MLOps Engineering
Machine Learning Operations V2.0.0
Sessions 4 - 5

MsC in Business Analytics and Data Science

Madrid, May 2025

ie UNIVERSITY

Evaluation methodology v2.0.0

Class Participation: **0.0**
Final exam fail: **3.5/10.0**

Activity	Weight		Description	(Tentative) Deadline
Ind. Class participation	**20%**	0	Relevant contributions in class and/ or via Teams	**ON GOING**
1st Group ~~Presentation~~	**15%**	1	1st group deliverable (Business case, VC'ed, production-ready-code)	**SESSION 8 (29st May)**
Intermediate test	**10%**	2	Initial core concepts & fundamental best practices	**SESSION 9 (29th May)**
2nd Group Work Presentation	**25%**	3	Final group project - Presentation (End-to-end CI/ CD)	**SESSION 14 (25th Jun)**
Ind. Final Exam	**30%**	4	Final closed-book exam	**SESSIONS 15 (25th Jun)**
Total	**100%**			

Refactor notebooks into modular, testable code with config separation, logging, and CLI readiness to enable production integration

General best practices

1. **Modularization**: Break down code into smaller, reusable functions or classes
2. **Configuration Management**: Use configuration files or environment variables for settings
3. **Logging and Monitoring**: Add logging and possibly some basic monitoring metrics
4. **Error Handling**: Include comprehensive error handling and data validation
5. **Dependency Management**: List all dependencies and versions for reproducibility
6. **Documentation**: Include sufficient comments and documentation for maintainability
7. **Code Quality**: Ensure the code is clean, readable, and follows PEP 8 standards
8. **Testing**: Include unit tests to validate the functionality
9. **Security**: Remove any sensitive or secret info (.env)

ML Pipeline specific

1. **data_loader.py**: Load data from CSV, databases, or APIs
2. **data_validation.py**: Validate schema, check types, handle missing data
3. **preprocessing.py**: Clean, scale, encode, and transform input data
4. **features.py**: Create and select engineered features
5. **model.py**: Define, train, save, and load ML models
6. **evaluation.py**: Compute and log performance metrics
7. **inference.py:** Apply trained model to new data (predict pipeline)
8. **config.yaml**: Centralize parameters, paths, and environment configs
9. **utils.py** (optional): Shared utilities (e.g. logging setup, plotting functions)

1. **Modularization**

Structure main.py to isolate execution logic, enable reuse, handle inputs cleanly

