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.

11/2/2022

[1] or any IPython system

Git & starting new projects

[1] or any IPython system

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 🎉

[1] or any IPython system

Pre-commit hooks 🪝

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

```yaml
repos:
- repo: https://github.com/pre-commit/pre-commit-hooks
  rev: 'v4.3.0'
  hooks:
  - id: check-added-large-files
  - id: check-case-conflict
  - id: check-merge-conflict
  - id: detect-private-key
  - id: end-of-file-fixer
  - id: trailing-whitespace

- repo: https://github.com/codespell-project/codespell  # the spell checker with ~0 false positives
  rev: 'v2.1.0'
  hooks:
  - id: codespell
    # args: ["-I", "codespell.txt"]  # Optional to add exceptions

ci:
    autoupdate_schedule: monthly # default is weekly
```

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

[1] or any IPython system

Pre-commit hooks for python!

```
-   repo: https://github.com/psf/black  # Reformat code without compromises!
    rev: '22.6.0'
    hooks:
    -   id: black
    # or, if you also work with Jupyter notebooks
    # -   id: black-jupyter
-   repo: https://github.com/PyCQA/flake8  # Simple static checks
    rev: '5.0.1'
    hooks:
    -   id: flake8
        additional_dependencies: ['flake8-bugbear']
-   repo: https://github.com/pre-commit/mirrors-mypy  # Check typing (slightly more advanced)
    rev: 'v0.971'
    hooks:
    -   id: mypy
-   repo: https://github.com/asottile/pyupgrade  # Automatically upgrade old Python syntax
    rev: 'v2.37.2'
    hooks:
    -   id: pyupgrade
        args: [--py37-plus]
```

- **Try it out**: Go here for a quick step-by-step tutorial

[1] or any IPython system

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**:

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

[1] or any IPython system

SSH & Terminal Kung-fu

[1] or any IPython system

SSH Config

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

```
# Server I want to connect to
Host tiger*
    Hostname tiger.princeton.edu
    User kl5675

# Tunnel that I might use sometimes
Host tigressgateway
    Hostname tigressgateway.princeton.edu
    User kl5675

Host *-t
    ProxyJump tigressgateway
```

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

[1] or any IPython system

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
  - More escape sequences available! More information.
- **Caveat**: `C` option not available in multiplexed sessions.

[1] or any IPython system

Terminal kung-fu

- 💡 You can quickly search through your terminal history with `Ctrl` `R` - start typing
  - Hit `Ctrl` `R` to navigate between different hits
- 💡 You can reference the last word of the previous command with `` `!$` ``

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

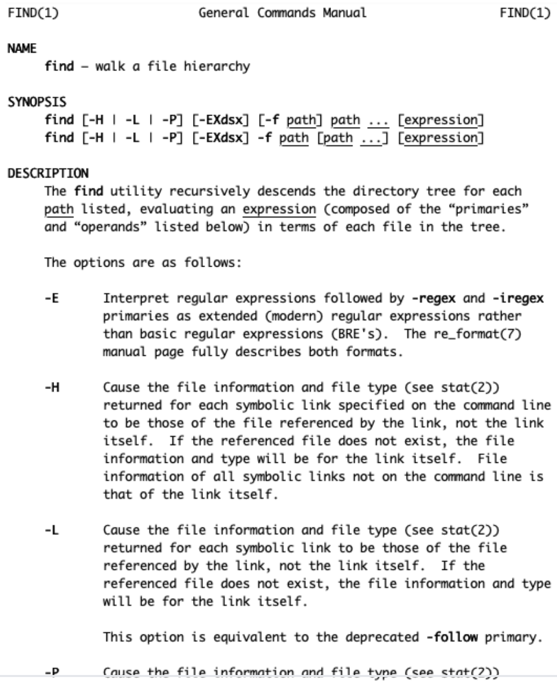
- 💡 Want to fix up a complicated command that just failed? Type `` `fc` `` to edit the command in your `` `$EDITOR` ``
- 💡 If you're using `` `bash` ``, consider switching to `` `zsh` `` (almost completely compatible) and install `` `oh-my-zsh` `` to get beautiful prompts, autocomplete on steroids and many small benefits

```
$ ~/D/P/x⇥
~/Document/Projects/xonsh/
$ part⇥
this-is-part-of-a-filename
```

