

Writing Robust & Reproducible Data Science Code

by Boje Deforce



Overview

- **Introduction**
- **Designing Clean Code with Reusable Structure**
 - Object-Oriented Programming for DS
 - Abstract Base Classes
- **Readable, Safe, and Maintainable Code**
 - Type-hinting with beartype
 - Google-style docstrings
 - Ruff: automated code linting & formatting
 - Unit testing
- **Experimentation & Configuration**
 - Hydra for structured config management
 - Hatch for reproducible environments
- **Handling Data at Scale**
 - Efficient dataframes with Polars
 - Reading/Writing from the cloud (fsspec, s3fs, ...)
- **Bridging Notebooks & Scripts**
 - Jupyter for collaborative reproducibility

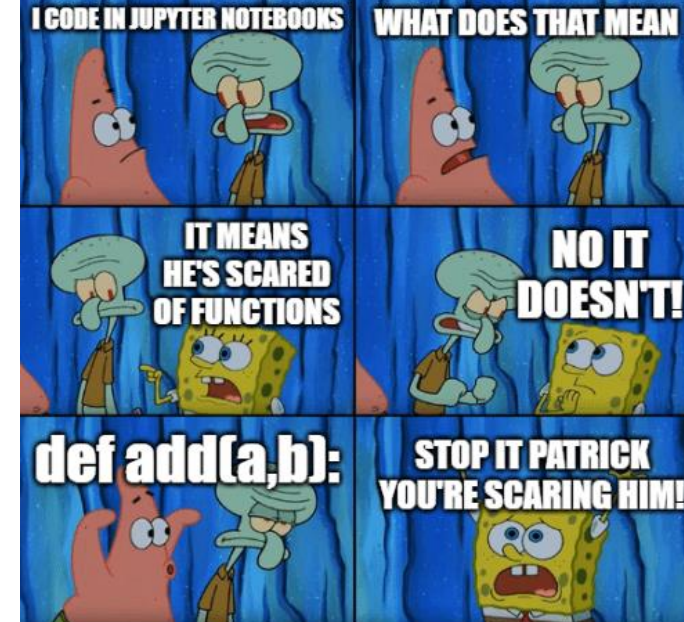
Overview

- **Introduction**
- **Designing Clean Code with Reusable Structure**
 - Object-Oriented Programming for DS
 - Abstract Base Classes
- **Readable, Safe, and Maintainable Code**
 - Type-hinting with beartype
 - Google-style docstrings
 - Ruff: automated code linting & formatting
 - Unit testing
- **Experimentation & Configuration**
 - Hydra for structured config management
 - Hatch for reproducible environments
- **Handling Data at Scale**
 - Efficient dataframes with Polars
 - Reading/Writing from the cloud (fsspec, s3fs, ...)
- **Bridging Notebooks & Scripts**
 - Jupyter for collaborative reproducibility

Introduction

- Data science projects often start messy, but they don't have to stay that way.
- Structuring your code improves **reusability**, **collaboration**, and **debugging**.
- Good design helps you focus on what actually matters: the **science** and the **insight**.
- Reproducibility is **not optional**, especially in business or highly regulated environments.