- Define a **main()** function to encapsulate the program's entry point
- Use **if __name__ == "__main__":** to ensure code executes only when the script is run directly
- Keep the main() **function concise**; **delegate tasks** to other functions or modules
- Avoid executing code at the top level; place **execution logic within functions**
- Include a **docstring** at the top of the file to describe the script's purpose
- Handle **command-line** arguments e.g. using the **argparse** module for flexibility
- Implement proper **error handling** within the main() function to manage exceptions gracefully
- Ensure that main.py serves as a **clear and minimal entry point**, delegating complex logic to other modules

```python
"""
main.py – Illustrates Python main script best practices

This script shows how to:
- Encapsulate execution logic inside a main() function
- Use argparse for flexible input handling
- Delegate tasks to modular functions
- Prevent top-level code execution
- Gracefully handle errors
"""

import argparse


def process_name(name: str) -> str:
    """Return a formatted greeting message."""
    return f"Hello, {name}!"


def main():
    """Main entry point for the script."""
    try:
        parser = argparse.ArgumentParser(description="Greet the user by name.")
        parser.add_argument("name", help="Name of the person to greet")
        args = parser.parse_args()

        message = process_name(args.name)
        print(message)

    except Exception as e:
        print(f"An error occurred: {e}")


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

Quick & Dirty exercise

1. Create a environment.yml
2. Create a main.py
3. Run the CLI file
4. Create a module.py
5. Create a package (__init__.py)

Basic working environment set up…

1. Ensure you have all Extensions installed on your VSC
   - Pylint
   - Flake8
   - Autopep8
   - Jupyter
   - Git/ Git Bash
   - Github
   - Github Actions
   - Github copilot
   - DVC

2. Usefull commands:
   - `conda config --add channels conda-forge`
   - `conda env list`
   - `conda env create -f environment.yml`
   - `conda activate <env_name>`
   - `conda env update -f environment.yml --prune`
   - `conda env export > environment.yml`

"Any fool can write code that a computer can understand. ***Good programmers write code that humans can understand****."*

-Martin Fowler

*"**Good code is its own best documentation**. As you're about to add a comment, ask yourself, 'How can I improve the code so that this comment isn't needed?'"-Steve McConnell*

6. Documentation

ie UNIVERSITY

READMES | How to Write a REA

udacity.com/course/writing-readmes--ud777

UDACITY Part of Accenture

Learn ▾ Catalog Individuals Business ▾ Government

Log In Join for Free

Limited-Time Offer: See What You Can Save With Your Personalized Discount – **Save Now**.

All Programs ▸ School Of Programming And Development ▸ Writing READMEs

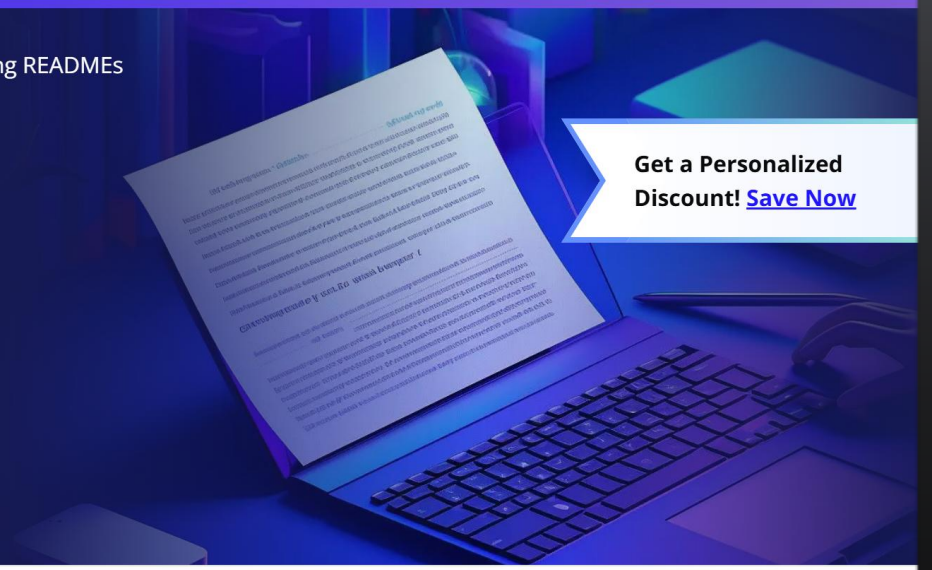
Free

Writing READMEs

Course

Documentation is an important part of the development process. Learn to write READMEs using Markdown so your code can be used by other humans!

Enroll Now ▶



Get a Personalized Discount! Save Now

Last Updated December 19, 2023

Prerequisites:

No experience required

6. Documentation

Course Lessons

5. Dependency Management

Use requirements.txt for pip-only setups, and environment.yml for **full reproducible Conda environments** with **system dependencies**

Feature	requirements.txt	environment.yml
Tool	pip	conda
Format	Plain text	YAML
Scope	Python packages only	Python and non-Python packages
Environment Name	Not specified	Defined under `name:`
Channels	Not specified	Defined under `channels:`
Python Version	Not specified	Specified as a dependency
Cross-platform	Yes	Yes
Usage	`pip install -r requirements.txt`	`conda env create -f environment.yml`
Export Command	`pip freeze > requirements.txt`	`conda env export > environment.yml`
Use Case	Simple Python projects	Complex projects with multiple dependencies

4. Error Handling

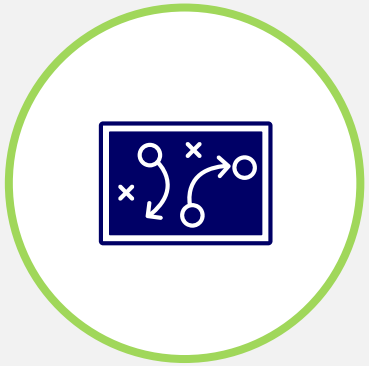
The Try: except: raise [sys.exit(1)] pattern

