Python Project Development

Agenda

1. Python Modules
2. Python Application Layout
3. Data Science Python Project Layout
4. Cookiecutter Tool
5. Quiz

Python Module

- A module allows you to logically organize your Python code.
- Grouping related code into a module makes the code easier to understand and use.
- A module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

Python Modules – Hands On Tutorial

Layout 1: One-Off Script

.gitignore: This is a file that tells Git which kinds of files to ignore. [Available in github]

helloworld.py: This is the script that you're distributing. As far as naming the main script file goes, I recommend that you go with the name of your project (which is the same as the name of the top-level directory).

LICENSE: This plaintext file describes the license you're using for a project. It's always a good idea to have one if you're distributing code. The filename is in all caps by convention. [Available in github]

README.md: This is a Markdown file documenting the purpose and usage of your application. Crafting a good README is an art, but you can find a shortcut to mastery here.

requirements.txt: This file defines outside Python dependencies and their versions for your application.

setup.py: This file can also be used to define dependencies, but it really shines for other work that needs to be done during installation. You can read more about both setup.py and requirements.txt in our guide to Pipenv.

tests.py: This script houses your tests

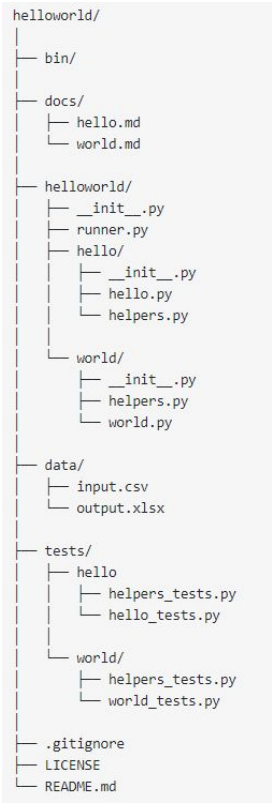
Layout 2: Installable Package

```
helloworld/
│
├── helloworld/
│   ├── __init__.py
│   ├── helloworld.py
│   └── helpers.py
│
├── tests/
│   ├── helloworld_tests.py
│   └── helpers_tests.py
│
├── .gitignore
├── LICENSE
├── README.md
├── requirements.txt
└── setup.py
```

The only difference here is that your application code is now all held in the helloworld subdirectory:

1. directory is named after the package
2. added a file called __init__.py
3. tests are written in a separate module.

Layout 3: App with Internal Packages



bin/: This directory holds any executable files. The most important point to remember is that your executable shouldn't have a lot of code, just an import and a call to a main function in your runner script. If you are using pure Python or don't have any executable files, you can leave out this directory.

/docs: With a more advanced application, you'll want to maintain good documentation of all its parts.

helloworld/: This is similar to helloworld/ in the previous structure, but now there are subdirectories. As you add more complexity, you'll want to use a "***divide and conquer***" tactic and split out parts of your application logic into more manageable chunks. Remember that the directory name refers to the overall package name, and so the subdirectory names (hello/ and world/) should reflect their package names.

data/: Having this directory is helpful for testing. It's a central location for any files that your application will ingest or produce. Depending on how you deploy your application, you can keep "production-level" inputs and outputs pointed to this directory, or only use it for internal testing.

tests/: Here, you can put all your tests—unit tests, execution tests, integration tests, and so on.

Data Science Project Layout

```
├── LICENSE
├── Makefile           <- Makefile with commands like `make data` or `make train`
├── README.md          <- The top-level README for developers using this project.
├── data
│   ├── external       <- Data from third party sources.
│   ├── interim        <- Intermediate data that has been transformed.
│   ├── processed      <- The final, canonical data sets for modeling.
│   └── raw            <- The original, immutable data dump.
│
├── docs               <- A default Sphinx project; see sphinx-doc.org for details
│
├── models             <- Trained and serialized models, model predictions, or model summaries
│
├── notebooks          <- Jupyter notebooks. Naming convention is a number (for ordering),
│                         the creator's initials, and a short `-` delimited description, e.g.
│                         `1.0-jqp-initial-data-exploration`.
│
├── references         <- Data dictionaries, manuals, and all other explanatory materials.
│
├── reports            <- Generated analysis as HTML, PDF, LaTeX, etc.
│   └── figures        <- Generated graphics and figures to be used in reporting
│
├── requirements.txt   <- The requirements file for reproducing the analysis environment, e.g.
│                         generated with `pip freeze > requirements.txt`
│
├── setup.py           <- makes project pip installable (pip install -e .) so src can be imported
├── src                <- Source code for use in this project.
│   ├── __init__.py    <- Makes src a Python module
│   │
│   ├── data           <- Scripts to download or generate data
│   │   └── make_dataset.py
│   │
│   ├── features       <- Scripts to turn raw data into features for modeling
│   │   └── build_features.py
│   │
│   ├── models         <- Scripts to train models and then use trained models to make
│   │   │                 predictions
│   │   ├── predict_model.py
│   │   └── train_model.py
│   │
│   └── visualization  <- Scripts to create exploratory and results oriented visualizations
│       └── visualize.py
│
└── tox.ini            <- tox file with settings for running tox; see tox.readthedocs.io
```

In this layout, **src** directory can be named after project package name as well.

Data Science Project Layouts can vary depending on what all components it contains. Some of the components are:

a. Data Engineering Pipelines (airflow / celery / API / SQL)
b. Model Training Pipeline (CLI)
c. Model Serving (API)
d. Model Validation (CLI)
e. Visualization
f. Report Generation

REF: https://drivendata.github.io/cookiecutter-data-science/

Python Virtual Environment

- Virtual Environment: the main purpose of Python virtual environments is to create an isolated environment for Python projects. This means that each project can have its own dependencies, regardless of what dependencies every other project has.

- Tools to create virtual environments:
    - virtualenv
    - pyenv
    - pipenv
    - poetry

Ref:
https://realpython.com/python-virtual-environments-a-primer/
https://docs.python-guide.org/dev/virtualenvs/

Python Packaging Tutorial

- https://packaging.python.org/tutorials/packaging-projects/
    - Publish the python package on the test pypi repo.

- Files Required for setup tools and build tools:
    - pyproject.toml is the file that tells build tools what system you are using and what is required for building.
    - setup.py is the build script for setuptools

Python Automated Testing

- Pytest

Running from virtual env: `python -m pytest`

Reference

- https://realpython.com/python-modules-packages/
- https://realpython.com/python-application-layouts
- https://the-hitchhikers-guide-to-packaging.readthedocs.io/en/latest/quickstart.html
- https://github.com/audreyr/cookiecutter-pypackage
- https://docs.python-guide.org/