

# Everything\* you didn't know you needed

\*blatant marketing nonsense

Kilian Lieret and Henry Schreiner

Princeton University

CoDaS-HEP school 2023





Slides available as [open source](#), contributions welcome.

7/21/2023

# Git & starting new projects

# 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: '4.4.0'
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: 'v2.2.5'
14    hooks:
15      - id: codespell
16        # args: ["-I", "codespell.txt"] # Optional to add exceptions
17
18  ci:
19    autoupdate_schedule: monthly # default is weekly
```

See [Scientific Python Development Guidelines](#) for many more, updated weekly!

<sup>1</sup>or any IPython system

# Pre-commit hooks for python!

```
1     # Reformat code without compromises!
2     -   repo: https://github.com/psf/black
3       rev: '23.7.0'
4     hooks:
5     -   id: black
6     # Static checks
7     -   repo: https://github.com/astral-sh/ruff-pre-commit
8       rev: 'v0.0.278'
9     hooks:
10    -   id: ruff
11        args: [--fix, --show-fixes]
12    # Check typing (slightly more advanced)
13    -   repo: https://github.com/pre-commit/mirrors-mypy
14      rev: 'v1.4.1'
15    hooks:
16    -   id: mypy
```

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

# Cookiecutter

- **!?** **Problem:** Setting up 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
- **Example:** `Scientific Python Cookie`: All the features, all the best-practices
- 💡 **Pro-tip:** `cruft` is a cookiecutter extension that allows to propagate updates to the template back to the projects that use it
- 💡 **Pro-tip:** `copier` is another powerful tool that supports updates (and works with the template above).
- **Trying it out:**

```
1 pipx install cookiecutter
2 # alternative: cruft or copier
3 cookiecutter gh:scientific-python/cookie
4 # e.g., select project type = hatchling
5 # for a very simple start that grows with you.
```

# SSH & Terminal Kung-fu

# SSH Config

- **!?** **Problem:** Typing long server names 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     ProxyJump tigressgateway
```

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
  - More escape sequences available! [More information.](#)
- **Caveat:** `c` option not available in multiplexed sessions.

# Terminal kung-fu

- 💡 You can quickly search through your terminal history with `ctrl R` - start typing
  - Hit `ctrl R` to navigate between different hits
  - 🔥 Install `fzf` as a plugin to your shell to get an interactive fuzzy finding pop-up instead (interactively filter through list of results rather than just seeing one)
- 💡 You can reference the last word of the previous command with `!$`

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

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

# How to shell...

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

<pre>FIND(1)          General Commands Manual          FIND(1)  NAME   find - walk a file hierarchy  SYNOPSIS   find [-H   -L   -P] [-EXd sx] [-f path] path ... [expression]   find [-H   -L   -P] [-EXd sx] -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))</pre>	<pre>~ 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/**/*.*' -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</pre>
--	--

# How to shell...

- **!?** **Problem:** Understanding cryptic bash commands
- 💡 **Solutions:** Go to [explainshell.com](https://explainshell.com)

**explainshell.com**

about ⓘ :0{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 or more pipelines separated by the `&&` and `||` control operators,

# 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:

```
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
```

# 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`

```
fuchur@nyari-c-01t-voparnet180-18-9-124-32.princeton.edu /Users/fuchur/Documents/22/git_sync/everything-you-didnt-now-you-needed/README.md
13 <div align="center">
2 <h1><a href="https://klieret.github.io/everything-you-didnt-now-you-needed">Everything you
2 <h2><a href="https://klieret.github.io/everything-you-didnt-now-you-needed">Tips and tricks for python, the command line and more.</h2></p>
338 <p align="center"><a href="https://git-scm.com/book/en/v2/GitHub-Contributing-to-a-Project">
1 <a href="https://github.com/klieret/everything-you-didnt-now-you-needed/actions">1</sup> to reload all (!) modules every time you execute code