```
try:
    # Code that may raise an exception
except ExceptionType:
    # Code that runs if the specified exception occurs
raise
```

```python
import argparse


def main() -> None:
    """Script entry point"""
    parser = argparse.ArgumentParser(description="test")
    parser.add_argument("numerator", help="numbers to divide")
    parser.add_argument("denominator", help="numbers to divide by")
    args = parser.parse_args()

    try:
        numerator = int(args.numerator)
        denominator = int(args.denominator)
        result = numerator / denominator
        print(f"Result: {result}")
    except ZeroDivisionError:
        print("Error: Division by zero is not allowed.")
        raise
if __name__ == "__main__":
    main()
```

Feature	No try/except	With try/except + raise
Error is caught	No	Yes
Custom error message	No	Yes
Traceback shown	Yes	Yes (after custom message)
Execution stops on error	Yes	Yes (still re-raises)
Debug clarity	Raw only	Custom + traceback
Good for notebooks/shared demos	Less friendly	More transparent + instructive

3. Logging - When preparing your message think about these four questions



Why

are you telling the story?

Call to action?

Inspire or motivate?

Alignment?

What

is your key message?

What is the key question?

What do they need to know "so what"?

Who

is your audience?

What is their perspective?

What are their needs?

How

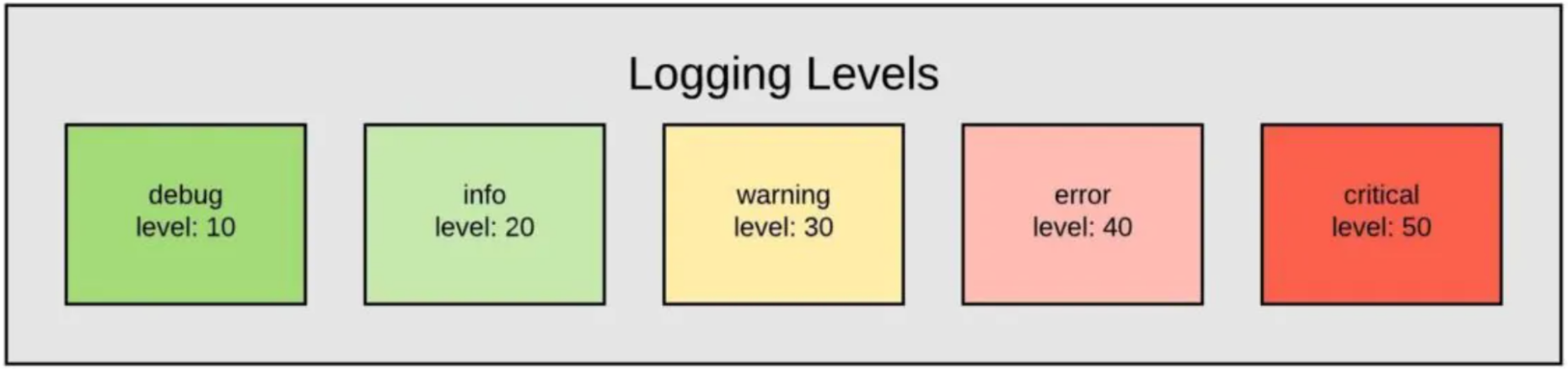
are you communicating it?

Are you informing or discussing?

Does it support your message?

Source: BCG-internal studies

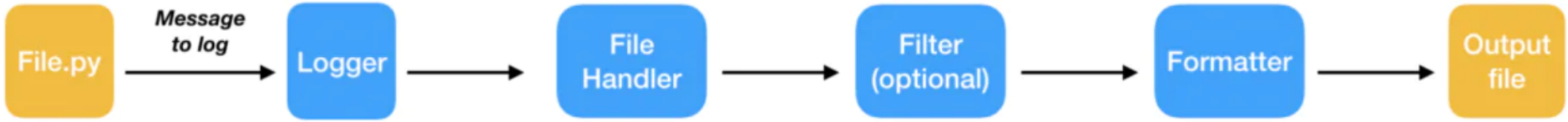
The logging priority scale in Python consists of the following 5 levels



- **DEBUG**: the messages written in this level provide detailed insights about the application. It is typically used by developers when they find a bug. Have you ever embedded a print statement to check the status of the variables when you have found a bug? If you log a message in the 'debug level' instead, you can avoid having to remove the print statement when your debugging phase is over, and you can reuse the logged message when the next bug appears 😉
- **INFO**: in this level, the messages are used to confirm that the application is running smoothly. We can regard the messages at this level as being the checkpoints of our program.
- **WARNING**: the messages written in this level provide insights that indicate something unusual is going on. The program should be able to handle the problem, but the output could be weird.
- **ERROR**: as suggested by the name, the messages in this level suggest that the application is not working correctly. The output will definitely be affected.
- **CRITICAL**: the highest level. The application will stop running soon and will no longer produce an output.

Source: https://www.codemotion.com/magazine/ai-ml/big-data/logging-in-python-a-broad-gentle-introduction/

The logging module defines functions and classes to include structured logs in our Python application.



As we can see there are three main actors:

- **LOGGER**: this is the main class of the module. We 'interact' with the logging system through this class of instances. Using its methods, we can write log messages in the log file.
- **HANDLER**: this class determines where the log messages will be written.
- **FORMATTER**: this class is used to decide the format of the log file. With a proper formatter, we can easily add the 'three w's' to our logger.

Python logging handlers let you send logs to files, console, email, remote servers, system logs, or anywhere you need, **just by changing the handler**

