Setting up a Python project

This is a tutorial showing you how to set up a Python project for data analysis and machine learning.

Step by step, we will enhance the capabilities of the project, starting from a minimal setup, ending with a fully functional project structure that you can use as a template for your own projects.

Features

We will increase the capabilities of the project step by step:

Major version 1 - basic project setup & version control with `git`

- ✅ Dedicated project folder with "everything" inside
- ✅ A `README.md` file as starting point for all project documentation
- ✅ Configuration of project and dependencies in `pyproject.toml`
- ✅ A Python virtual environment for dependency management (managed with `uv`)
- ✅ Script files for reusable code
- ✅ Jupyter notebooks for interactive data analysis
- ✅ Repository structure for data, notebooks, scripts (i.a.)
- ✅ Version control with `git`
- ✅ Synchronization with GitHub

Major version 2 - Introducing further best practices from Software Development

- ✅ Packaging the project as a Python package for easy reuse
- ✅ Logging with `logging` module (instead of print statements)
- ✅ Testing with `pytest`
- ✅ Linting of code with `ruff`

Further enhancements can be added later, such as:

- ☐ Documentation generation with `Sphinx`
- ☐ Pre-commit hooks for automated checks before committing code.
- ☐ GitHub Actions for automated checks before code is published.

Prerequisites

It is assumed that you have already installed the required software as described in the installation instructions in the readme.

Major Version 1 - basic project setup & version control with `git`

Version 1.0 - Initial project setup

Features

✅ Dedicated project folder with "everything" inside
✅ A `README.md` file as starting point for all project documentation
✅ Configuration of project and dependencies in `pyproject.toml`
✅ A Python virtual environment for dependency management (managed with `uv`)
✅ Script files for reusable code
🔲 Not yet: Version control with `git` - but we will already have a `.gitignore` template for typical files not to be synced with git.

1. Create a new folder for the project

Create a new folder for your project, e.g. `my-project` and open it in VS Code.

> **IMPORTANT Guidelines for choosing a project directory**: 1. you should store python projects on a local drive (**not on a network drive**) and avoid using special characters or spaces in the directory name.
>
> 1. Furthermore, things will run more smoothly, if you pick a folder that is **not synchronized with OneDrive** (e.g. usually `C:\Users\<you>\projects`. In contrast, `C:\Users\<you>\Documents\projects` might cause issues as the `Documents`, `Desktop` and similar folders are usually synchronized with OneDrive).

2. Initialize with `uv`

Open a terminal, then run:

```
uv init --python=3.13
```

This will create several files: - `.gitignore`: Specifies files and folders that should not be tracked by git, preconfigured for typical Python projects - `.Python-version`: Specifies the Python version for the virtual environment - `main.py`: A simple Python script with a print statement - `README.md`: A markdown file for project documentation - `pyproject.toml`: Configuration file for the Python project and dependencies

Feel free to browse and inspect these files.

> **If `uv` is not available**: let me know; we can create the basic structure manually.

3. Edit the `pyproject.toml` file (e.g. project name, version)

4. Setup a minimal virtual environment

with uv:

Run the following command in the terminal, to create a new virtual environment with Python 3.13 (will be downloaded automatically if not installed):

```
uv venv
```

or, without uv:

```
python3 -m venv .venv
```

5. Activate the virtual environment

Activating the virtual environment (venv) differs between operating systems:

On Windows (PowerShell):

```
Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass
.\.venv\Scripts\activate
```

On macOS or Linux:

```
source .venv/bin/activate
```

6. Run the main script

Run the main script with:

```
python main.py
```

7. *Optional*: Update the version in `pyproject.toml`

Edit the `pyproject.toml` file to set the version to `1.0`:

```
version = "1.0"
```

Version 1.1 - add jupyter notebooks

Features

In addition to the previous version, we add the following features:

✅ Jupyter notebooks for interactive data analysis

1. Add dependencies for running jupyter notebooks

Run the following command in the terminal:

```
uv add ipykernel
```

or, without uv:

```
pip install ipykernel
```

2. Create a folder for notebooks

Use the file explorer in VS Code to create a new folder `notebooks` or open a terminal in the project folder and run:

```
mkdir notebooks
```

3. Create a new notebook with a simple print statement

Use VS Code to create a new file `notebooks/analysis.ipynb` and add a code cell with the following content:

```
print("Hello, Jupyter!")
```

4. Run the notebook

Use the "Run" button in VS Code to run the code cell. You should see the output `Hello, Jupyter!` below the cell.