Our goal today: *learn best practices to help you manage/write better code*



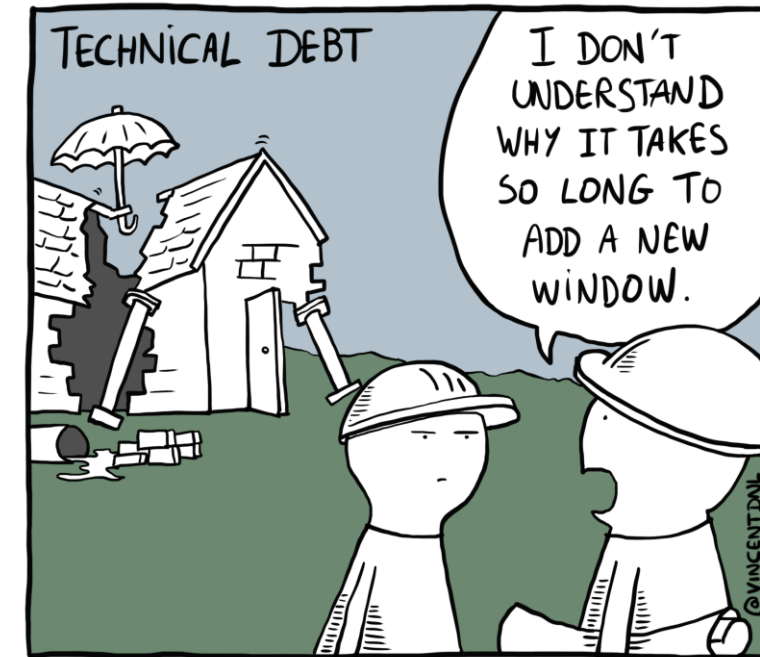
Introduction

Can Developers Embrace “Vibe Coding” Without Enterprise Embracing AI Technical Debt?

How to Manage Tech Debt in the AI Era

Moderne raises \$30M to solve technical debt across complex codebases

Great, but even better to avoid in the first place.

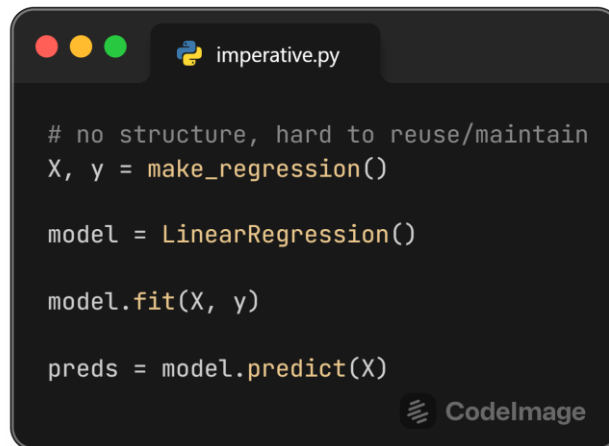


Overview

- **Introduction**
- **Designing Clean Code with Reusable Structure**
 - Object-Oriented Programming for DS
 - Abstract Base Classes
- **Readable, Safe, and Maintainable Code**
 - Type-hinting with beartype
 - Google-style docstrings
 - Ruff: automated code linting & formatting
 - Unit testing
- **Experimentation & Configuration**
 - Hydra for structured config management
 - Hatch for reproducible environments
- **Handling Data at Scale**
 - Efficient dataframes with Polars
 - Reading/Writing from the cloud (fsspec, s3fs, ...)
- **Bridging Notebooks & Scripts**
 - Jupyter for collaborative reproducibility

Designing Clean Code with Reusable Structure

- Why?
 - Avoid rewriting pipelines for every project (cf., DRY principle)
 - Make it easy to switch models or data sources
 - Think in interfaces, not implementations
 - Clean structure = easier testing, debugging, and scaling.



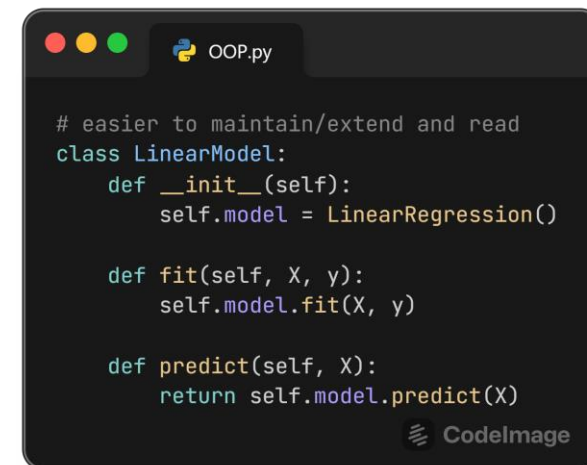
```
# no structure, hard to reuse/maintain
X, y = make_regression()

model = LinearRegression()

model.fit(X, y)

preds = model.predict(X)
```