```
console = logging.StreamHandler()
```

Handler Name	Description / Destination	Example Use Case
StreamHandler	Console or any file-like stream (stdout/stderr)	Show logs in terminal, Jupyter, VS Code Output
FileHandler	Log file on disk	Save logs for later review, debugging, audit
RotatingFileHandler	Log file, automatically rotates when file gets large	Keep log files small, manage disk space
TimedRotatingFileHandler	Log file, rotates at time intervals (e.g. midnight)	Separate logs by day/week/hour
SMTPHandler	Sends logs via email (SMTP server)	Alert admins on errors or critical events
HTTPHandler	Sends logs to an HTTP server (as POST/GET)	Centralized logging, web-based dashboards
WatchedFileHandler	Like FileHandler, but checks file changes on disk	Useful with log rotation tools outside Python
SocketHandler	Sends logs over TCP/IP sockets	Remote log collection
DatagramHandler	Sends logs over UDP sockets	Remote log collection with UDP
SysLogHandler	Sends logs to Unix syslog daemon	Integrate with system-wide logging
NTEventLogHandler	Sends logs to Windows event log	Integrate with Windows system logs
MemoryHandler	Stores logs in memory, flushes to another handler on condition	Buffer logs, reduce writes for performance
QueueHandler	Sends logs to a queue (e.g. multiprocessing.Queue)	Multi-process safe logging
NullHandler	Swallows all log messages (does nothing)	Use in libraries to avoid errors if no logging config

With a simple line we can log in two different places with two different levels

```
import logging

file_handler = logging.FileHandler("logfile.log")
file_handler.setLevel(logging.WARN)
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.DEBUG)

logger = logging.getLogger()
logger.addHandler(file_handler)
logger.addHandler(console_handler)
logger.setLevel(logging.DEBUG)