```
1 %load_ext autoreload
2 %autoreload 2 # reload everything
```

- **More granular:**

```
1 %load_ext autoreload
2 %autoreload 1 # <-- reload only some modules
3
4 # Mark for reloading
5 %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](#)

<sup>1</sup>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

# 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: 0.6.1
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 ...
```

# Static code checkers and Jupyter notebooks

- **!?** **Problem:** I still have lots of code in my notebooks. I still want to apply tools on notebooks.
- 💡 **Solution:** Black and Ruff support jupyter notebooks natively!
- 💡 **Pro-tip:** `nbqa` allows other tools to be applied, as well.

```
1     # Reformat code without compromises!  
2 -   repo: https://github.com/psf/black  
3     rev: '23.7.0'  
4     hooks:  
5       - id: black-jupyter  
6     # Static checks  
7 -   repo: https://github.com/astral-sh/ruff-pre-commit  
8     rev: 'v0.0.278'  
9     hooks:  
10    - id: ruff  
11      types_or: [python, pyi, jupyter]  
12      args: [--fix, --show-fixes]
```

# Various tricks with Jupyter

- **!?** **Problem:** I want to preview/run Jupyter notebooks in my terminal.
- **💡 Solution:**
  - `!pipx run nbpreview`` if you're only interested in previewing
  - `!pipx run rich-cli`` also works
  - `!pipx run nbterm`` for interactively executing notebooks in the terminal

# Python

# 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: 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: Make testing fun

## Basics

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

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

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

## First tip: your `project.toml` file

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

# 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://learn.scientific-python.org/development/guides/pytest> is a great place to look for tips!

# pytest: Make testing fun

Fixtures allow reuse, setup, etc

## 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"
```

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

```
1 @pytest.fixture
2 def my_complex_object():
3     mco = MyComplexObject(...)
4     mco.xyz = "asf"
5     ...
6     return mco
7
8 def test_get_value(my_complex_object):
9     assert my_complex_object.get_value() == ...
10
11 def test_other_property(my_complex_object):
12     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".

# 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.

```
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? (requires `pipx install mypy``)

```
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).

👉 Example files available [here](#).

# 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(self) -> str:
5          ...
6
7  def pester_duck(a_duck: Duck) -> None:
8      print(a_duck.quack())
9
10 class MyDuck:
11     def quack(self) -> str:
12         return "quack"
13
14 # Check explicitly that MyDuck is implementing the Duck protocol
15 if typing.TYPE_CHECKING:
16     _: Duck = typing.cast(MyDuck, None)
```



# Type checking (pre-commit)

```
1 - repo: https://github.com/pre-commit/mirrors-mypy
2   rev: "v1.4.1"
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)

# GitHub Actions: pages deploy

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

```
1  on:
2    workflow_dispatch:
3    pull_request:
4    push:
5
6  permissions:
7    contents: read
8    pages: write
9    id-token: write
10
11 concurrency:
12   group: ${{ github.workflow }}-${{ github.ref }}
13   cancel-in-progress: true
```

