The Tools of Data-Driven Engineering

Handout Edition

Niels Skovgaard Jensen

Handout README

This is the handout version of the lecture slides.

- Most references to tools have links to their respective websites.
- There are links to all slides in the Contents slide
- I reference a lot of different tools so you can investigate them yourself. My recommended tools are usually marked in **bold**.

My hope for this lecture:

- Show you where to look for good SciML development practices
- Get you thinking about what fits your use-cases
- Give you a heads-up of what is expected in industry

What you **should** do after this lecture:

- Investigate tools and reconsider your workflows
- Understand that a little work up front can make a lot of things easier

What you **should not** do:

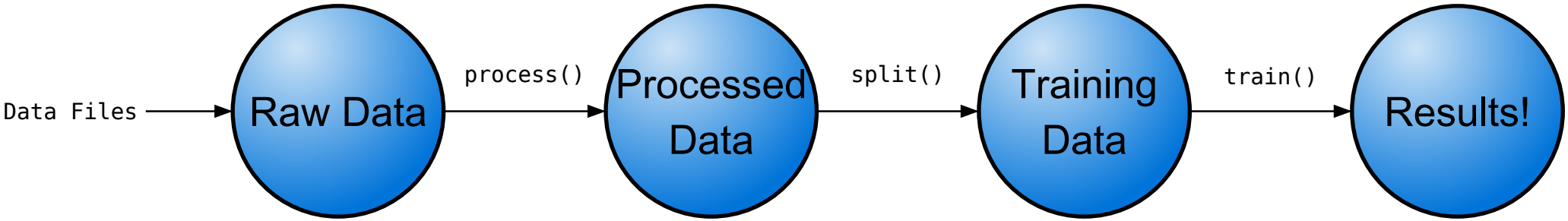
- Panic that you don't use *all* these tools
- Spend all your time trying to learn *all* these tools

Tools are supposed to make your life easier
(*But it will take a little work and interest*)