CodeImage



```
# easier to maintain/extend and read
class LinearModel:
    def __init__(self):
        self.model = LinearRegression()

    def fit(self, X, y):
        self.model.fit(X, y)

    def predict(self, X):
        return self.model.predict(X)
```

CodeImage

Intermezzo: Two styles of writing code

Imperative Programming

- Step-by-step instructions: “do this, then that”
- Code is ordered like a script
- Harder to reuse, extend, or test

Object-Oriented Programming (OOP)

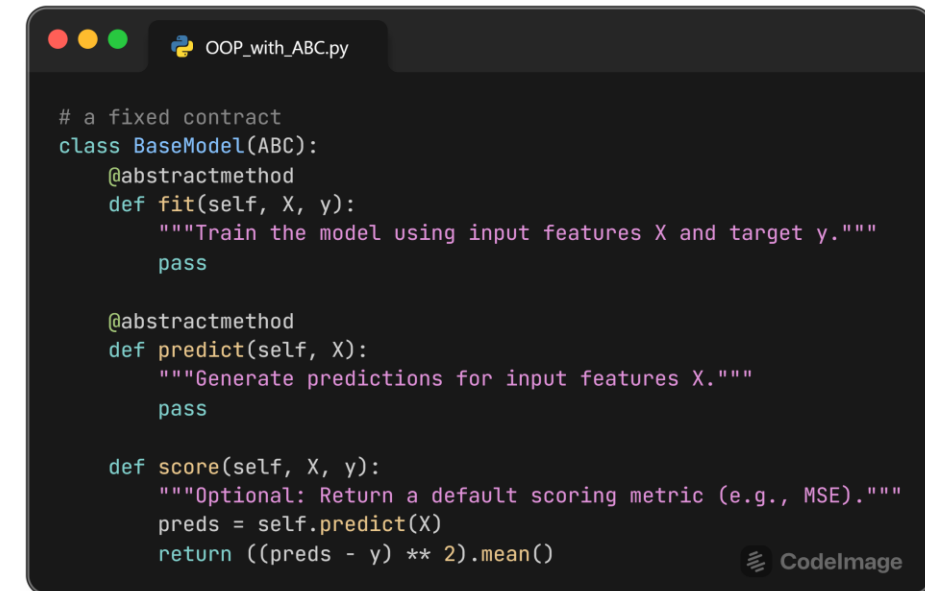
- Bundle behavior into objects
 - Define reusable components
 - Easier to test, extend, and maintain
-
- ✓ You can swap models without rewriting your pipeline
 - ✓ You can test behavior in isolation
 - ✓ You structure assumptions in code

Basic Object-Oriented Programming Concepts

Term	Definition
Class	A blueprint for creating objects (e.g., Model, Dataset)
Object	An instance of a class – contains attributes and methods
Method	A function defined inside a class
Attribute	A variable attached to an object (e.g., self.weights)
Inheritance	One class inherits attributes/methods from another
Encapsulation	Wraps data and methods that work on the data within one unit, called a class
Polymorphism	Different objects can be used interchangeably if they share the same interface (e.g., model_1.predict(), model_2.predict())