5. *Optional*: Update the version in `pyproject.toml`

Version 1.2 - add dependencies (i.e. pandas, openpyxl) and some data

Features

In addition to the previous version, we add the following features:

✅ Repository structure for data, notebooks, scripts (i.a.)

1. Add pandas as a dependency

Run the following command in the terminal:

```
uv add pandas openpyxl
```

or, without uv:

```
pip install pandas openpyxl
```

This will add `pandas` as a dependency to the project and install it in the virtual environment.

2. Create a `data` folder with subfolders for different data processing stages

Use the file explorer in VS Code to create a new folder `data` with subfolders `raw`, `interim`, `prepared`, and `exports` or open a terminal in the project folder and run:

```
mkdir -p data/raw
mkdir -p data/interim
mkdir -p data/prepared
mkdir -p data/exports
```

3. Download some example data

Download the example data file from http://www.tobiaskeller.net/data/data_raw.zip and unzip it into the `data/raw` folder.

4. Load and inspect the data in a jupyter notebook

Create a new notebook `notebooks/data_inspection.ipynb` and add the following code to load and inspect the data:

```
import pandas as pd
df = pd.read_excel("../data/raw/wdi_reduced.xlsx", sheet_name="wdi")
```

```
df.head()
```

5. Go through the data-loading-checklist

Refer to the checklist for data loading issues if you encounter any issues loading the data.

6. *Optional*: Update the version in `pyproject.toml`

🧠 Live-Quiz on Kahoot 🏆

Go to https://www.kahoot.it and enter the game PIN to join the quiz.

Version 1.3 - Track your code with `git` and sync with GitHub

Features

In addition to the previous version, we add the following features:

✅ Version control with `git`
✅ Synchronization with GitHub

> 💡 **Tip (Reminder):** If you want your initial branch to be named `main` instead of the default (`master` in older Git versions), you can add:
>
> `bash`
> `git config --global init.defaultBranch main`
>
> This ensures consistency with platforms like GitHub, which default to `main`.

1. Initialize a git repository

Run the following command in the terminal:

```
git init
```

This will initialize a new git repository in the project folder.

2. Add the `data` folder to .gitignore

> 💡 **Note:** You should generally keep data files out of git!

In order to keep the data out of git, we add the `data` folder to the `.gitignore` file. This ensures that data files are not tracked by git. Add the following line to the file `.gitignore`:

```
# Ignore data files but keep folder structure
data/**
!data/**/.gitkeep
```

3. Add `.gitkeep` files to all data folders

In order to keep the folder structure in git, we add an empty file named `.gitkeep` to all data folders. This ensures that the folders are tracked by git, even if they are empty. Create an empty file named `.gitkeep` in each of the following folders.

You can use the terminal to create the files (commands differ between operating systems):

On Windows (PowerShell):

```
New-Item -Path data/raw/.gitkeep -ItemType File
New-Item -Path data/interim/.gitkeep -ItemType File
New-Item -Path data/prepared/.gitkeep -ItemType File
New-Item -Path data/exports/.gitkeep -ItemType File
```

On macOS or Linux:

```
touch data/raw/.gitkeep
touch data/interim/.gitkeep
touch data/prepared/.gitkeep
touch data/exports/.gitkeep
```

4. Add and commit all non-data files

> **Note:** You should Commit `pyproject.toml`, `uv.lock`, and `.Python-version` for reproducibility.

Run the following commands in the terminal:

```
git add .
git add data/**/.gitkeep --force # on some systems, we need to enforce the inclusion the first t
git commit -m "initial project setup"
git branch -M main # this is only necessary if you have not globally set the default branch to m
```

5. Create a new repository on GitHub

Login to your GitHub account and create a new repository named `my-project` (or another name of your choice). Create a **bare** repository, i.e. **do not initialize** the repository with a README, .gitignore, or

license.

6. Setup SSH keys for authentication (if not done yet)

If you have not set up SSH keys for authentication with GitHub yet, follow the instructions at https://docs.github.com/en/authentication/connecting-to-github-with-ssh to generate a new SSH key pair and add the public key to your GitHub account.

