Notebooks to Packages: Creating Reusable Python Code

Michal Grzadkowski
January 26, 2024
Intro

- BMath, SM in Computer Science
- 10+ years working in software development within academia
- Princeton RSE since August 2021
- Currently working on cryo-EM applications under Ellen Zhong
Agenda

1. Introduction
2. Notebooks
3. Scripts
4. Packages
5. Classes (OOP)

Submitted notebooks:

a. classifier-tuning-exam
b. soil-microbe-dormancy
Why Notebooks?

- Easy to get started, easy to use
- Interactive coding, interactive plots
- Splitting pipelines into cells allows for selective execution
Our UFO Sightings Notebook

1. Read in a .csv file containing rows of sightings
2. Filter out bad rows with malformed sightings
3. Make plots showing state-wise sightings
4. Train a time series regressor to predict sightings
Our UFO Sightings Notebook

1. Read in a .csv file containing rows of sightings
2. Filter out bad rows with malformed sightings
3. Make plots showing state-wise sightings
4. Train a time series regressor to predict sightings
Our UFO Sightings Notebook

1. Read in a .csv file containing rows of sightings
2. Filter out bad rows with malformed sightings
3. Make plots showing state-wise sightings
4. Train a time series regressor to predict sightings
Problems with Notebooks

⇒ limited ability to execute blocks of code selectively
Problems with Notebooks

⇒ limited ability to create programming control structures at the macro level (cells, functions, etc.)
Problems with Notebooks

predict_sightings.ipynb

⇒ limited ability to parametrize (user chooses filter A or filter B)

Load results

```python
# Specify the workdir and the epoch number (0-based index) to analyze
WORKDIR = '...'  
EPOCH = 49  # CHANGE ME
```

```
x = "loam"
n: Zr, ET_max, ET_w, sfc, sw, sh, s1, beta, Ks, nn, km, hh, beta1.soil, beta2, beta22 = soil(x)
```

```
length = 730 # simulation time (days)
s0 = 0.5 # Soil moisture at the beginning of the simulation [-]
Ha = 530 # Atmospheric concentration of hydrogen* 
dt = 0.001 # time interval (days)
dry = 100  # Initial dryness [-]
```
Problems with Notebooks

- One bad cell breaks the rest
- Can only run the code from Jupyter
- Difficult to keep track of dependencies
- Designed for demos and initial exploration
Converting to Scripts

predict_sightings.py

```python
def main():
    Read
    Filter
    Plot
    Predict

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

⇒ create a Python file that can be executed from command line:

```bash
gthon predict_sightings.py```

Converting to Scripts

predict_sightings.ipynb

predict_sightings.py

```python
def main():
    Read
    Filter
    Plot
    Predict

if __name__ == '__main__':
    main()

$> python predict_sightings.py
```
Converting to Scripts

$> jupyter nbconvert --to script sightings.ipynb

- nbconvert tool concatenates cells together, but nothing else
- Put code into main() function and call main() within an if __name__ == '__main__' clause
- Clean up import statements, variable declarations, etc.
- Debug using print statements or pdb.set_trace()
Converting to Scripts

predict_sightings.py

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("years", nargs=2)
    parser.add_argument("--states", nargs='+')

if __name__ == '__main__':
    main()
Turning our cells into functions makes it easier to create control structures that take advantage of parametrization!
Next Step: Packaging

A stand-alone script is ok, but:

- No support for dependency management
- Size will eventually become unmanageable as project grows
- Only one `main()`
- How to share with others?

```python
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("years", nargs=2)
    parser.add_argument("--states", nargs='+')
    data = read(args.years, args.states)
    data = filter(data)
    if args.plot:
        plot()
    predict()

if __name__ == "__main__":
    main()
```
Next Step: Packaging

```
predict_sightings.py

from .read import read_data
from .filter import filter_data
from .plotting import create_plot
from .learning import train_model

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("years", nargs=2)
    parser.add_argument("--states", nargs='+')
    data = read_data(args.years, args.states)
    data = filter_data(data)
    if args.plot:
        create_plot()
    train_model()

if __name__ == "__main__":
    main()
```
Next Step: Packaging

**predict_sightings.py**

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

**plot_sightings.py**

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

**plot_sightings2.py**

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

**read.py**

```python
import pandas as pd

def read_data():
    Read
```

**filter.py**

```python
import pandas as pd

def filter_data():
    Filter
def special_filter1():
```

**plotting.py**

```python
import matplotlib

def create_plot():
    Plot
def create_plot2():
```

**learning.py**

```python
import sklearn

def train_model():
    Predict
def train_model2():
```
Next Step: Packaging

- How to organize the files and directories in our package?
  - Simplest: create one directory for code
  - Create other directories for demos, data, etc.
  - Put project metadata in root folder
Next Step: Packaging

- One metadata file required for packaging: `pyproject.toml`
- Information needed by Python to create the package
- Also useful for specifying:
  - commands to install
  - which directories contain code vs. data
  - etc.

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

[project]
name = "PredUFO"
version = "2.0.0-a0"
authors = [
  {name="Michal Grzadkowski", email="mgrpzad@princeton.edu"},
]
description = "predicting UFO sightings"
readme = "README.md"
requires-python = ">=3.9"
classifiers = [
  "Programming Language :: Python :: 3",
  "License :: OSI Approved :: MIT License",
  "Operating System :: OS Independent",
]
dependencies = ["pandas", "plotly", "jupyter", "imageio", "matplotlib",
                "scikit-learn<1.1", "nbconvert", "nbformat"]

[project.urls]
Homepage = "https://github.com/michal-g/Notebooks-to-Packages"

[project.scripts]
predUFO-USA = "predufo.command_line:predict_usa"
predUFO-Canada = "predufo.command_line:predict_canada"

[tool.setuptools]
packages = ["predufo"]

[tool.setuptools.package-data]
"*" = ["*.csv"]
```
Sharing Your Package

Find, install and publish Python packages with the Python Package Index

Search projects

Or browse projects

The Python Package Index (PyPI) is a repository of software for the Python programming language.

PyPI helps you find and install software developed and shared by the Python community. Learn about installing packages.

Package authors use PyPI to distribute their software. Learn how to package your Python code for PyPI.
Sharing Your Package

You are using TestPyPI - a separate instance of the Python Package Index that allows you to try distribution tools and processes without affecting the real index.

Test Python package publishing with the Test Python Package Index

Search projects

Or browse projects

189,193 projects 1,019,536 releases 2,119,532 files 179,612 users

The Python Package Index (PyPI) is a repository of software for the Python programming language. PyPI helps you find and install software developed and shared by the Python community. Learn about installing packages.

Package authors use PyPI to distribute their software. Learn how to package your Python code for PyPI.
Object Oriented Programming

- Design code around objects as opposed to functions
  - Objects have functions that act on itself or other objects
    - Objects create instances with internal states
- Does not add functionality, but improves legibility and scalability of a codebase (arguably)
- Best understood through worked examples!