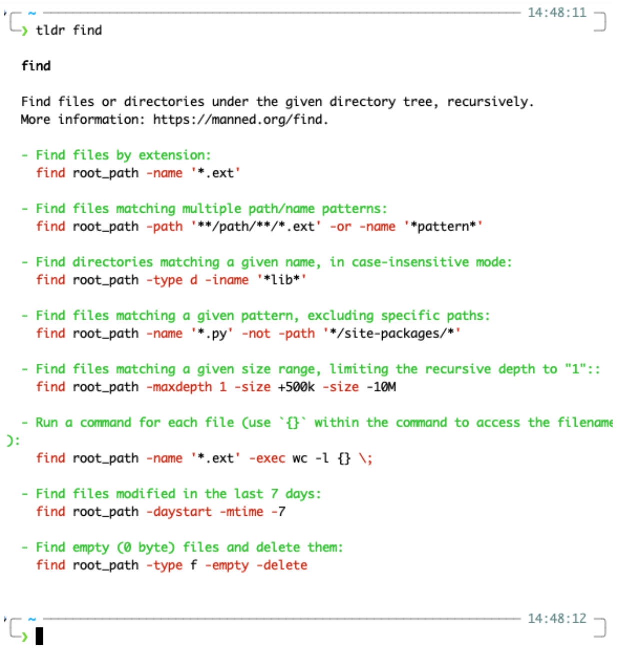
[1] or any IPython system

How to shell...

- **!? Problem:** `man` pages are wasting your time?
- 💡 **Solution:** Try `tldr` (`pipx install tldr`). Compare:



```
FIND(1)                General Commands Manual                FIND(1)

NAME
     find – walk a file hierarchy

SYNOPSIS
     find [-H | -L | -P] [-EXdsx] [-f path] path ... [expression]
     find [-H | -L | -P] [-EXdsx] -f path [path ...] [expression]

DESCRIPTION
     The find utility recursively descends the directory tree for each
     path listed, evaluating an expression (composed of the “primaries”
     and “operands” listed below) in terms of each file in the tree.

     The options are as follows:

     -E      Interpret regular expressions followed by -regex and -iregex
             primaries as extended (modern) regular expressions rather
             than basic regular expressions (BRE's).  The re_format(7)
             manual page fully describes both formats.

     -H      Cause the file information and file type (see stat(2))
             returned for each symbolic link specified on the command line
             to be those of the file referenced by the link, not the link
             itself.  If the referenced file does not exist, the file
             information and type will be for the link itself.  File
             information of all symbolic links not on the command line is
             that of the link itself.

     -L      Cause the file information and file type (see stat(2))
             returned for each symbolic link to be those of the file
             referenced by the link, not the link itself.  If the
             referenced file does not exist, the file information and type
             will be for the link itself.

             This option is equivalent to the deprecated -follow primary.

     -P      Cause the file information and file type (see stat(2))
```

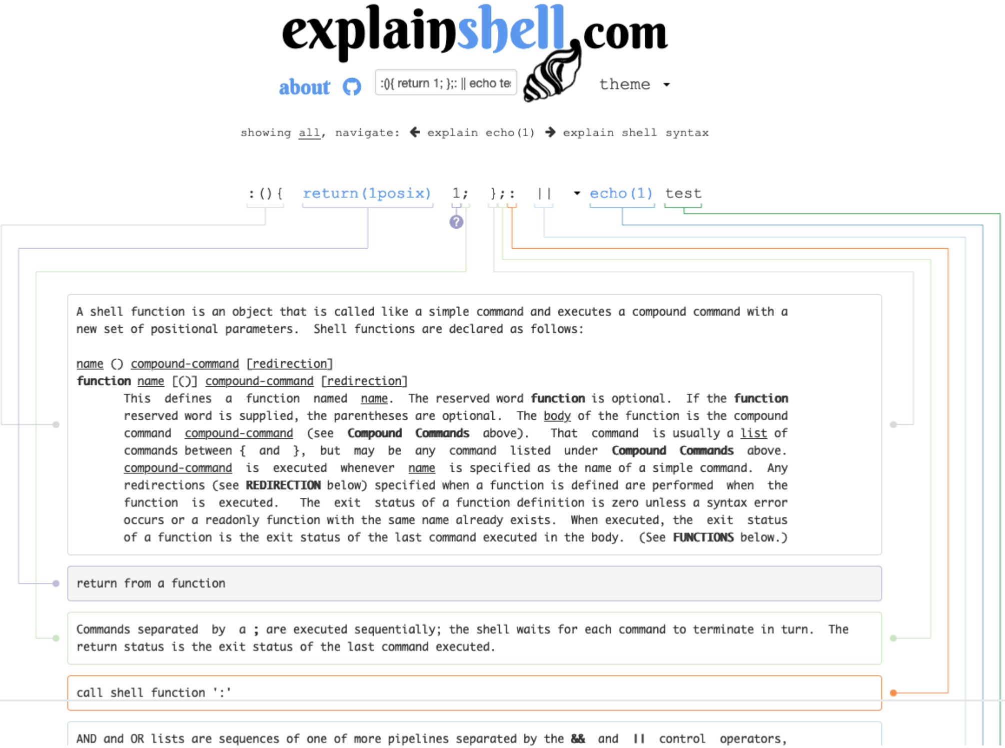


```
┌ ~ ─────────────────────────────────────────────────── 14:48:11 ┐
└› tldr find

  find

  Find files or directories under the given directory tree, recursively.
  More information: https://manned.org/find.

  - Find files by extension:
    find root_path -name '*.ext'

  - Find files matching multiple path/name patterns:
    find root_path -path '**/path/**/*.ext' -or -name '*pattern*'

  - Find directories matching a given name, in case-insensitive mode:
    find root_path -type d -iname '*lib*'

  - Find files matching a given pattern, excluding specific paths:
    find root_path -name '*.py' -not -path '*/site-packages/*'

  - Find files matching a given size range, limiting the recursive depth to "1"::
    find root_path -maxdepth 1 -size +500k -size -10M

  - Run a command for each file (use `{}` within the command to access the filename
):

    find root_path -name '*.ext' -exec wc -l {} \;

  - Find files modified in the last 7 days:
    find root_path -daystart -mtime -7

  - Find empty (0 byte) files and delete them:
    find root_path -type f -empty -delete


┌ ~ ─────────────────────────────────────────────────── 14:48:12 ┐
└› █
```