Designing Clean Code with Reusable Structure

- Fragmentation: we end up with *many similar* versions of classes
- We need a **fixed contract** that all our models need to adhere to
- Insert **Abstract Base Classes (ABCs)**:
 - Safety: Forces you to implement required methods
 - Reusability: You can plug models into shared code (e.g.,: training loops, validators)
 - Readability: Defines expected behavior up front – no guessing



```
OOP_with_ABC.py

# a fixed contract
class BaseModel(ABC):
    @abstractmethod
    def fit(self, X, y):
        """Train the model using input features X and target y."""
        pass

    @abstractmethod
    def predict(self, X):
        """Generate predictions for input features X."""
        pass

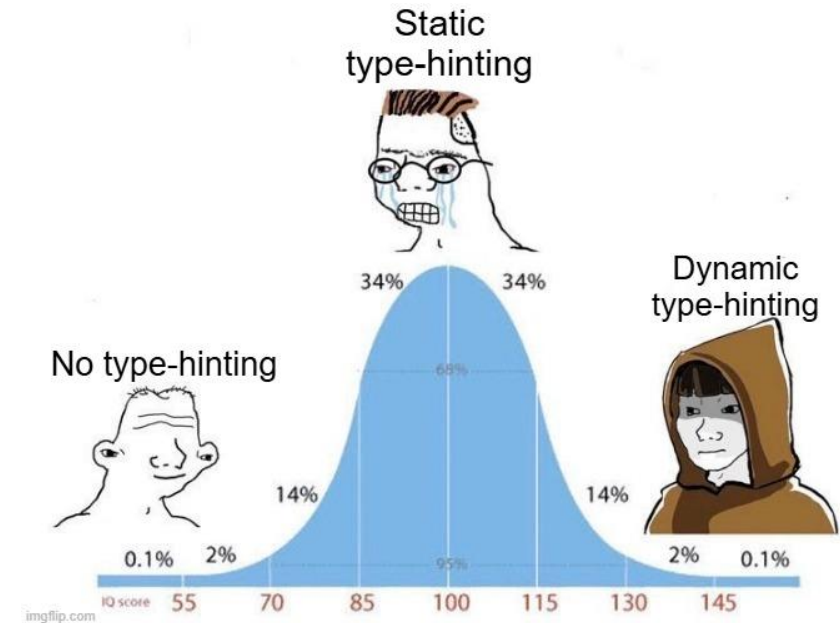
    def score(self, X, y):
        """Optional: Return a default scoring metric (e.g., MSE)."""
        preds = self.predict(X)
        return ((preds - y) ** 2).mean()
```

Overview

- Introduction
- Designing Clean Code with Reusable Structure
 - Object-Oriented Programming for DS
 - Abstract Base Classes
- **Readable, Safe, and Maintainable Code**
 - Type-hinting with beartype
 - Google-style docstrings
 - Ruff: automated code linting & formatting
 - Unit testing
- Experimentation & Configuration
 - Hydra for structured config management
 - Hatch for reproducible environments
- Handling Data at Scale
 - Efficient dataframes with Polars
 - Reading/Writing from the cloud (fsspec, s3fs, ...)
- Bridging Notebooks & Scripts
 - Jupyter for collaborative reproducibility

Type-hinting

- Why?
 - Prevent bugs early
 - (before running your code all night)
 - Make code more readable + IDE-friendly
 - Help collaborators understand your code



Type-hinting

Static type checkers (e.g. mypy)

- Analyze code before it runs
- Catch common mistakes early
- Integrates with IDEs

What about:

- Runtime?
- Notebooks?
- Validate input data?



Dynamic type checkers (e.g. beartype)

- Enforces type hints at runtime
- Simply to use with decorators
- Raises exceptions when inputs or outputs are wrong (e.g., jaxtyping)

```
beartype.py

@beartype
class LinearModel:
    def __init__(self):
        self.model = LinearRegression()

    def fit(self, X: np.ndarray, y: np.ndarray) → None:
        self.model.fit(X, y)

    def predict(self, X: np.ndarray) → np.ndarray:
        return self.model.predict(X)
```

Doc strings

- Why?
 - Explain what your function or class does
 - Show expected input/output types
 - Help others (and future you!) understand your code
 - Powers autocompletion and documentation tools
 - Pdoc, PyDoctor, ...



Pro tip: Write your docstring before you write the function. It forces you to think clearly.