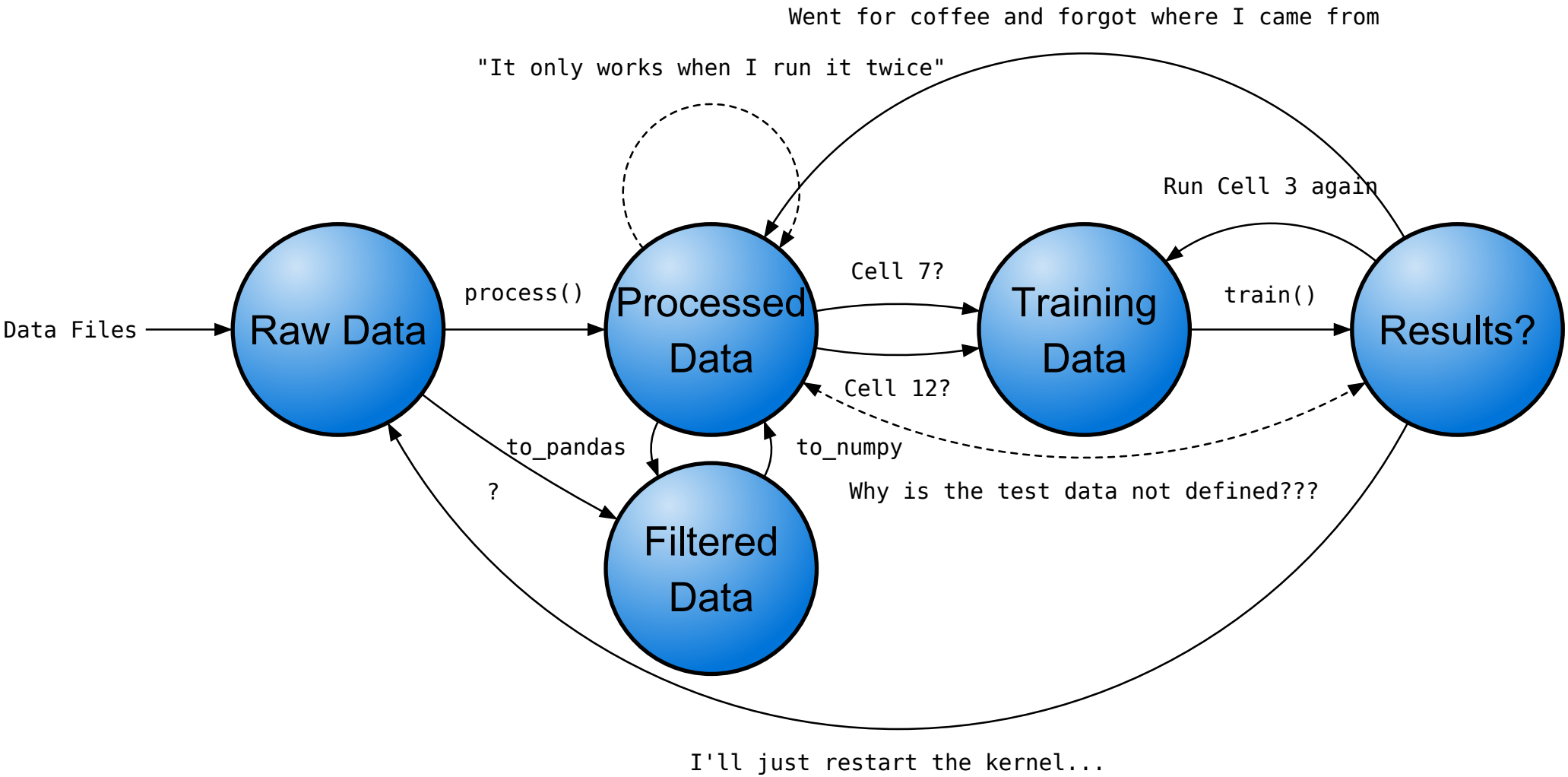
Contents

Analysis as a DAG

Each Step Should Only Depend on the Previous



How Notebooks Often Look



How to ensure Acyclicity

- Prefer scripts or library code for things you want reproduced
  - Or use reproducible notebooks such as Marimo
- Consider raw data *immutable*
- Design a data pipeline into your project

```
├── data
│   ├── interim   <- Intermediate data that has been transformed.
│   ├── processed <- The final, canonical data sets for modeling.
│   └── raw       <- The original, immutable data dump
```

- TIP: You can use command-run files, such as *make* or *just*, to string together processing and experimentation scripts.

Environment Reproducibility

Reproducibility in CompSci

- We are in one of the few sciences with almost entirely controlled experimentation
  - Reproducibility is (relatively) easy
- SciML is a very experiment-driven field
  - Reproducibility is harder
  - But much more important

Python Environments

There are many ways to use python and install packages:

- pip, venv, conda, **uv**
  - uv is quickly becoming the modern industry standard
- Reproducibility when using uv
  - pyproject.toml – Defines requirements
  - uv.lock – locks requirements
  - uv creates isolated environments per project
- uv makes it easy to:
  - Share and reproduce environments across computers
  - Ensure you are using the right environment when running code
- It is also *really* fast

Basic uv commands

Create a new project

```shell
uv init --lib # Create a library
uv init --script # Standalone script
```

Add a dependency

```shell
uv add numpy #numpy, torch, jax, etc...
```

Run a file

```shell
uv run myscript.py # Uses the venv uv has created
```

Reproduce the project

```shell
uv sync # Reproduce the environment on a new computer
```

Reuse code when possible

- uv makes it easy to create a shared library of code

```shell
uv init --lib my_library
```

- This creates a folder structure like:

```
my_library/
├── pyproject.toml        # Project configuration file
└── src
    └── my_library        # Library source code
        ├── __init__.py
        └── hello.py      # Example module
```