[1]or any IPython system

How to shell…

- **!? Problem:** Understanding cryptic bash commands
- 💡 **Solutions:** Go to explainshell.com



explainshell.com

about

:(){ return 1; };: || echo te

theme

showing all, navigate: ← explain echo(1) → explain shell syntax

```
:(){ return(1posix) 1; };: || echo(1) test
```

A shell function is an object that is called like a simple command and executes a compound command with a new set of positional parameters. Shell functions are declared as follows:

name () compound-command [redirection]
function name [()] compound-command [redirection]

This defines a function named name. The reserved word **function** is optional. If the **function** reserved word is supplied, the parentheses are optional. The body of the function is the compound command compound-command (see **Compound Commands** above). That command is usually a list of commands between { and }, but may be any command listed under **Compound Commands** above. compound-command is executed whenever name is specified as the name of a simple command. Any redirections (see **REDIRECTION** below) specified when a function is defined are performed when the function is executed. The exit status of a function definition is zero unless a syntax error occurs or a readonly function with the same name already exists. When executed, the exit status of a function is the exit status of the last command executed in the body. (See **FUNCTIONS** below.)

return from a function

Commands separated by a ; are executed sequentially; the shell waits for each command to terminate in turn. The return status is the exit status of the last command executed.

call shell function ':'

AND and OR lists are sequences of one of more pipelines separated by the **&&** and **||** control operators,

[1]or any IPython system

File navigation and completion

- **!? 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

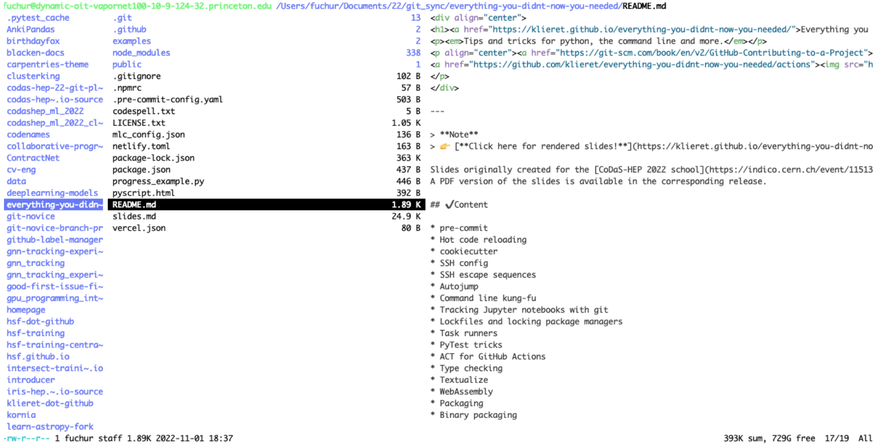
Usage:

```
cd codas-hep  # <-- autojump remembers this

cd ../../my-directory
cd some-subfolder

j codas  # <-- get back to codas-hep folder
```