(or have ChatGPT do it) 

Code Linting & Formatting

- **Formatting**: Makes your code look consistent (spacing, indentation, ...)
- **Linting**: Analyzes your code for errors, bad practices, and style violations
- Why?
 - Clean code is easier to read, review*, and debug
 - Linting catches mistakes early (like unused imports or untyped functions)
 - Consistency = fewer bugs & better collaboration
 - Helps enforce the PEP 8 style guide (Python's coding standard)

**code peer reviewing is/should be standard practice in all companies and is even in uprise in academia (e.g. AutoML 2025)*

Ruff: One tool to rule them all

- Written in Rust → blazing fast
- Lints + Formats + Sorts imports
- Can be integrated in IDE (e.g. VS Code)

Unit testing

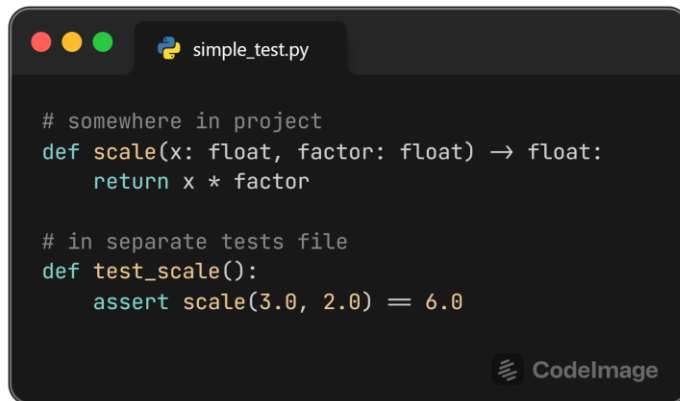
- **Unit testing**: A small test that checks *one specific behavior* of a function or method
- Why?
 - Catch bugs early and with precision
 - Catch unexpected behavior when you refactor
 - Make code safer to reuse across projects
 - Make code easier to review
 - Build trust in your models and metrics within org



Unit testing

Simple test

- Shapes
- Simple arithmetic
- ...



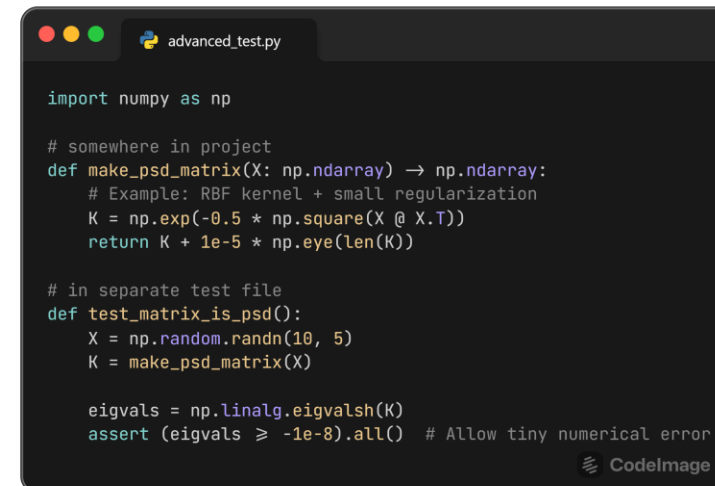
```
# somewhere in project
def scale(x: float, factor: float) → float:
    return x * factor

# in separate tests file
def test_scale():
    assert scale(3.0, 2.0) == 6.0
```

CodeImage

Advanced test

- Check scientific logic
- Check edge cases
- ...



```
import numpy as np

# somewhere in project
def make_psd_matrix(X: np.ndarray) → np.ndarray:
    # Example: RBF kernel + small regularization
    K = np.exp(-0.5 * np.square(X @ X.T))
    return K + 1e-5 * np.eye(len(K))

# in separate test file
def test_matrix_is_psd():
    X = np.random.randn(10, 5)
    K = make_psd_matrix(X)

    eigvals = np.linalg.eigvalsh(K)
    assert (eigvals ≥ -1e-8).all() # Allow tiny numerical error
```

CodeImage

Unit testing

- We scratched only the smallest surface
- Unit testing becomes easier with LLMs
- Many Python libs to help:
 - Hypothesis
 - Pytest
- Test-driven development: first write tests, then code
 - Does not always work well for data science

Quick recap

- We have seen how to:
 - Write clean reusable code
 - How to ensure your code is safe and reliable
- But now:
 - How do we run this efficiently?