Which can then be imported into a script

my_script.py

```python
from my_library.hello import hello
```

Share code in libraries, use the code in scripts

```
awesome_sciml_project/
├── pyproject.toml        # Project configuration file
└── src
    └── my_library        # Library source code
        ├── __init__.py
        └── data.py      # Example module
        └── train.py
        └── models.py
├── data/
    └── processed
    └── interim
    └── raw
├── scripts/ # Create seperate scripts with some agreed-on naming scheme
    └── 1.0-nsj-first_visualisation.py
    └── 2.0-apek-fit_data.py
```

For more inspiration: [Cookiecutter Datascience Template]

Coding Practices

Linting and Formatting

linting – Automated coding style guides

formatting – Automated code aesthetics

- Modern Python has converged on a style
- It is much easier to read consistently styled-code
- Formatting tools make it easy to follow the style!

The industry standard formatter/linter for python is Ruff

Formatting Example

ugly_code.py Python

```python
import numpy as np
import torch
def foo(
    x,y ,
    z, u, items=[]
):
    q = 2; t = np.array([1, 2, 3]); z = 4
    assert (x > 0, "must be positive")
    if x == None or y == True:
        return x -        y
    if u > 0:
        return  y+z
    else:
        return x+ y
if __name__ == "__main__":
    result = foo(1,  2  ,3,
     4)
    print("result:",   result   )
```

Shell

```
uvx ruff format ugly_code.py
```

formatted.py Python

```python
import numpy as np


def foo(x, y, z, u, items=[]):
    q = 2
    t = np.array([1, 2, 3])
    z = 4
    assert (x > 0, "must be positive")
    if x == None or y == True:
        return x - y
    if u > 0:
        return y + z
    else:
        return x + y


if __name__ == "__main__":
    result = foo(1, 2, 3, 4)
    print("result:", result)
```

Linting Example

formatted.py Python

```python
import numpy as np


def foo(x, y, z, u, items=[]):
    q = 2
    t = np.array([1, 2, 3])
    z = 4
    assert (x > 0, "must be positive")
    if x == None or y == True:
        return x - y
    if u > 0:
        return y + z
    else:
        return x + y


if __name__ == "__main__":
    result = foo(1, 2, 3, 4)
    print("result:", result)
```

Shell

```
uvx ruff check formatted.py
```

Shell

```
LOTS OTHER ERRORS
...

E712 Avoid equality comparisons to `True`; use `y:` for truth checks
  --> unformatted.py:8:21
   |
 6 |     q = 2; t = np.array([1, 2, 3]); z = 4
 7 |     assert (x > 0, "must be positive")
 8 |     if x == None or y == True:
   |                      ^^^^^^^^^
 9 |         return x -        y
10 |     if u > 0:
   |
help: Replace with `y`

Found 7 errors.
No fixes available (4 hidden fixes can be enabled with the `--unsafe-fixes` option).
```

Python type-hinting


```python
x: float = 1.0
y: np.ndarray = np.array([2.0])
t: Tensor = torch.tensor([2.0])

def foo(x: np.ndarray) -> float:
  return float(x.item())
my_func: Callable = foo
```

Python is a *dynamically* typed language.

In Python, type hints are structured comments (they do not affect runtime). But they can be analyzed by external software

This allows for *static type checking* – A powerful tool to catch bugs before you run your code

Static Type Checkers

- Examples: **mypy**, pyright, **basedpyright**, (ty, pyrefly – In Beta)
- Can easily be run in your IDE
- Use uv tool (uvx) to experiment with them:

```shell
uvx mypy check <my_code>.py
```

```shell
uvx basedpyright check <my_code>.py
```

```shell
uvx ty check <my_code>.py
```

Static Type Checker Example

static_broken.py Python

```python
def add(a: int, b: int) -> int:
    return a + b

def greet(name: str) -> str:
    return "Hello, " + name

result = add("hello", 5)
message = greet(42)
```

Shell

```
uvx mypy static_broken.py
```

Text

```
typing_example/static_broken.py:7: error: Argument 1 to "add" has incompatible type "str"; expected "int"  [arg-type]
typing_example/static_broken.py:8: error: Argument 1 to "greet" has incompatible type "int"; expected "str"  [arg-type]
Found 2 errors in 1 file (checked 1 source file)
```

Caught **before** running the code!