[1] or any IPython system

File navigation and completion

- **!? Problem:** I like visual file managers, but I'm working on a server...
- 💡 **Solution:** Use a terminal file manager, e.g., `ranger` (`pipx install ranger-fm`)



```
fuchur@dynamic-oit-vapornet100-10-9-124-32.princeton.edu /Users/fuchur/Documents/22/git_sync/everything-you-didnt-now-you-needed/README.md
 .pytest_cache        .git                                  13  <div align="center">
 AnkiPandas           .github                                2  <h1><a href="https://klieret.github.io/everything-you-didnt-now-you-needed/">Everything you
 birthdayfox          examples                               2  <p><em>Tips and tricks for python, the command line and more.</em></p>
 blacken-docs         node_modules                         338  <p align="center"><a href="https://git-scm.com/book/en/v2/GitHub-Contributing-to-a-Project">
 carpentries-theme    public                                 1  <a href="https://github.com/klieret/everything-you-didnt-now-you-needed/actions"><img src="h
 clusterking          .gitignore                         102 B  </p>
 codas-hep-22-git-pl~ .npmrc                              57 B  </div>
 codas-hep~.io-source .pre-commit-config.yaml            503 B
 codashep_ml_2022     codespell.txt                        5 B  ---
 codashep_ml_2022_cl~ LICENSE.txt                       1.05 K
 codenames            mlc_config.json                    136 B  > **Note**
 collaborative-progr~ netlify.toml                       163 B  > 👉 [**Click here for rendered slides!**](https://klieret.github.io/everything-you-didnt-no
 ContractNet          package-lock.json                  363 K
 cv-eng               package.json                       437 B  Slides originally created for the [CoDaS-HEP 2022 school](https://indico.cern.ch/event/11513
 data                 progress_example.py                446 B  A PDF version of the slides is available in the corresponding release.
 deeplearning-models  pyscript.html                      392 B
 everything-you-didn~ README.md                         1.89 K  ## ✓Content
 git-novice           slides.md                         24.9 K
 git-novice-branch-pr vercel.json                         80 B  * pre-commit
 github-label-manager                                           * Hot code reloading
 gnn-tracking-experi~                                           * cookiecutter
 gnn_tracking                                                   * SSH config
 gnn_tracking_experi~                                           * SSH escape sequences
 good-first-issue-fi~                                           * Autojump
 gpu_programming_int~                                           * Command line kung-fu
 homepage                                                       * Tracking Jupyter notebooks with git
 hsf-dot-github                                                 * Lockfiles and locking package managers
 hsf-training                                                   * Task runners
 hsf-training-centra~                                           * PyTest tricks
 hsf.github.io                                                  * ACT for GitHub Actions
 intersect-traini~.io                                           * Type checking
 introducer                                                     * Textualize
 iris-hep.~.io-source                                           * WebAssembly
 klieret-dot-github                                             * Packaging
 kornia                                                         * Binary packaging
 learn-astropy-fork
-rw-r--r-- 1 fuchur staff 1.89K 2022-11-01 18:37                                              393K sum, 729G free  17/19  All
```

- **!? Problem**: I search through files a lot, but `grep` is slow/inconvenient...
- 💡 **Solution:** There are faster and more feature-rich alternatives. Example: ripgrep, `ag`, ...
- **!? Problem:** Even with tab completion, completing file names is cumbersome.
- 💡 **Solution:** Try type-ahead-searching/fuzzy matching, e.g., with `fzf` with shell integration, e.g., `vim ../**` Tab starts searching with `fzf` in parent dir

[1] or any IPython system

Python

[1] or any IPython system

Hot code reloading

- **!? Problem:**
  1. I have some code in a notebook and some code in a python file/library.
  2. I update my python file/library.
  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 notebook[1] to reload all (!) modules every time you execute code

```
%load_ext autoreload
%autoreload 2  # reload everything
```

- **More granular**:

```
%load_ext autoreload
%autoreload 1  # <-- reload only some modules

# Mark for reloading
%aimport foo
```

- **Warning:** These tricks don't *always* work, but it should save you from a lot of restarts
- **Try it out!** Follow our instructions here.
- **More information**: See the autoreload documentation

[1] or any IPython system

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). Example: `` `nbdime` ``
  4. Avoid large amounts of code in notebooks so that the issue is less important; create python packages and use hot code reloading instead

[1] or any IPython system

Tracking Jupyter notebooks with git

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

```yaml
- repo: https://github.com/kynan/nbstripout
  rev: 0.5.0
  hooks:
  - id: nbstripout
```

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

```bash
pipx install jupytext
echo "*.ipynb" >> ~/.gitignore  # <-- tell git to ignore noteboks
jupytext --to py mynotebook.ipynb
# Now you have mynotebook.py
git commit mynotebook.py -m "..."
git push
# People modify the file online
git pull  # <-- mynotebook.py got updated
jupytext --sync  # <-- update mynotebook.ipynb
# Now make changes to your mynotebook.ipynb
jupytext --sync  # <-- now mynotebook.py got updated
git commit ... && git push ...
```

Satic code checkers and Jupyter notebooks

- **!? Problem:** I still have lots of code in my notebooks. I still want to apply tools like black, pyupgrade, … on the notebooks.
- 💡 **Solution:** `nbqa` allows to apply a lot of tools to Jupyter notebooks

```
$ pip install nbqa

$ nbqa black my_notebook.ipynb
reformatted my_notebook.ipynb
All done! ✨ 🍰 ✨
1 files reformatted.


$ nbqa pyupgrade my_notebook.ipynb --py37-plus
Rewriting my_notebook.ipynb
```

[1] or any IPython system

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!

