[T]) -> Iterable[Tuple {
2     """Iterate and generate a tuple with a flag fo
3     iter_values = iter(values)
4     try:
5         previous_value = next(iter_values)
6     except StopIteration:
7         return
8     for value in iter_values:
9         yield False, previous_value
10        previous_value = value
11    yield True, previous_value

```

```

{ 'foo': {
  3.1427,
  (
    'Paul Atreides',
    'Vladimir Harkonnen',
    'Thufir Hawat'
  )
},
  atomic': (False, True, None)
}

```

Markdown # Markdown

Supports much of the *markdown* `_syntax_!`

- Headers
- Basic formatting: **bold***, *italic**, ``code``
- Block quotes
- Lists, and more...

Supports much of the *markdown* syntax!

- Headers
- Basic formatting: **bold**, *italic*, `code`
- Block quotes
- Lists, and more...

+more! Progress bars, columns, styled logging handler, tracebacks, etc...

Textual: GUI? No, TUI!

New "CSS" version coming soon!

Rich-cli: Rich as a command line tool

```

-bash
> rich loop.py
from typing import Iterable, Tuple, TypeVar

T = TypeVar("T")

def loop_first(values: Iterable[T]) -> Iterable[Tuple[bool, T]]:
    """Iterate and generate a tuple with a flag for first value."""
    iter_values = iter(values)
    try:
        value = next(iter_values)
    except StopIteration:
        return
    yield True, value
    for value in iter_values:
        yield False, value
    
```

```

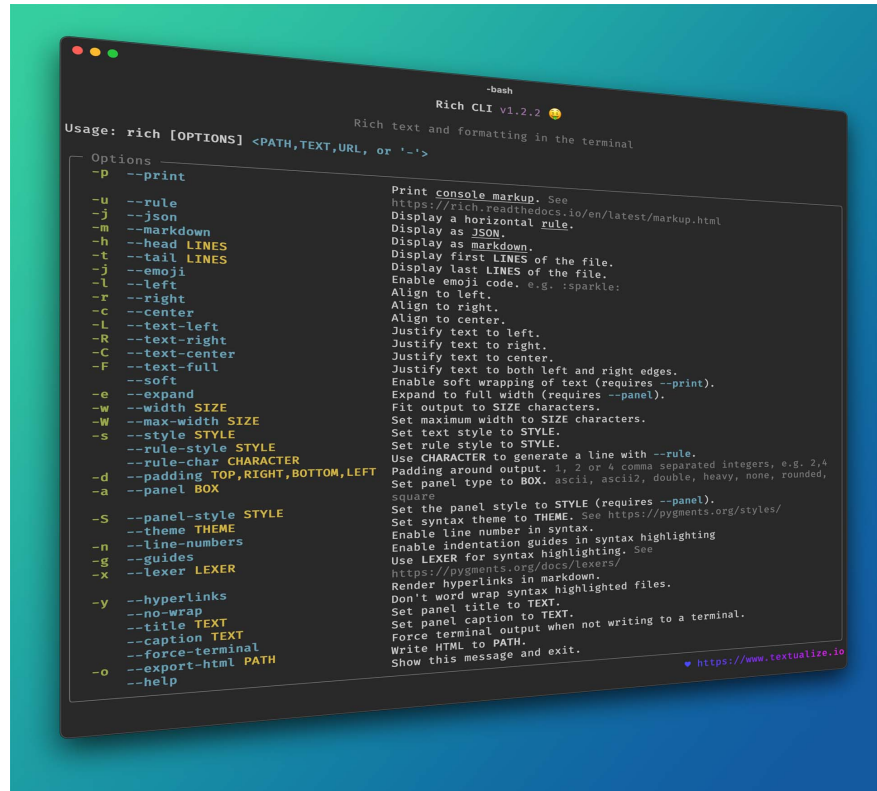
-bash
Rich Library

Rich contains a number of builtin renderables you can use to create elegant output in your CLI and help you debug your code.