How it can look in your IDE

static_broken.py Python

```python
def add(a: int, b: int) -> int:
    return a + b

def greet(name: str) -> str:
    return "Hello, " + name

result = add("hello" | expected int, 5)
message = greet(42 | expected str)
```

Dataclasses

Python

```python
from dataclasses import dataclass

# Init example from python documentation
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart
z = Complex(1.0,1.0)

# Dataclass equivalent
@dataclass
class Complex:
    r: float
    i: float
z = Complex(1.0,1.0) # Create in the same way

@dataclass
class Dataset: # They are excellent for packing data
    mat: jnp.array
    coordinate: list[float]
    desc: str
    ... # Whatever your heart desires
```

Organising Configuration in Dataclasses

train.py Python

```python
@dataclass
class TrainingConfig:
    model: torch.nn.Module
    train_data: Data
    test_data: Data
```

train.py Python

```python
@dataclass
class TrainingResult:
    model_final: torch.nn.Module
    train_loss: list[float]
    test_loss: list[float]
```

train.py Python

```python
config = TrainingConfig(model = ..., ... )

def train(config: TrainingConfig) -> TrainingResult:
    ...
    return TrainingResult(
        model_final=model,
        train_loss=train_loss,
        ...
    )
result = train(config)
result.model_final # Access trained model or other results easily

result2 = train(config2) # Easy to keep track of multiple runs!
```

Using classmethods as a flexible `__init__`

Python

```python
@dataclass
class Data:
    mat: torch.Tensor
    R: torch.Tensor
    # name, version, metadata... Whatever

    @classmethod
    def from_file(cls, path: Path) -> 'Data':
        # load data from file
        mat, R = load_file(path)
        return cls(mat = mat, R = R)

    @classmethod
    def from_numpy(cls, array: np.ndarray) -> 'Data':
        # convert numpy array to Data instance
        mat = torch.from_numpy(array)
        R = torch.eye(mat.size(1))  # Example: identity matrix as R
        return cls(mat = mat, R = R)
```

Dataclasses in Python – Structs in Julia

- Julia has built-in type system
- Types are enforced at runtime
- Static typing is not as mature as Python
- Dataclasses are (roughly) equivalent to Julia structs
- Multiple-dispatch with structs is one of Julia's superpowers

Making Your Scripts Reconfigurable

- Bad Practice: Changing code to alter variables
  - Each experiment changes code state
- Good Practice: Making your scripts reconfigurable
  - Save the command, and reproduce your result in the future
- Tip: LLM Coding tools are excellent at making your scripts reconfigurable.

Non-Reconfigurable

Python

```python

@dataclass
class TrainConfig:
    lr: float = 1e-3
    epochs: int = 1000
    batch_size: int = 32

def train(config: TrainConfig):
    ...

if __name__ == "__main__":
    config = TrainConfig()
    train(config)
```

Reconfigurable

```python
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--lr", type=float, default=0.001)
parser.add_argument("--epochs", type=int, default=10)
parser.add_argument("--batch-size", type=int, default=32)

... # Same as previous example
if __name__ == "__main__":
    args = parser.parse_args()
    config = TrainConfig(lr=args.lr, epochs=args.epochs, batch_size=args.batch_size)
    train(config)
```

Now you can change the script when running it

```shell
uv run <myscript.py> --lr 0.002 --epochs 100 --batch-size 64
```

