Week 4 day 3: Projects

- multiple Diminsion example - Numpy

Projects

Should have set of goals to reach.
- It's okay if not all are complete

Collaborative Projects are okay
- but have individual goals & output

Should include
- a few planned plots/graphics
-

Anatomy of a Python Project

project - sometimes has longer name
- .git should be a git repo
- setup.py this makes it pip installable (we will skip this)
- notebooks/ this is where you use your code
- tests/ tests go here
- examples/ examples of use
- docs/ documentation
- scripts/ runnable python files
- project/ this folder is for code
  - __init__.py this tells Python this is a package
  - code.py various code files
  - sub1/ sub packages can go here
  - data/ small amounts of data too
- requirements.txt a list of libraries you use (several forms)

Example: Project MCIntegrator

```
mcintegrator
  └ .git
  └ mcint
      └ __init__.py
      └ simple.py
  └ notebooks
      └ Use simple.ipynb
```

Writing a .py file

run

```
#!/usr/bin/env python3
    └ only on runnable files
```

mostly for Python 2

```
from __future__ import ...
    └ add optional features
```

```
import ...
from ... import ...
```

imports that you need

```
def ...
```

– functions and classes

run

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

(if file is runnable, this makes sure it is also importable)

So, what happens when you try to import a library?

You type:

```
import thing
```

Python looks for:

```
thing.py
thing/__init__.py
```

Python looks through the list:

```
sys.path
```

in order, one at a time.

note: if you set PYTHONPATH in your shell, that will be added to sys.path on startup.

There are several "site-packages" locations - this is where Pip installs to. You at least have a system one and a user one.

You usually also get your current folder, too.

for now, the following hack works:

```
import sys
sys.path.append('..')
```

This adds the Parent directory to the path.

~ I was not trying to draw a face...

Note: the real solution? use setup.py and Pip or Pipenv. But not now.

Writing a good package

- **Tests** - you should write tests - the more, the better. PyTest is great and much simpler than unittest from the standard library. Nose used to be common, but has been replaced by pytest.
- **Documentation** - we have it easy - notebooks help. But still make it a habit to write docstrings everywhere. Comments too as needed.
- **Style** - This helps you read others code, and others read your code. There are lots of tools to help. PEP 8 (Python Enhancement Proposal) covers style.

IDE's (Integrated Development Environments)

- PyCharm - Hard to beat. only issue is its massive feature set.
- Spyder - comes with anaconda. Looks like MatLab.
- Jupyter Notebook/Lab - no features, but can edit files
- IDLE - very simple editor that comes with Python.

Installing Packages

PyPI & pip

- works everywhere
- Great for Pure Python
- Often OK for binaries (new)
- still poor for some binary packages

pip install package

other repos is uncommon.

Anaconda Cloud & conda

- only for anaconda Python
- Best binary support
- Can mange Python version too
- Graphical installer

conda install package

Can use other channels, too:

-c conda-forge

Note: CERN ROOT does not like conda (yet).

Virtual Environments (advanced)

You often need a specific set of libraries - possibly locking or limiting the version (reproducable research). You also may want to ensure you know exactly what you have. You also might want to avoid messing with your system Python.

Solution: virtual environments

- Pipenv - this is new, but recommended by Python.org

- conda - Your "system" environment is "root" - and you can add more.

- You can enter/exit a V.E.
- Packages are shared (1 download) and linked in
- You can usually turn on/off access to the system packages
- You can provide a package list
- You can get an exact package list