[1] or any IPython system

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!

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

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

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

[1] or any IPython system

Locking package managers

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

```
pdm init # Setup environment using existing lockfile or general requirements

# Modify pyproject.toml as needed

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:

```
pdm update
```

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

[1] or any IPython system

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.

```python
import nox

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

[1] or any IPython system

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.

```python
import nox

@nox.session
def tests(session: nox.Session) -> None:
    """
    Run the unit and regular tests.
    """
    session.install(".[test]")
    session.run("pytest", *session.posargs)
```

[1] or any IPython system

Task runners

Example 1: adapted from `PyPA/manylinux`

```python
@nox.session(python=["3.9", "3.10", "3.11"])
def update_python_dependencies(session):
    session.install("pip-tools")
    session.run(
        "pip-compile", # Usually just need this
        "--generate-hashes",
        "requirements.in", # and this
        "--upgrade",
        "--output-file",
        f"requirements{session.python}.txt",
    )
```

Example 2: `python.packaging.org`

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

```python
@nox.session(py="3")
def build(session, autobuild=False):
    session.install("-r", "requirements.txt")
    shutil.rmtree(target_build_dir,
                  ignore_errors=True)

    if autobuild:
        command = "sphinx-autobuild"
        extra_args = "-H", "0.0.0.0"
    else:
        command = "sphinx-build"
        extra_args = (
            "--color",
            "--keep-going",
        )

    session.run(
        command, *extra_args,
        "-j", "auto",
        "-b", "html",
        "-n", "-W",
        *session.posargs,
        "source", "build",
    )
```

[1] or any IPython system

pytest: Make testing fun

Basics

`pytest` discovers test functions named `test_...` in files `test_...`. For example:

```
def test_funct():
    assert 4 == 2**2
```

To run: `pip install pytest` and then `pytest` to discover & run them all.

First tip: your `project.toml` file

```
[tool.pytest.ini_options]
minversion = "6.0"  # minimal version of pytest
# report all; check that markers are configured; check that config is OK
addopts = ["-ra", "--strict-markers", "--strict-config"]
xfail_strict = true  # tests marked as failing must fail
filterwarnings = ["error"]
log_cli_level = "info"
testpaths = ["tests"]  # search for tests in "test" directory
```

[1] or any IPython system

pytest: Make testing fun

- `--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

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

[1] or any IPython system

pytest: Make testing fun

Approx

```python
def test_approx():
    0.3333333333333 == pytest.approx(1 / 3)
```

This works natively on arrays, as well!

Test for errors

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

Marks

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

Fixtures allow reuse, setup, etc

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

```python
@pytest.fixture
def my_complex_object():
    mco = MyComplexObject(...)
    mco.xyz = "asf"
    ...
    return mco

def test_get_value(my_complex_object):
    assert my_complex_object.get_value() == ...

def test_other_property(my_complex_object):
    assert my_complex_object.property == ...
```

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".

[1] or any IPython system

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.

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

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

How do we use it? (requires `` `pipx install mypy` ``)

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

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

👉 Example files available here.

[1] or 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!

```python
from __future__ import annotations


def f(x: int) -> list[int]:
    return list(range(x))


def g(x: str | int) -> None:
    if isinstance(x, str):
        print("string", x.lower())
    else:
        print("int", x)
```

[1] or any IPython system

Type checking (Protocol)

But Python is duck-typed! Nooooooo!

Duck typing can be formalized by a Protocol:

```python
from typing import Protocol  # or typing_extensions for < 3.8

class Duck(Protocol):
    def quack(self) -> str:
        ...

def pester_duck(a_duck: Duck) -> None:
    print(a_duck.quack())

class MyDuck:
    def quack(self) -> str:
        return "quack"

# Check explicitly that MyDuck is implementing the Duck protocol
if typing.TYPE_CHECKING:
    _: Duck = typing.cast(MyDuck, None)
```

[1] or any IPython system

Type checking (pre-commit)

```yaml
- repo: https://github.com/pre-commit/mirrors-mypy
  rev: "v0.971"
  hooks:
    - id: mypy
      files: src
      args: []
      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 for 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)

[1] or any IPython system

GitHub Actions: pages deploy

Bonus: About a week ago GitHub Actions added direct deploy to pages!