Advanced Tool: Hydra

Version Control

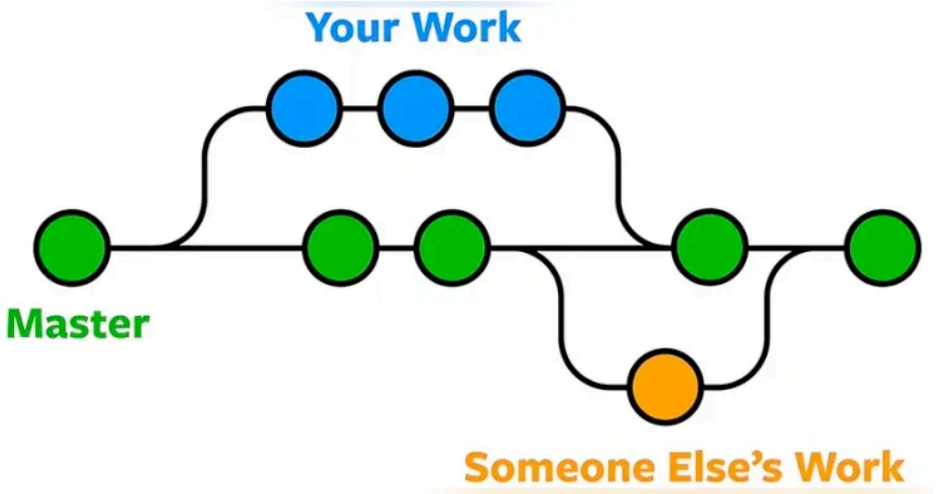
Git – Really quick

Git is a distributed version-control system.

- Enables collaboration
- And *restoration* of code state

Each change is made in a *commit*

Multiple chained *commits* form *branches*



Git is your friend

External git services:

- GitHub
- GitLab
- *Makes collaboration easy*

Remember: Git $\neq$ github

Local git repos are also very useful

Advanced Tool: Jujutsu

GUIs for GIT

- GitGraph in VSCode – Free GUI for using git
- GitHub Desktop – Free GUI for using GitHub
- Smartgit

But most power comes from the command line

Git for Data-Driven Engineering

Some pragmatic guidelines:

- Use commits to restore code state
- When something works – commit
- When you have a result – commit
- Before you experiment – commit
- Forgot to commit before experimenting? Use `git stash`
  - Saves your uncommitted changes and restores a clean state
  - `git stash pop` to bring them back

.gitignore – What not to Commit

- Git tracks *everything* by default – you must tell it what to ignore
- Create a `.gitignore` file in your project root:

```
__pycache__/        # Python cache files
.venv/              # Virtual environment
data/               # Large datasets
*.pt                # PyTorch model checkpoints
*.ckpt              # Other checkpoints
.env                # API keys and secrets
wandb/              # Experiment logs
```

Pre-Commit

Pre-commit is a tool to make sure everyone follows the same linter/ formatter guidelines.

Example: Before each commit run:

- Linting
- Formatting
- uv sync (To make uv.lock)

Pre-commit ensures a commit cannot be made if the tools fail

It is especially useful for collaboration

Advanced: CI/CD pipelines.z

Exercises

Exercises

Like the lecture, these exercises are made to give an introduction to the tools that have been discussed. Thus you are free to choose and experiment for yourself. The goal is to create a forum for discussing how we work, use tools and do the best possible SciML.

Comprehension questions: Discuss these questions in groups

- What is meant by treating your analysis as a DAG? Why is this desirable?
- Why should raw data be treated as immutable?
- What is the difference between a lockfile and a requirements.txt?
- What is the difference between *linting* and *formatting*?
- Which style of typing does Python have, and what is the advantage, what is the disadvantage?
- Which types of errors can static type checking fix?
- What is a dataclass, and why is it useful for organizing experiment configurations and data?
- Why is it better to make scripts reconfigurable (e.g. with argparse) rather than editing variables in the code?
- What is git, and how does it aid in reproducibility?
- Why is it useful to commit before experimenting with your code?

The following exercises all assume that `uv` is installed on your computer. All other used tools are installed as `uv` tools through the `uvx` command.

If you use another programming language, spend your effort on understanding how you can do these things in your language to replicate the effects and coding standards.

If you are a tool wizard and all this seems trivial, there is a list of more advanced tools at the buttom you can investigate.