logger.critical("Something critical")
logger.error("An error")
logger.warning("A warning")
logger.info("My info is that you are here")
logger.debug("I'm debugging")
```

Refactor notebooks into modular, testable code with config separation, logging, and CLI readiness to enable production integration

General best practices

1. **Modularization**: Break down code into smaller, reusable functions or classes
2. **Configuration Management**: Use configuration files or environment variables for settings
3. **Logging and Monitoring**: Add logging and possibly some basic monitoring metrics
4. **Error Handling**: Include comprehensive error handling and data validation
5. **Dependency Management**: List all dependencies and versions for reproducibility
6. **Documentation**: Include sufficient comments and documentation for maintainability
7. **Code Quality**: Ensure the code is clean, readable, and follows PEP 8 standards
8. **Testing**: Include unit tests to validate the functionality
9. **Security**: Remove any sensitive or secret info (.env)

ML Pipeline specific

1. **data_loader.py**: Load data from CSV, databases, or APIs
2. **data_validation.py**: Validate schema, check types, handle missing data
3. **preprocessing.py**: Clean, scale, encode, and transform input data
4. **features.py**: Create and select engineered features
5. **model.py**: Define, train, save, and load ML models
6. **evaluation.py**: Compute and log performance metrics
7. **inference.py:** Apply trained model to new data (predict pipeline)
8. **config.yaml**: Centralize parameters, paths, and environment configs
9. **utils.py** (optional): Shared utilities (e.g. logging setup, plotting functions)

1st Group assignment expectations and guidelines (Tech section)

Category	Details
Code Quality	• The code is following the **PEP8 coding style** (No errors on VSC > PROBLEMS tab) e.g. Meaningful names and syntax; properly commented and formatted; Imports are ordered, etc.
Documentation	• There's a comprehensive **README.rm** providing a general (preliminary) overview of the project, instructions on how to use the code, run tests, etc. [Anyone should be able to run the code from those instructions] • All functions and files have **docstrings**, and **commenting** is used appropriately i.e. **Why NOT What**
Testing	• **Tests cover all functions** and relevant artifacts e.g. Inputs are in the expected file format? Outputs are saved in the right places? Edge cases? (pytest-cov)
Logging	• **Each function** is complete with **logging** (INFO or otherwise), and Logs are stored in a **.log file** (Distinguishing between logs relevant to keep track of, and those to be shown on the terminal)
Config mgmt.	• A **config.yaml** (**python_env.yaml**) file(s) is used for Configuration Management and environment variables for settings (Avoid hardcoding variables!!)
Dependencies	• **Dependencies** are managed through a **environment.yml** and/or **conda.yml** file, listing all dependencies and versions for reproducibility
Error handling	• Error Handling thru **try/ except/ raise** provides a comprehensive error handling and **data type validation**
Artifacting	• Relevant reports, notebooks, or images of plots from the EDA phase are stored on a respective folder • A model(s) is properly stored (joblib or pickle) for future production use in inference phases
ML Pipeline Modules	The code modules cover and end-to-end ML process e.g. **1. main**.py (with **argparse** for CLI) **2. data_loader**.py: Load data from CSV, databases, or APIs **3. data_validation**.py: Validate schema, check types, handle missing data **4. preprocessing**.py: Clean, scale, encode, and transform input data **5. features**.py: Create and select engineered features **6. model**.py: Define, train, save, and load ML models **7. evaluation**.py: Compute and log performance metrics **8. inference**.py: Apply trained model to new data (predict pipeline) 9. utils.py (optional): Shared utilities (e.g. logging setup, plotting functions)
Version Controlling	GitHub is used for Version Control and development, demonstrating: • Proficiency with basic operations e.g. add, commit, push, etc. • Follow best practices (as discussed in class) for: 1. Project creation 2. Branching (Check out features, debugging, releases, development vs. main) 3. Pulls Request reviewing (Open, review and merge) 4. Tagging and releasing (version control and release) 5. Collaboration & Communication (Discussions and resolutions)

Code review check-list (Part of group assignment rubric)

1. Is the code clean and modular?
2. Can I understand the code easily?
3. Does it use meaningful names?
4. Is there duplicated code?
5. Can I provide another layer of abstraction?
6. Is each function and module necessary?
7. Is each function or module too long?
8. Is the code efficient?
9. Are there loops or other steps I can vectorize?
10. Can I use better data structures to optimize any steps?
11. Can I shorten the number of calculations needed for any steps?
12. Can I use generators or multiprocessing to optimize any steps?
13. Is the documentation effective?
14. Are inline comments concise and meaningful? i.e. Why not What
15. Is there complex code that's missing documentation?
16. Do functions use effective docstrings?
17. Is the necessary project documentation provided? i.e. README.md
18. Is the code well tested?
19. Does the code high test coverage?
20. Do tests check for interesting/ edge cases?
21. Are the tests readable?
22. Can the tests be made more efficient?
23. Is the logging effective?
24. Are log messages clear, concise, and professional?
25. Do they include all relevant and useful information?
26. Do they use the appropriate logging level?

Source: Udacity

Simplified project structure (illustrative)