```yaml
on:
  workflow_dispatch:
  pull_request:
  push:

permissions:
  contents: read
  pages: write
  id-token: write

concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true
```

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup Pages
        id: pages
        uses: actions/configure-pages@v1

      # Static site generation, latex, etc. here

      - name: Upload artifact
        uses: actions/upload-pages-artifact@v1
        with:
          path: dist/

  deploy:
    if: github.ref == 'refs/heads/main'
    needs: build
    environment:
      name: github-pages
      url: ${{ steps.deployment.outputs.page_url }}
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to GitHub Pages
        id: deployment
        uses: actions/deploy-pages@v1
```

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)!

```
# Install with something like brew install act

act  # Runs on: push

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

[1] or any IPython system

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)

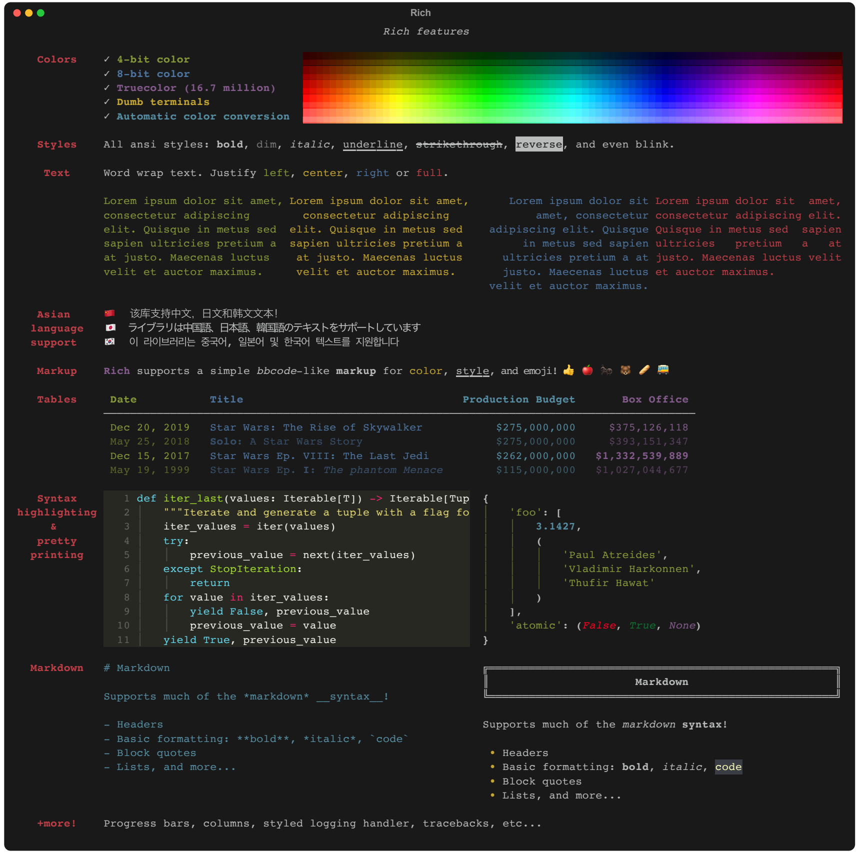
```python
from rich.progress import Progress
import asyncio

async def lots_of_work(n: int, progress: Progress) -> None:
    for i in progress.track(range(n), description=f"[red]Computing {n}..."):
        await asyncio.sleep(.1)

async def main():
    with Progress() as progress:
        async with asyncio.TaskGroup() as g:
            g.create_task(lots_of_work(40, progress))
            g.create_task(lots_of_work(30, progress))