Overview

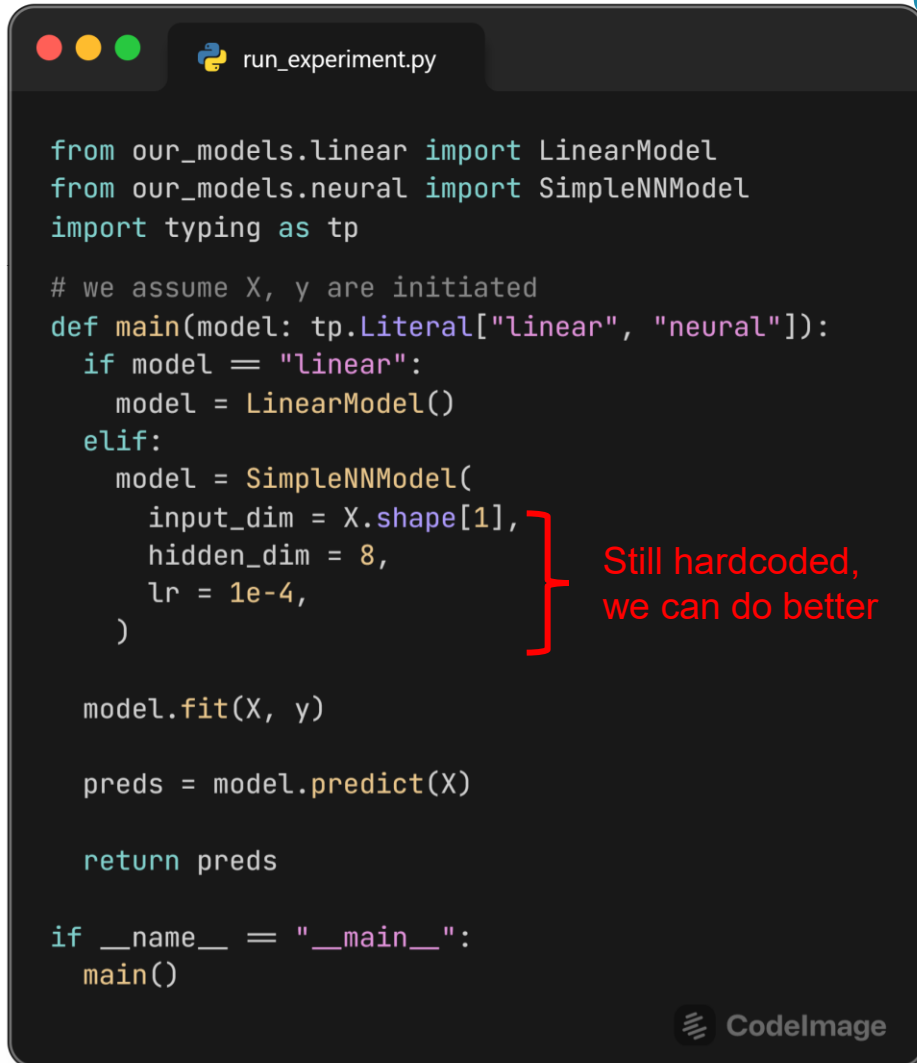
- **Introduction**
- **Designing Clean Code with Reusable Structure**
 - Object-Oriented Programming for DS
 - Abstract Base Classes
- **Readable, Safe, and Maintainable Code**
 - Type-hinting with beartype
 - Google-style docstrings
 - Ruff: automated code linting & formatting
 - Unit testing
- **Experimentation & Configuration**
 - Hydra for structured config management
 - Hatch for reproducible environments
- **Handling Data at Scale**
 - Efficient dataframes with Polars
 - Reading/Writing from the cloud (fsspec, s3fs, ...)
- **Bridging Notebooks & Scripts**
 - Jupyter for collaborative reproducibility