```
1  jobs:
2    build:
3      runs-on: ubuntu-latest
4      steps:
5        - uses: actions/checkout@v3
6        - name: Setup Pages
7          id: pages
8          uses: actions/configure-pages@v3
9
10       # Static site generation, latex, etc. here
11
12        - name: Upload artifact
13          uses: actions/upload-pages-artifact@v2
14          with:
15            path: dist/
16
17    deploy:
18      if: github.ref == 'refs/heads/main'
19      needs: build
20      environment:
21        name: github-pages
22        url: ${{ steps.deployment.outputs.page_url }}
23      runs-on: ubuntu-latest
24      steps:
25        - name: Deploy to GitHub Pages
26          id: deployment
27          uses: actions/deploy-pages@v2
```

# 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
def iter_last(values: Iterable[T]) -> Iterable[Tuple {
  """Iterate and generate a tuple with a flag fo
  iter_values = iter(values)
  try:
    previous_value = next(iter_values)
  except StopIteration:
    return
  for value in iter_values:
    yield False, previous_value
    previous_value = value
  yield True, previous_value

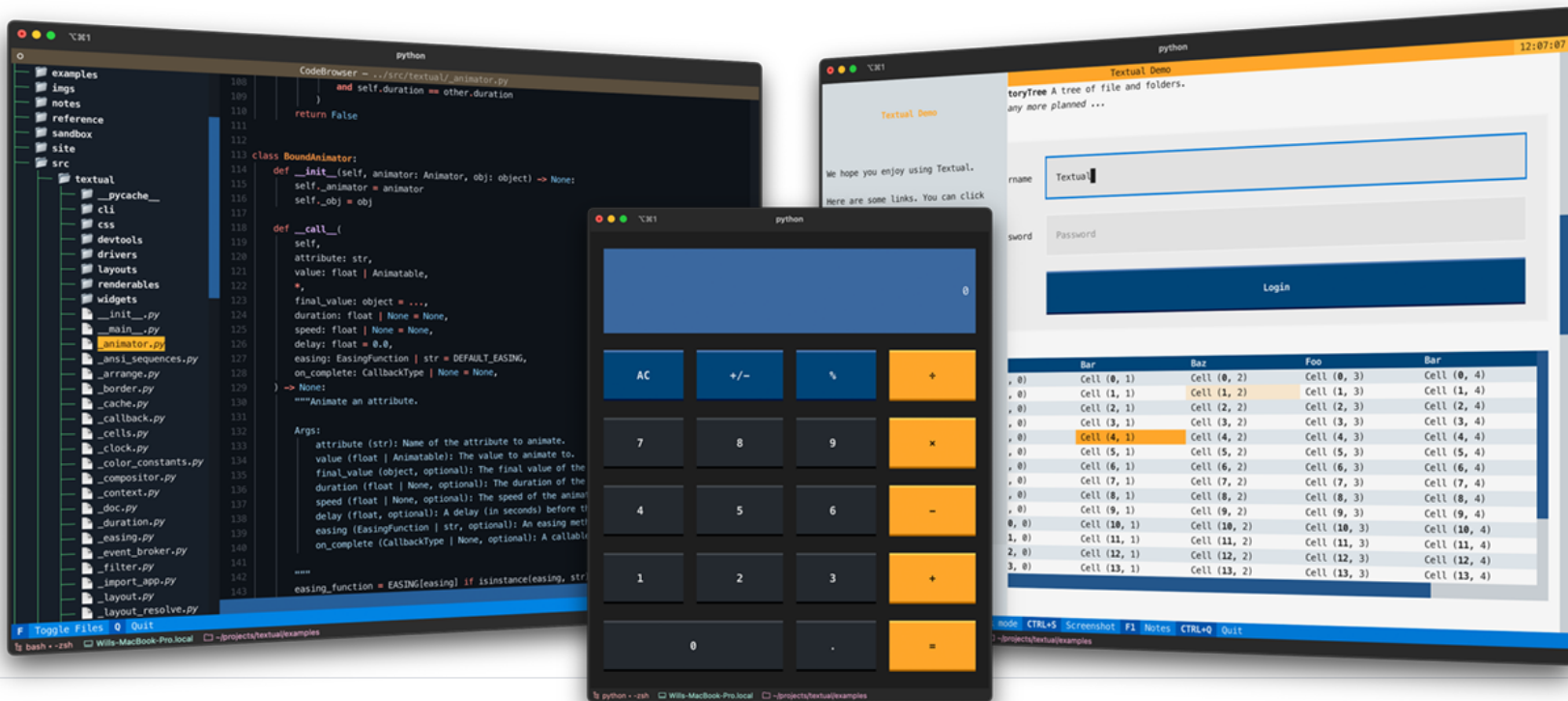
Markdown
# Markdown
Supports much of the *markdown* _syntax_!
- Headers
- Basic formatting: **bold**, *italic*, `code`
- Block quotes
- Lists, and more...

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!



for 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.
-l --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.
-e --expand       Enable soft wrapping of text (requires --print).
-w --width SIZE  Expand to full width (requires --panel).
-W --max-width SIZE
                  Fit output to SIZE characters.
-s --style STYLE  Set maximum width to SIZE characters.
                  Set text style to STYLE.
-S --rule-style STYLE
                  Set rule style to STYLE.
-r --rule-char CHARACTER
                  Set CHARACTER to generate a line with --rule.
-d --padding TOP,RIGHT,BOTTOM,LEFT
                  Use CHARACTERS to generate a line with --padding.
-a --panel BOX   Padding around output. 1, 2 or 4 comma separated integers, e.g. 2,4
                  Set panel type to BOX. ascii, ascii2, double, heavy, none, rounded,
                  square
                  Set the panel style to STYLE (requires --panel).
-S --panel-style STYLE
                  Set panel style to STYLE (requires --panel).
-n --theme THEME Set syntax theme to THEME. See https://pygments.org/styles/
-g --guides      Enable line number in syntax.
-x --lexer LEXER Use indentation guides in syntax highlighting. See
                  https://pygments.org/docs/lexers/
-y --lexer LEXER Use LEXER for syntax highlighting. See
                  https://pygments.org/docs/lexers/
-o --lexer LEXER Render hyperlinks in markdown.
                  Don't word wrap syntax highlighted files.
--hyperlinks     Don't word wrap syntax highlighted files.
--no-wrap        Set panel title to TEXT.
--title TEXT     Set panel caption to TEXT.
--caption TEXT   Set panel title when not writing to a terminal.
--force-terminal Force terminal output when not writing to a terminal.
--export-html PATH
                  Write HTML to PATH.
--help          Show this message and exit.