```
project_name/
├── data/          # Raw and processed data
├── notebooks/     # Jupyter notebooks for exploration
├── src/           # Source code modules
│   ├── data/      # Data loading and preprocessing
│   ├── features/  # Feature engineering
│   ├── models/    # Model definitions and training
│   └── utils/     # Utility functions
├── tests/         # Unit and integration tests
├── configs/       # Configuration files (YAML/JSON)
├── environment.txt
└── README.md
```

ie UNIVERSITY

fmind/cookiecutter-mlops-pack

github.com/fmind/cookiecutter-mlops-package?tab=readme-ov-file

uv.lock — feat(release): create v4.1.0 from mlops-python-package (#2) — 2 months ago

README | Code of conduct | MIT license

Cookiecutter - MLOps Package

release v4.1.0 | license MIT

Jumpstart your MLOps projects with this comprehensive Cookiecutter template.

The template provides a robust foundation for building, testing, packaging, and deploying Python packages and Docker Images tailored for MLOps tasks.

Related resources:

- MLOps Coding Course (Learning): Learn how to create, develop, and maintain a state-of-the-art MLOps code base.
- MLOps Python Package (Example): Kickstart your MLOps initiative with a flexible, robust, and productive Python package.
- LLMOps Coding Package (Example): Example with best practices and tools to support your LLMOps projects.

Philosophy

This Cookiecutter is designed to be a common ground for diverse MLOps environments. Whether you're working with Kubernetes, Vertex AI, Databricks, Azure ML, or AWS SageMaker, the core principles of using Python packages and Docker images remain consistent.

This template equips you with the essentials for creating, testing, and packaging your AI/ML code, providing a solid base for integration into your chosen MLOps platform. To fully leverage its capabilities within a specific environment, you might need to combine it with external tools like Airflow for orchestration or platform-specific SDKs for deployment.

You have the freedom to structure your `src/` and `tests/` directories according to your preferences. Alternatively, you can draw inspiration from the structure used in the MLOps Python Package project for a ready-made implementation.

https://donate.stripe.com/4gw8xT9oVb...

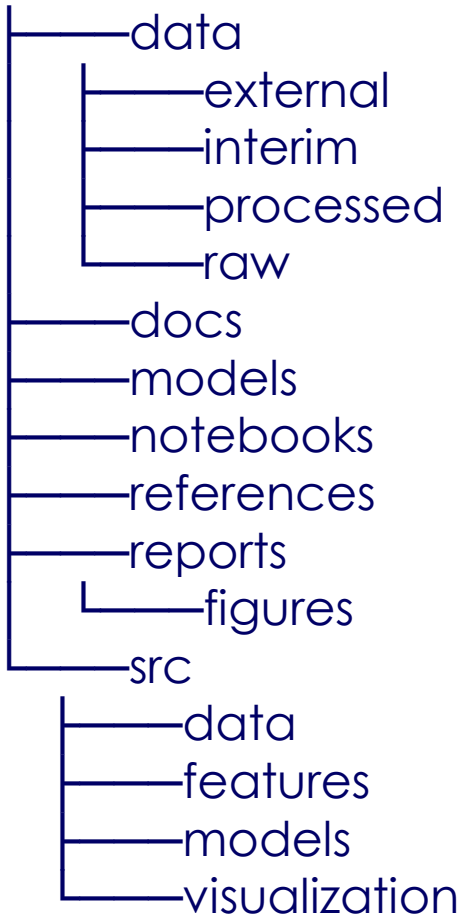
Languages

Just 71.7% Python 25.9% Dockerfile 2.4%

Steps – Setting up the working environment

1. **Create new conda environment from yml file**
   - conda env create -f environment.yml
   - conda activate <env_name>
2. **Create new project structure (cookiecutter)** <- **preferred** to do it **manually**
   - cookiecutter https://github.com/drivendataorg/cookiecutter-data-science -c v1
   - cookiecutter https://github.com/fmind/cookiecutter-mlops-package
3. **Create a GitHub repo**
   - **Locally**
     - git init
     - git add .
     - git commit -m "Initial commit“
   - **On GitHub**
     - ‘New’
     - ‘Create repository’
     - Copy URL
   - **Locally**
     - git remote add origin https://github.com/username/repository.git
     - git branch -M main
     - git push -u origin main

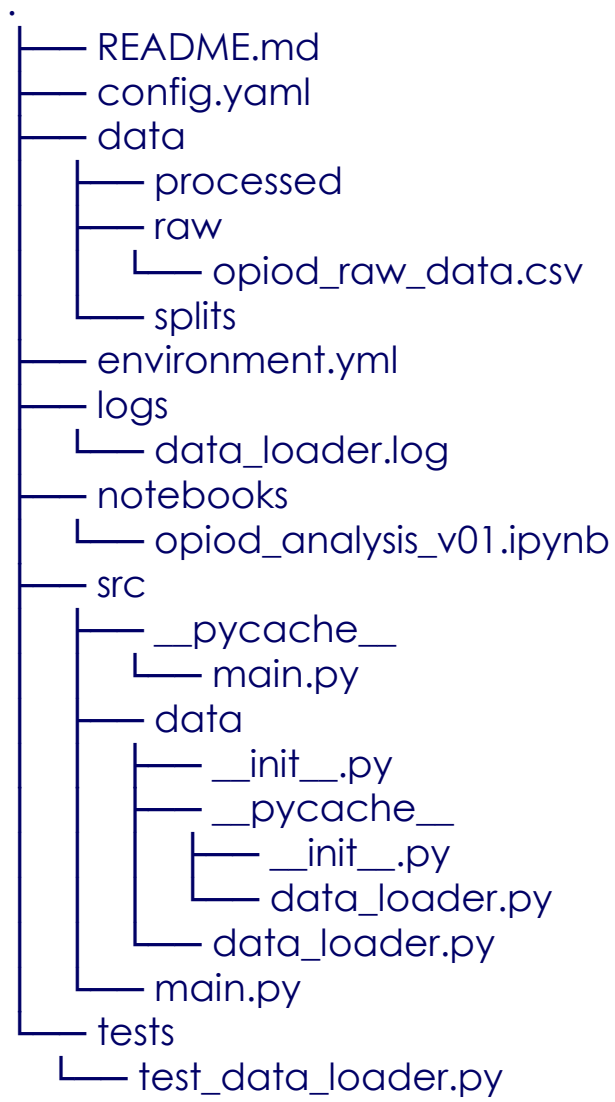
Standard
Cookiecutter DataScience
project structure