Installing uv Guide:

https://docs.astral.sh/uv/getting-started/installation/

Note that you can use conda to install uv if you do not want to use the standalone installer.

```shell
conda install conda-forge::uv
```

2.1 – Environment

uv documentation: https://docs.astral.sh/uv/

1. Create a new uv project with a src library

```shell
uv init --lib myproject
```

2. Use `cd myproject` to go into the folder and read the pyproject.toml
3. Add a dependency of your choice (e.g. numpy)

```shell
uv add numpy
```

4. Read the pyproject.toml again and notice the changes. Also investigate the uv.lock file.
5. Write a small script that imports your added dependency and run it with `uv run`

2.2 – Formatting and Linting

Ruff documentation: https://docs.astral.sh/ruff/

1. Find a short piece of code you have written previously
2. Run the ruff formatter on it and observe the changes

```shell
uvx ruff format <my_code>.py
```

3. Run the ruff linter and try to understand the warnings it gives (if any)

```shell
uvx ruff check <my_code>.py
```

4. Try fixing the linting errors, either manually or with the `--fix` flag

2.3 – Typing

Python typing documentation: https://docs.python.org/3/library/typing.html

1. Find a piece of code you have previously written that does not have type hints.
2. Type-hint the code as best you can
3. Run mypy on the type-hinted code to see what errors come up:

```shell
uvx mypy <your_code>.py
```

4. Try to understand the errors (if any) and think about what types of bugs they might introduce, then fix all the errors.
5. (Extra) Now run the stricter basedpyright type checker and investigate and fix the errors it points to.

```shell
uvx basedpyright <your_code>.py
```

2.4 – Git

1. Initialize a git repository in your project folder

```shell
git init
```

2. Add your files and make an initial commit

```shell
git add .
git commit -m "Initial commit"
```

3. Make a small change to your code (e.g. add a print statement)
4. Use `git diff` to see what changed, then stage and commit the change

```shell
git add <my_code>.py
git commit -m "Describe the change"
```

5. Use `git log` to view your commit history
6. Create a repository on GitHub and push your project to it

```shell
# Create a repo on github.com, then:
git remote add origin https://github.com/<your-username>/<your-repo>.git
git push -u origin main
```

7. (Extra) Create a new branch, make a change on it, and merge it back into main

```shell
git checkout -b experiment
# make changes and commit
```

```
3  git checkout main
4  git merge experiment
```

2.5 – Reproducibility

1. Download the code that was made during the lecture demonstration on GitHub using the GitHub CLI, GitHub Desktop or git directly.
2. Run `uv sync` to reproduce my environment
3. Run my example script with `uv run`

This is how easy reproducibility should be!

2.6 – Investigate Your IDE

Most of the tools you have just used have plugins for VS Code and many other IDEs, investigate these plugins and how to use them.

Advanced Tools

If all of these excercies seem trivial and you are already a tool wizard. Then here are some advanced tools you can look into that are great for more complicated projects and users.

Version Control: Jujutsu A git-compatible version control system with superpowers. (for an amazing Jujutsu TUI: jjui)

Argparsing: Hydra Create configurable and reproducible experiemnts directly from your python dataclasses.

einops – Self documenting tensor operations in Jax, Torch and Numpy For a deep-dive on einstein-like notation: https://grrddm.notion.site/Einsums-in-the-wild-bd773f01ba4c463ca9e4c1b5a6d90f5f?v=4b86eb72820943c083c84c2674f67ac4

Jaxtyping: Runtime type checking for tensor shapes in Jax, Torch, Numpy and more....

Typst: An amazing LaTeX and Overleaf alternative.

Oxidise Your Terminal: A list of really cool Rust-based terminal tools https://www.youtube.com/watch?v=rWMQ-g2QDsI&themeRefresh=1 Honorable mentions: fish, zoxide, fzf, eza,

Writing Python like it is Rust: https://kobzol.github.io/rust/python/2023/05/20/writing-python-like-its-rust.html A cool perspective on how to write maintainable and well-typed Python code.