```

# 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!

## Pyodide

A distribution of CPython 3.11 including ~100 binary packages like SciPy, Pandas, boost-histogram (Hist), etc.

Examples:

- <https://learn.scientific-python.org/development/guides/repo-review>

## 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/latest/pyscript.css" />
8    <script defer src="https://pyscript.net/latest/pyscript.js"></script>
9  </head>
10 <body>
11   <script type="py-script">
12     print("Hello, World!")
13   </script>
14 </body>
15 </html>
```

<https://docs.pyscript.net/latest/tutorials/getting-started.html>

# 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://learn.scientific-python.org/development/guides/packaging-simple/>
- <https://packaging.python.org/en/latest/tutorials/packaging-projects>

`scientific-python/copier` supports 12 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:** I've been working on making it easy!

# 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 funded to work on scikit-build for three years!  
CMake for Python packaging.

Org of several packages:

- Scikit-build
- Scikit-build-core
- 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@v3
15
16      - name: Build wheels
17        uses: pypa/cibuildwheel@v2.14
18
19      - uses: actions/upload-artifact@v3
20        with:
21          path: ./wheelhouse/*.whl
```

# CMake example

You can make a working example with just three files!

## CMakeLists.txt

```

1  cmake_minimum_required(VERSION 3.15...3.26)
2  project(${SKBUILD_PROJECT_NAME} LANGUAGES CXX)
3
4  set(PYBIND11_FINDPYTHON ON)
5  find_package(pybind11 CONFIG REQUIRED)
6
7  pybind11_add_module(_core MODULE src/main.cpp)
8  install(TARGETS _core DESTINATION ${SKBUILD_PROJECT_NAME})

```

## pyproject.toml

```

1  [build-system]
2  requires = ["scikit-build-core", "pybind11"]
3  build-backend = "scikit_build_core.build"
4
5  [project]
6  name = "package"
7  version = "0.1.0"

```

## src/main.cpp

```

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

```