Intermezzo: Running code

- We could run everything in a notebook, but what about:
 - Running on the cloud
 - Running several experiments
 - Collaboration with engineers or MLOps
 - Reproducing your exact results 6 months later
 - ...



Intermezzo: Running code



```
run_experiment.py

from our_models.linear import LinearModel
from our_models.neural import SimpleNNModel
import typing as tp

# we assume X, y are initiated
def main(model: tp.Literal["linear", "neural"]):
    if model == "linear":
        model = LinearModel()
    elif:
        model = SimpleNNModel(
            input_dim = X.shape[1],
            hidden_dim = 8,
            lr = 1e-4,
        )

    model.fit(X, y)

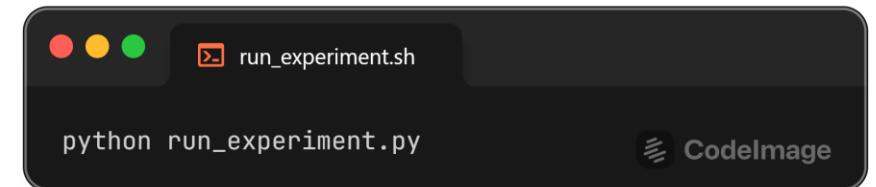
    preds = model.predict(X)

    return preds

if __name__ == "__main__":
    main()
```

Still hardcoded,
we can do better

CodeImage



```
run_experiment.sh

python run_experiment.py
```

CodeImage

Config management

```
run_experiment.py

from our_models.linear import LinearModel
from our_models.neural import SimpleNNModel
import typing as tp

with open("config.yaml", "r") as f:
    cfg = yaml.safe_load(f)

# we assume X, y are initiated
def main(config: tp.Dict[str, tp.Any]):
    if config["model"] == "linear":
        model = LinearModel()
    elif:
        model = SimpleNNModel(
            input_dim = X.shape[1],
            hidden_dim = config["hidden_dim"],
            lr = config["lr"],
        )

    model.fit(X, y)

    preds = model.predict(X)

    return preds

if __name__ == "__main__":
    main()
```

CodeImage

```
config.yaml

model: neural
hidden_dim: 8
lr: 0.0001
```

CodeImage

```
run_experiment.sh

python run_experiment.py
```

CodeImage



Config management with Hydra

- Why?

Manual YAML	Hydra
Manual file reading	Automatic config injection
Dict lookups everywhere	Type-safe <code>cfg.attr</code> access
CLI parsing is manual	CLI overrides built-in
No defaults/composability	Uses defaults lists

Config management with Hydra

Bringing it all together:

- Create our code files
- Create desired structure for config files as dataclass
- Create our config files
- Create main file
- Let Hydra tie it all together

Such that:

- Both code and configs are safe and readable
- We can easily track experiments
- A new person can easily understand code by reading it

See live example and [tutorials](#) in official docs

Environment: The final piece of reproducibility

- What can go wrong:
 - Different Python versions → different model behavior
 - One library update → silent API change
 - A model runs on your laptop but crashes on someone else's

Same code, same config, same data—but without the right environment, you won't get the same results.