asyncio.run(main())
```

[1] or any IPython system

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 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.

               Lorem ipsum dolor sit amet, Lorem ipsum dolor sit amet,     Lorem ipsum dolor sit Lorem ipsum dolor sit  amet,
               consectetur adipiscing        consectetur adipiscing          amet, consectetur consectetur adipiscing elit.
               elit. Quisque in metus sed  elit. Quisque in metus sed   adipiscing elit. Quisque Quisque in metus sed  sapien
               sapien ultricies pretium a  sapien ultricies pretium a       in metus sed sapien ultricies   pretium   a   at
               at justo. Maecenas luctus    at justo. Maecenas luctus     ultricies pretium a at justo. Maecenas luctus velit
               velit et auctor maximus.     velit et auctor maximus.      justo. Maecenas luctus et auctor maximus.
                                                                          velit et auctor maximus.

    Asian      🇨🇳  该库支持中文，日文和韩文文本！
   language    🇯🇵  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
   support     🇰🇷  이 라이브러리는 중국어, 일본어 및 한국어 텍스트를 지원합니다

    Markup     Rich supports a simple bbcode-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     def iter_last(values: Iterable[T]) -> Iterable[Tup  {
 highlighting      """Iterate and generate a tuple with a flag fo      'foo': [
      &            iter_values = iter(values)                              3.1427,
    pretty         try:                                                    (
   printing            previous_value = next(iter_values)                      'Paul Atreides',
                   except StopIteration:                                       'Vladimir Harkonnen',
                       return                                                  'Thufir Hawat'
                   for value in iter_values:                               )
                       yield False, previous_value                      ],
                       previous_value = value                          'atomic': (False, True, None)
                   yield True, previous_value                      }

   Markdown    # Markdown                                          ┌──────────────────────────────────────┐
                                                                   │               Markdown               │
               Supports much of the *markdown* __syntax__!         └──────────────────────────────────────┘

               - Headers                                           Supports much of the markdown syntax!
               - Basic formatting: **bold**, *italic*, `code`
               - Block quotes                                       • Headers
               - Lists, and more...                                 • Basic formatting: bold, italic, code
                                                                    • Block quotes
                                                                    • Lists, and more...

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

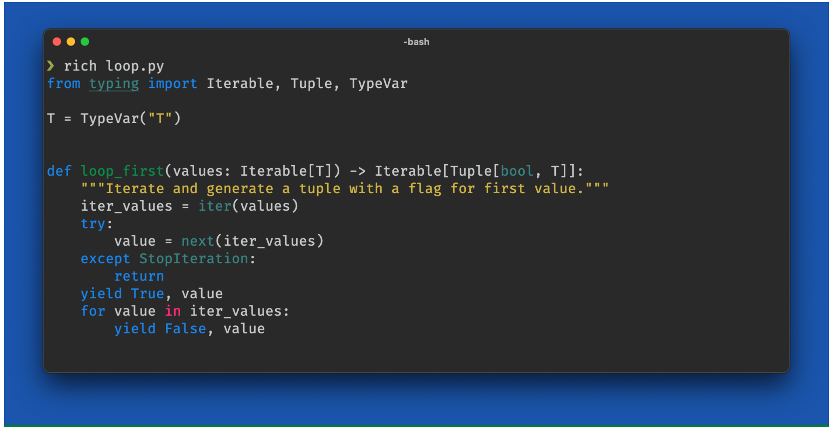
[1] or any IPython system

Textual: GUI? No, TUI!

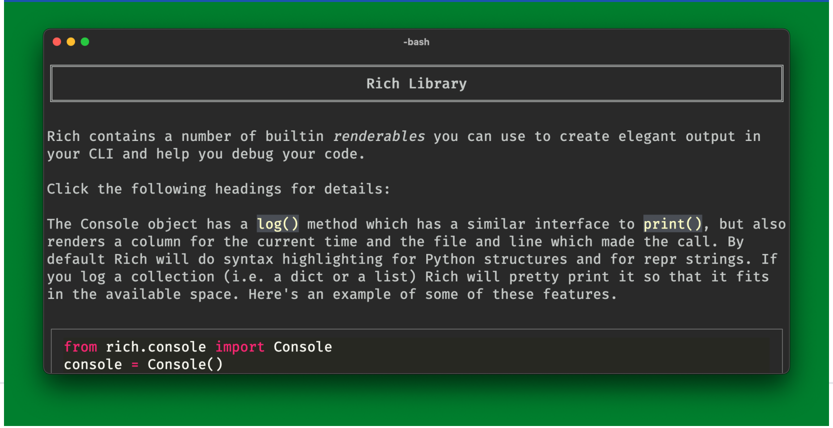
New "CSS" version coming soon!

[1] or any IPython system

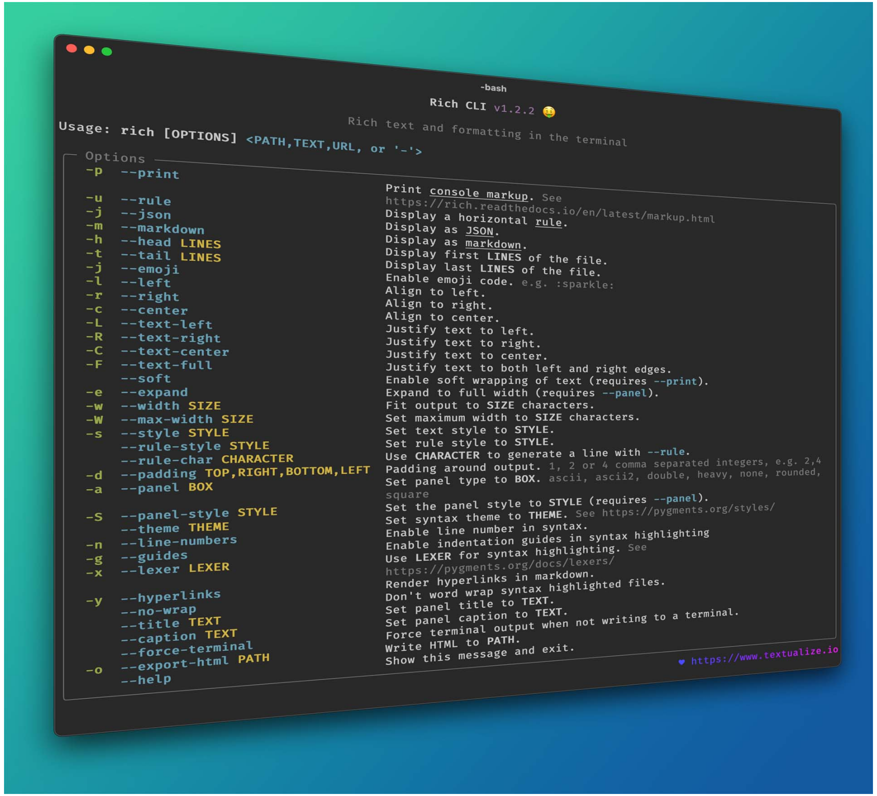
Rich-cli: Rich as a command line tool