```
environment.yml
name: cookie
channels:
  - conda-forge
  - defaults
dependencies:
  - python=3.13
  - pip
  - pip:
      - cookiecutter

> conda env create -f environment.yml
> conda activate cookie

# DataScience
> cookiecutter
https://github.com/drivendataorg/cookiecu
tter-data-science -c v1

# MLOps
> cookiecutter
https://github.com/fmind/cookiecutter-
mlops-package
```

Steps – Refactoring the Jupyter Notebook

4. **Move the Jupyter Notebook to /notebooks**
5. **Run and Check Jupyter Notebook**
6. **Translating the . Ipynb to .py**
   - jupytext --to py notebook.ipynb
   - Pairing Notebooks (optional)
     - jupytext --set-formats ipynb,py notebook.ipynb
7. **Checking formatting standards**
   - Pylint and autopep8 installed on VSC
   - Use GitHub Copilot (AI agent) for refactoring and code optimization (be **careful** on what it does to your code!)
8. **Create individual modules for each section (ML step)**
   1. Checking formatting standards
   2. Add logging
   3. Error Handling / Try – Exception - Raise
   4. Maintain dependencies updated (environment.yml)
   5. Include sufficient comments and documentation
   6. Code Quality: Ensure the code is clean, readable, and follows PEP 8 standards
9. **Generate test suit for each function** (**Do not move forward** until your module passes your tests)




```
> python -m src.data.data_loader

2025-05-15 23:45:57 - INFO - root -
Loaded data from
./data/raw/opiod_raw_data.csv (csv),
shape=(1000, 22)
Data loaded successfully. Shape:
(1000, 22)

> python -m src.main

2025-05-16 01:12:19 - INFO - root -
Pipeline started
2025-05-16 01:12:19 - INFO - root -
Loaded data from
./data/raw/opiod_raw_data.csv (csv),
shape=(1000, 22)
2025-05-16 01:12:19 - INFO - root -
Data loaded successfully. Shape:
(1000, 22)
Data loaded. Shape: (1000, 22)
2025-05-16 01:12:19 - INFO - root -
Pipeline completed successfully
```