Hatch for reproducible, isolated envs

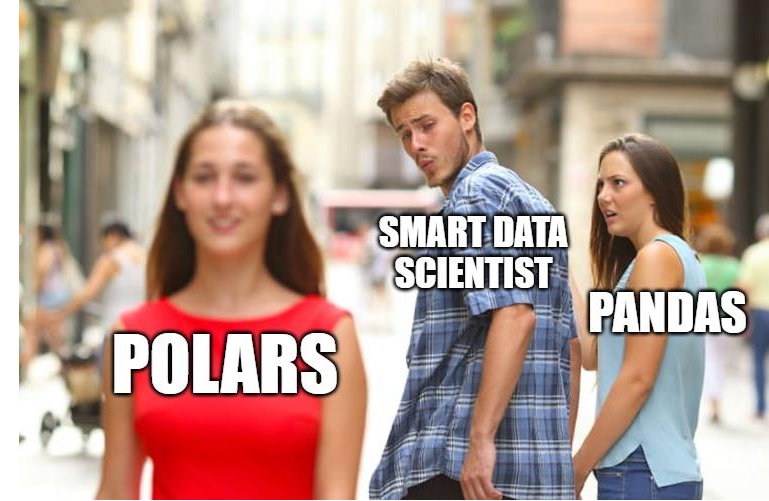
- A modern Python *project manager* and *environment manager*
- Handles **builds**, **packaging**, and **scripts**
- Integrates with **testing**, **linting**, and **release automation**
- Onboarding a new teammate?
 - One file → `pyproject.toml`
 - One command → `hatch shell`
- Common commands:
 - “`hatch new`” to create new environment
 - “`hatch env create`” to create environment from `pyproject.toml` file
 - “`hatch env prune`” to clean environment
 - “`hatch shell`” to launch environment
 - “`hatch test`” to run tests

Overview

- **Introduction**
- **Designing Clean Code with Reusable Structure**
 - Object-Oriented Programming for DS
 - Abstract Base Classes
- **Readable, Safe, and Maintainable Code**
 - Type-hinting with beartype
 - Google-style docstrings
 - Ruff: automated code linting & formatting
 - Unit testing
- **Experimentation & Configuration**
 - Hydra for structured config management
 - Hatch for reproducible environments
- **Handling Data at Scale**
 - Efficient dataframes with Polars
 - Reading/Writing from the cloud (fsspec, s3fs, ...)
- **Bridging Notebooks & Scripts**
 - Jupyter for collaborative reproducibility

Polars vs. Pandas

- Blazing fast: Built in Rust with native multi-threading
→ 10-100x faster than Pandas on many tasks
- Lazy execution engine: Optimizes entire computation graphs like Spark, but without the complexity
- Cleaner syntax: Functional, chainable, much like pandas (but no more `.apply()` chaos)
- Lightweight + scalable: Great performance even on laptops and can handle streaming data



Data in the cloud

- There are many connectors nowadays:
 - S3FS, gcsfs, adlfs
 - AWS' botocore
 - ...
- Once connected:
 - Polars
 - Pandas
 - ...

Overview

- **Introduction**
- **Designing Clean Code with Reusable Structure**
 - Object-Oriented Programming for DS
 - Abstract Base Classes
- **Readable, Safe, and Maintainable Code**
 - Type-hinting with beartype
 - Google-style docstrings
 - Ruff: automated code linting & formatting
 - Unit testing
- **Experimentation & Configuration**
 - Hydra for structured config management
 - Hatch for reproducible environments
- **Handling Data at Scale**
 - Efficient dataframes with Polars
 - Reading/Writing from the cloud (fsspec, s3fs, ...)
- **Bridging Notebooks & Scripts**
 - Jupyter for collaborative reproducibility

Notebooks revisited

- Main problems of notebooks:
 - Allow for “bad practice” code writing
 - “Spaghetti code”
 - Hard to version control
 - Poor for collaboration
 - ...
- But they are also:
 - Interactive and intuitive
 - Easy to prototype with
 - Good for demo’s
 - ...



Notebooks revisited

- jupy
+ text

- Insert JupyterText
 - Syncs notebooks with .py files
 - Makes it version controllable
 - ...