Click the following headings for details:

The Console object has a log() method which has a similar interface to print(), but also renders a column for the current time and the file and line which made the call. By default Rich will do syntax highlighting for Python structures and for repr strings. If you log a collection (i.e. a dict or a list) Rich will pretty print it so that it fits in the available space. Here's an example of some of these features.

from rich.console import Console
console = Console()
    
```



WebAssembly

- **!?** **Problem:** Distributing code is hard. Binder takes time to start & requires running the code on someone else's machine.
- **💡 Solution:** Use the browser to *run* the code with a WebAssembly distribution, like Pyodide.
Python 3.11 officially supports it now too! Binaries may be provided around 3.12!

Pyodide

A distribution of CPython 3.10 including ~100 binary packages like SciPy, Pandas, boost-histogram (Hist), etc.

Examples:

- <https://henryiii.github.io/level-up-your-python/live/lab/index.html>
- <https://scikit-hep.org/developer/reporeview>

PyScript

An Python interface for Pyodide in HTML.

WebAssembly - PyScript

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="utf-8">
5    <meta name="viewport" content="width=device-width, initial-scale=1">
6    <title>Hello, World!</title>
7    <link rel="stylesheet" href="https://pyscript.net/alpha/pyscript.css" />
8    <script defer src="https://pyscript.net/alpha/pyscript.js"></script>
9  </head>
10 <body>
11   <py-script>print("Hello, World!")</py-script>
12 </body>
13 </html>
```

<https://realpython.com/pyscript-python-in-browser>

Modern packaging

- **!?** **Problem:** Making a package is hard.
- **💡** **Solution:** It's not hard anymore. You just need to use modern packaging and avoid old examples.

```
1 [build-system]
2   requires = ["hatchling"]
3   build-backend = "hatchling.build"
4
5 [project]
6   name = "package"
7   version = "0.0.1"
```

Other metadata should go there too, but that's the minimum. See links:

- <https://scikit-hep.org/developer/pep621>
- <https://packaging.python.org/en/latest/tutorials/packaging-projects>

`scikit-hep/cookie` supports 11 backends; hatchling is recommended for pure Python. For compiled extensions: See next slides(s). 😊

Binary packaging

- **!?** **Problem:** Making a package with binaries is hard.
- 💡 **Solution:** Some parts are easy, and I'm working on making the other parts easy too!

Making the code

Use a tool like pybind11, Cython, or MyPyC. It's hard to get the C API right!

```
1  #include <pybind11/pybind11.hpp>
2
3  int add(int i, int j) {
4      return i + j;
5  }
6
7  PYBIND11_MODULE(example, m) {
8      m.def("add", &add);
9  }
```

Header only, pure C++! No dependencies, no pre-compile step, no new language.

Configuring the build

I'm working on scikit-build for the next three years! CMake for Python packaging.

Currently based on distutils & setuptools - but will be rewritten!

Org of several packages:

- Scikit-build
- CMake for Python
- Ninja for Python
- moderncmakedomain
- Examples

Building the binaries

Redistributable wheel builder.

- Targeting macOS 10.9+
- Apple Silicon cross-compiling 3.8+
- All variants of manylinux (including emulation)
- musllinux
- PyPy 3.7-3.9
- Repairing and testing wheels
- Reproducible pinned defaults (can unpin)

Local runs supported too!

```
1 pipx run cibuildwheel --platform linux
```

GitHub actions example

```
1 on: [push, pull_request]
2
3 jobs:
4   build_wheels:
5     runs-on: ${{ matrix.os }}
6     strategy:
7       matrix:
8         os:
9           - ubuntu-22.04
10          - windows-2022
11          - macos-11
12
13    steps:
14      - uses: actions/checkout@v4
15
16      - name: Build wheels
17        uses: pypa/cibuildwheel@v2.8.1
18
19      - uses: actions/upload-artifact@v3
20        with:
21          path: ./wheelhouse/*.whl
```