Programming and Data Science for Biology (PDSB)

Session 5
Spring 2018

Submitting assignments

Lecture/

Notebooks/

Code-Review/

Assignment/

```
## Use `cp` to copy files for submitting
cp Notebooks/3.1.ipynb
Assignment/myname-3.1.ipynb

## Then ONLY add/commit Assignment files
git add Assignment/myname-3.1.ipynb
git commit -m “adding assignment file 3.1”
git push

## They key thing is to NOT do this
git add Notebooks/myname-3.1.ipynb
git commit -m “why am I changing the notebooks?”
git push
```

Course projects

It’s OK if you don’t have an idea yet.

It’s great if you do have an idea, but stay open to ideas for how to implement it.

Requirements:

+ Uses packages or methods not covered in class.
+ That it is a tool: other people can reuse your program to accomplish some task. It’s not just a notebook running a single example or study.
+ That it’s packaged and distributed professionally on GitHub.

```
## Come to my office hours

You need approval for your project.

## We will have an in-class hackathon

Get help from your peers on ideas for
implementing your planned program.
```

Python advanced: Classes, and imports

Python Classes

Object oriented programming.
Organized, simple code.

Common questions:

- What is self?
- What is _init_?
- When to return vs. store results
- How to structure the object
- Py2/Py3 differences (ugh).
- *Test your code!*

```
## a simple class with an init function
class Simple:
    def __init__(self, name):
        self.name = name

## an instance of the class object
x = Simple('deren')
x.name
deren
```

Packaging: Writing Python API, CLI, and packing for distribution.

Packaging Python code

As with most things, there are multiple ways to package and distribute Python code, but some ways are better than others. We'll learn the proper way to make and distribute a Python package.

```
# installable globally
pip install mypackage

# importable globally
import mypackage

# has metadata
mypackage.__version__
0.1
```

The setuptools package

The standard library has a package named setuptools that can be used to package your files together into a single shareable **source code** file.

We will use setuptools in combination with pip to package and install our code in the most modern way.

The setuptools package

We will define a **project** and a **package**, although we will use the terms interchangeably many times. The difference is that within a project you could have multiple packages, which are sometimes also called **modules.** If your project includes a single package then the project and package names should be the same.

```
# project directory
mkdir biocode

# inside, the package dir and setup.py
cd biocode
mkdir biocode
touch setup.py
```

The setuptools package

The **project** is stored with ‘name’, and the **package** is the directory where our Python source code will be. Version is just a string, we can make it whatever we want.

```
# a simple setup.py script
from setuptools import setup
setup(
     name="biocode",
     packages=["biocode"],
     version="0.1",
     )
```

Source code and _init_ files

You can think of the source code directory biocode/biocode as a Class object, and we need to give it an _init_ function so that it knows what to do when it starts, which right now is nothing (the file is empty).

```
# project directory
touch biocode/__init__.py
```

Add code to _init_

From the init file you can add metadata tags for the package, and you can **import** code.

```
# in __init__.py
__version__ = “0.1”
from .script import *
```

Add code to be imported

The init file should import all of the code that you want to be accessible from your package object when it is imported. This is just like local imports, the files being imported by _init_ should be located in its same directory or lower. I'll copy some scripts here to demonstrate.

```
# in __init__.py
__version__ = "0.1"
from .script import *

# I copied the file script to this dir
cp ~/othercode/script.py ./
```

Writing and testing good code: A workflow using SublimeText3 and jupyter notebooks.

Designing code for Python users

You want your code to be easy to read both stylistically (indentation, spacing), but also logically (declare objects in certain places even if they’re empty and fill them later).

Pythonista style guide, *Pep8*, and *pycodestyle*.

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to
break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
```

Getting help from a Text editor

Text editors can help you write your code according to style guides by making it easier to see your code through **syntax highlighting,** and by finding errors through the use of **linters.**

```
## syntax highlighting shows different
## types of objects in your code
def function(arg1, arg2):
    “Doc string here”
    x = 3
    return x + 4

## linters highlight errors in your code
def function(arg1, arg2: [invalid syntax]
    ...
```

Sublimetext

Fork the repository for today's class.
In your browser open:
Notebooks/5.1-sublimetext.ipynb

Assignments and readings

Assignment is due Friday at 5pm

Code review is due Monday by class.

Assignment: Link to Session 5 repo

Readings: See syllabus

Collaborate: Work together in this gitter chatroom