Lecture 08

Python Packages

Packaging Python Modules

Why create your own python modules?

- Better organize and use library code
- Easily install project (e.g. using `pip/conda`).
- Manage dependencies on a project-by-project basis.
- Better leverage *virtual environments* to make set-up easier, and minimize breaking your own Python installation.
- Docker may be overkill; manage Python using Python.

Ideal Usage of Packaging

- My Python-based project `detect_malware` is packaged as a Python module and uploaded to PYPI (Python Package Index).

- Other that want to replicate the project merely run from *anywhere* on the user's computer:

  - `pip install detect_malware`

  - `detect_malware data analysis train`

Python Modules

- What happens when the statement `import pydotplus` is executed in Python?

```
In [5]: import pandas as pd
        import numpy as np
        from matplotlib import pyplot as plt
        %matplotlib inline

        from sklearn.tree import DecisionTreeClassifier as DT
        from sklearn.tree import export_graphviz
        from IPython.display import Image
        import pydotplus
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-5-673d987bf678> in <module>()
      8 from sklearn.tree import export_graphviz
      9 from IPython.display import Image
---> 10 import pydotplus

ModuleNotFoundError: No module named 'pydotplus'
```

Module Search Path

- Upon import, Python checks for `pydotplus.py` in:
    - The *current directory* (if being run interactively) or in the same location as the python script.
    - In the directories contained in the PYTHONPATH environment variable (if it exists). Access this environment variable in Python via `sys.path`
    - In the site-packages directory, in the python installation location.
- To check *where* a python file is located, upon import, use the `module.__file__` attribute.

A Minimal Python Package

```
funniest/
    funniest/
        __init__.py
        text.py
    setup.py
```

- The project directory is the name of package. It contains:
    - The library code for the project (same name)
    - Package metadata files (e.g. `setup.py`).
- Any directory with library code should contain an `__init__.py` file that contains a list of importable functions/classes (makes a function *public*).

A Minimal Python Package

```
funniest/
    funniest/
        __init__.py
        text.py
    setup.py
```

setup.py

```
from setuptools import setup

setup(name='funniest',
      version='0.1',
      description='The funniest joke in the world',
      url='http://github.com/storborg/funniest',
      author='Flying Circus',
      author_email='flyingcircus@example.com',
      license='MIT',
      packages=['funniest'],
      zip_safe=False)
```

- Install the package via
  - `pip install .` (copy to site-package directory)
  - `pip install -e .` (develop mode: symlink to site-package directory)

Package Files

```
funniest/
    funniest/
        __init__.py
        text.py
    setup.py
```

In `__init__.py`:

```
from .text import joke
```

In `text.py`

```
def joke():
    joke1 = 'There are two kinds of data scientists:'
    joke2 = ' (1) those who can extrapolate'
    joke3 = ' from incomplete data.'
    return  joke1 + joke2 + joke3
```

(DEMO)

Adding Package Components

- Dependencies:
    - Specify modules used by your package in a `requirements.txt` file and pass the contents (as a list) to `setup`'s `install_requires` argument in `setup.py`
- Data:
    - Additional data that should be packaged *with the installation* are listed in a MANIFEST.in file, and `include_package_data=True` is passed to the setup function.
- Scripts:
    - Create globally usable scripts from library files (like `run.py`) passing the script location to the `scripts` keyword in the `setup` function.
- See many tutorials on web (e.g. this one).