Alternatively, you can also use HTTPS for authentication, but you will have to enter your username and password (or personal access token) every time you push changes to GitHub.

7. *Optional*: Update the version in `pyproject.toml`

8. Add the remote repository and push the code

It is assumed you have created a new repository on GitHub named `my-project`. Replace `my-project` with the name of your repository and `XXXXXXXXX` with your GitHub username in the commands below.

Alternative 1: Using SSH (recommended)

Run the following commands in the terminal (replace `XXXXXXXXX` with your GitHub username):

```
git remote add origin git@github.com:XXXXXXXXX/my-project.git
git push -u origin main
```

Alternative 2: Using HTTPS

Run the following commands in the terminal (replace `XXXXXXXXX` with your GitHub username):

```
git remote add origin https://github.com/XXXXXXXXX/my-project.git
git push -u origin main
```

9. Demo: track changes with git using the VS Code source control pane

We will learn how to: * stage changes * commit changes * push changes to GitHub * make changes directly in GitHub and pull them to the local repository * view the history of changes using the `Git Graph` extension * revert to an earlier version of our code

Exercise 1: Setup your own project from scratch

1. Create a new, empty project folder

2. Set up the project as a Python package (hint: use `uv init` if available, edit `pyproject.toml` where necessary)
3. Setup a virtual environment with Python 3.12 (*yes - a different version!*) and install `pandas` and `ipykernel` to add minimal dependencies for this project
4. Create a repository structure with folders for `data`, and `notebooks`.
5. Download the file "bank+marketing.zip". It contains (among others) the file `bank-additional-full.csv` with data from a marketing campaign of a Portuguese bank. Place that file in the `data/raw` folder.
6. Create a notebook that loads the data with pandas and inspects it (`pd.read_csv()`, note the delimiter is a ";", i.e. `pd.read_csv("PATHTOFILE", sep=";")`).
7. Initialize a git repository and sync it with a new GitHub repository in your account.
8. Commit and push your code to GitHub.

Major Version 2 - Introducing further best practices from Software Development

Version 2.0 - Packaging reusable code in modules

Features

In addition to the previous version, we add the following features:

✅ Packaging the project as a Python package for easy reuse

1. Create a `src` folder for the Python package

We are going to create a package named `my_project` (you can choose another name if you like). This enables you to reuse code in scripts and notebooks easily.

Use the file explorer in VS Code to create a new folder `src/my_project` or open a terminal in the project folder and run:

```
mkdir -p src/my_project
```

2. Create an empty `__init__.py` file in the package folder

Create an empty file `src/my_project/__init__.py`. This file indicates that the folder is a Python package.

3. Create a module `io.py` for data input/output functions

Create a new file `src/my_project/io.py` and add the following code:

```
"""Module for reusable input/output functions."""

import pandas as pd

def load_wdi_excel(path: str) -> pd.DataFrame:
    """Load the WDI Excel file into a DataFrame and rename some columns."""
    df = pd.read_excel(path, sheet_name="wdi")
    df = df.rename(
        columns={
            "NY_GDP_MKTP_CD": "gdp",
            "NY_GDP_MKTP_KD_ZG": "gdp_growth",
            "SP_POP_TOTL": "population"
        }
    )
    return df
```

4. Edit `pyproject.toml` to include the package

Edit the `pyproject.toml` file to include the package. Add the following lines at the end of the file:

```
[tool.uv]
package = true

[build-system]
requires = ["hatchling>=1.25"]
build-backend = "hatchling.build"

[tool.hatch.build.targets.wheel]
packages = ["src/my_project"]
```

or, if `uv` is not available:

```
[build-system]
requires = ["setuptools>=61.0", "wheel"]
build-backend = "setuptools.build_meta"
```

5. Install the package

> **Note:** Make sure to activate the virtual environment first!