```
-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()
```



```
-bash
                                  Rich CLI v1.2.2 🤑
                          Rich text and formatting in the terminal
Usage: rich [OPTIONS] <PATH,TEXT,URL, or '-'>
╭─ Options ─────────────────────────────────────────────────────────────────────╮
│ -p  --print                                 Print console markup. See          │
│                                             https://rich.readthedocs.io/en/latest/markup.html
│ -u  --rule                                  Display a horizontal rule.         │
│ -j  --json                                  Display as JSON.                   │
│ -m  --markdown                              Display as markdown.               │
│ -h  --head LINES                            Display first LINES of the file.   │
│ -t  --tail LINES                            Display last LINES of the file.    │
│ -j  --emoji                                 Enable emoji code. e.g. :sparkle:  │
│ -l  --left                                  Align to left.                     │
│ -r  --right                                 Align to right.                    │
│ -c  --center                                Align to center.                   │
│ -L  --text-left                             Justify text to left.              │
│ -R  --text-right                            Justify text to right.             │
│ -C  --text-center                           Justify text to center.            │
│ -F  --text-full                             Justify text to both left and right edges.
│     --soft                                  Enable soft wrapping of text (requires --print).
│ -e  --expand                                Expand to full width (requires --panel).
│ -w  --width SIZE                            Fit output to SIZE characters.     │
│ -W  --max-width SIZE                        Set maximum width to SIZE characters.
│ -s  --style STYLE                           Set text style to STYLE.           │
│     --rule-style STYLE                      Set rule style to STYLE.           │
│     --rule-char CHARACTER                   Use CHARACTER to generate a line with --rule.
│ -d  --padding TOP,RIGHT,BOTTOM,LEFT         Padding around output. 1, 2 or 4 comma separated integers, e.g. 2,4
│ -a  --panel BOX                             Set panel type to BOX. ascii, ascii2, double, heavy, none, rounded,
│                                             square
│ -S  --panel-style STYLE                     Set the panel style to STYLE (requires --panel).
│     --theme THEME                           Set syntax theme to THEME. See https://pygments.org/styles/
│ -n  --line-numbers                          Enable line number in syntax.
│ -g  --guides                                Enable indentation guides in syntax highlighting
│ -x  --lexer LEXER                           Use LEXER for syntax highlighting. See
│                                             https://pygments.org/docs/lexers/
│ -y  --hyperlinks                            Render hyperlinks in markdown.
│     --no-wrap                               Don't word wrap syntax highlighted files.
│     --title TEXT                            Set panel title to TEXT.
│     --caption TEXT                          Set panel caption to TEXT.
│     --force-terminal                        Force terminal output when not writing to a terminal.
│ -o  --export-html PATH                      Write HTML to PATH.
│     --help                                  Show this message and exit.
╰───────────────────────────────────────────────────────────────────────────────╯
                                                        ♥ https://www.textualize.io
```

[1]or any IPython system

WebAssembly

- **!? Problem:** Distributing code is hard. Binder takes time to start & requires running the code one 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.

[1] or any IPython system

WebAssembly - PyScript

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

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

[1] or any IPython system

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.

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

[project]
name = "package"
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; hatching is recommended for pure Python. For compiled extensions: See next slides(s). 😀

[1] or any IPython system

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!

[1] or any IPython system

Making the code

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

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

int add(int i, int j) {
    return i + j;
}

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

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

[1] or any IPython system

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

[1] or any IPython system

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!

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

GitHub actions example

```yaml
on: [push, pull_request]

jobs:
  build_wheels:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os:
          - ubuntu-22.04
          - windows-2022
          - macos-11

    steps:
    - uses: actions/checkout@v4

    - name: Build wheels
      uses: pypa/cibuildwheel@v2.8.1

    - uses: actions/upload-artifact@v3
      with:
        path: ./wheelhouse/*.whl
```

[1] or any IPython system