Run the following command in the terminal:

```
uv sync
```

or, without uv:

```
pip install -e .
```

6. Use the module in a jupyter notebook

Create a new notebook `notebooks/use_module.ipynb` and add the following code to use the `load_wdi_excel` function from the `my_project.io` module:

```
from my_project.io import load_wdi_excel

df = load_wdi_excel("../data/raw/wdi_reduced.xlsx")
df.head()
```

6. *Optional*: Update the version in `pyproject.toml`

7. *Optional*: build the package for distribution

In a previous step, we configured the project as a Python package. You can build the package for distribution with the following command:

```
uv build
```

This will create a `dist` folder with the package files that can be uploaded to PyPI or shared with others. When others download the `wheel` file, they can install it with `pip install <file>` or `uv add <file>`.

or, without uv:

```
pip install build
python -m build
```

Version 2.1 - Add proper logging and a helper function for logging setup

> **Background:** It is considered best practice to use the built-in `logging` module instead of print statements for logging messages in Python applications. This allows for better control over the logging output and levels (e.g. debug, info, warning, error). Furthermore, in production code, print may leak sensitive information to the console. With `logging`, clients can configure logging

> handlers to redirect logs to files or other destinations, and set appropriate logging levels (see https://docs.astral.sh/ruff/rules/print/ for more details).

Features

In addition to the previous version, we add the following features:

✅ Logging with `logging` module (instead of print statements)

1. Try out a simple logger in a script

Create a new file `scripts/logging_example.py` and add the following code:

```
import logging

def main():
    logging.basicConfig(level=logging.INFO)
    logger = logging.getLogger(__name__)
    logger.info("This is an info message")
    logger.debug("This is a debug message")
    logger.warning("This is a warning message")
    logger.error("This is an error message")
    logger.critical("This is a critical message")

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

Run the script with `uv run`; this ensures that the virtual environment is activated automatically:

```
uv run Python scripts/logging_example.py
```

Alternatively, activate the virtual environment yourself and run:

```
python scripts/logging_example.py
```

Note a few things: 1. We configure the logging level to `INFO`, so that debug messages are not shown. 2. We create a logger with `__name__`, so that the logger name corresponds to the module name. 3. There is no timestamp in the log message (which would be useful)

2. Improve the logging setup and change the log level to `DEBUG`

Edit the `scripts/logging_example.py` file to improve the logging setup:

```
import logging

def setup_logger(name: str, level: int | str = logging.DEBUG) -> logging.Logger:
```

```
    """Setup a logger with the given name and level."""
    logger = logging.getLogger(name)
    logger.setLevel(level)
    # Avoid duplicate handlers on re-import.
    if not any(isinstance(h, logging.StreamHandler) for h in logger.handlers):
        handler = logging.StreamHandler()
        handler.setFormatter(
            logging.Formatter("%(asctime)s - %(name)s - %(levelname)s - %(message)s")
        )
        logger.addHandler(handler)
    logger.propagate = False
    return logger

def main():
    logger = setup_logger(__name__, level=logging.DEBUG)
    logger.info("This is an info message")
    logger.debug("This is a debug message")
    logger.warning("This is a warning message")
    logger.error("This is an error message")
    logger.critical("This is a critical message")

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

3. Move the `setup_logger` function to a helper module in our package

Create a new file `src/my_project/logging_helper.py` and move the `setup_logger` function there.

4. Use the `setup_logger` function in our script

Edit the `scripts/logging_example.py` file to import and use the `setup_logger` function from the `my_project.logging_helper` module:

```
from my_project.logging_helper import setup_logger

def main():
    logger = setup_logger(__name__, level="DEBUG")
    logger.info("This is an info message")
    logger.debug("This is a debug message")
    logger.warning("This is a warning message")
    logger.error("This is an error message")
    logger.critical("This is a critical message")

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

Run the script again with:

```
uv run Python scripts/logging_example.py
```

or:

```
python scripts/logging_example.py
```

5. *Optional*: Update the version in `pyproject.toml`

Version 2.2 - Add testing with `pytest`

> **Background:** Testing is an important aspect of software development. It helps to ensure that the code works as expected and to catch bugs early. `pytest` is a popular testing framework for Python that makes it easy to write and run tests.

Features

In addition to the previous version, we add the following features:

✅ Testing with `pytest`

1. Add dependencies for testing

Run the following command in the terminal:

```
uv add pytest
```

or

```
pip install pytest
```

2. Create a `tests` folder for test files

Use the file explorer in VS Code to create a new folder `tests` or open a terminal in the project folder and run:

```
mkdir tests
```

3. Create a test file for the `logging_helper` module

Create a new file `tests/test_logging_helper.py` and add the following code:

```
import logging
import pytest
from my_project.logging_helper import setup_logger

@pytest.mark.parametrize("level", [logging.INFO, "INFO"])
def test_setup_logger_emits(level, caplog):
    logger = setup_logger("test_logger_unique", level=level)
    assert logger.name == "test_logger_unique"
    assert isinstance(logger, logging.Logger)
    caplog.set_level(logging.INFO, logger=logger.name)
    logger.info("hello")
    assert "hello" in caplog.text
```

4. Run the tests with `pytest`

Run the following command in the terminal:

```
uv run pytest
```

You should see output indicating that the test passed.

Again, you can also activate the virtual environment yourself and run:

```
pytest
```

5. Setup the test pane in VS Code

In VS Code, search for the "Testing" icon on the left sidebar (looks like a beaker) and click on it. Then click on "Configure Python Tests" and select `pytest`. This will create a `.vscode/settings.json` file with the following content:

```
{
    "Python.testing.pytestEnabled": true,
    "Python.testing.pytestArgs": [
        "tests"
    ],
    "Python.testing.unittestEnabled": false,
    "Python.testing.nosetestsEnabled": false
}
```

You can now run and debug tests directly from the test pane in VS Code.

6. *Optional*: Update the version in `pyproject.toml`

Version 2.3 - Add linting with `ruff`

> **Background:** Linting is the process of checking code for potential errors and enforcing coding standards. `ruff` is a fast and popular linter for Python that can help you to improve the quality of your code.

Features

In addition to the previous version, we add the following features:

✅ Linting of code with `ruff`

1. Add dependencies for linting

Run the following command in the terminal after activating the virtual environment:

```
uv add ruff
```

or

```
pip install ruff
```

2. Add a basic configuration for `ruff` in `pyproject.toml`

Edit the `pyproject.toml` file to add a basic configuration for `ruff`. Add the following lines at the end of the file:

```
[tool.ruff]
line-length = 100
```

3. Configure your VS code settings to indicate the line length limit

Edit the `.vscode/settings.json` file to add the following setting:

```
{
    ...

    "editor.rulers": [100],

    ...
}
```

4. Run `ruff` to lint the code

Run the following command in the terminal:

```
uv run ruff check .
```

or, after activating the virtual environment:

```
ruff check .
```

You will notice many issues in the test files.

5. Add exceptions for the test files and for print statements in notebooks

Edit the `pyproject.toml` file to add exceptions for the test files and for print statements in notebooks. Add the following lines at the end of the file:

```
[tool.ruff.lint.per-file-ignores]
"notebooks/*" = [
    "T201", # print ok in notebooks
    ]
"tests/*" = [
    "S101",    # use of assert detected
    "PLR2004", # magic values OK in tests
    "SLF001",  # access to private members OK in tests
    "INP001",  # no __init__.py for tests as they are automatically identified
    "ANN201",  # tests don't return by design
    ]
```

6. Install the `Ruff` extension in VS Code and inspect linting issues in the code

Install the "Ruff" extension in VS Code (search for "Ruff" in the extensions pane and install it).

7. *Optional*: Setup auto-formatting on save in VS Code

With the "Ruff" extension installed, setup auto-formatting on save in VS Code by adding the following lines to the `.vscode/settings.json` file:

```
{
    ...

    "editor.codeActionsOnSave": {
        "source.fixAll.ruff": true
    },

    ...
}
```

8. *Optional*: Auto-Fix linting issues in the code

```
uv run ruff check . --fix
```

or, after activating the virtual environment:

```
ruff check . --fix
```

9. *Optional*: Manually fix remaining linting issues in the code

Manually fix remaining linting issues in the code (e.g. in the test files).

10. *Optional*: Update the version in `pyproject.toml`

🧠 Live-Quiz on Kahoot 🏆

Go to https://www.kahoot.it and enter the game PIN to join the quiz.

Exercise 2: Improve your project from Exercise 1

1. create a module in `src/my_project/logging.py` reusing the `setup_logger` function from above. You may have to complete the `pyproject.toml` file and re-execute `uv sync` (if not done in an earlier step) to make the package work.
2. Use the logger in a script to log some messages.
3. setup `ruff` for linting and fix any issues in your code.
4. add a test file for your `logging` module and run the tests with `pytest`.