

Organizing Your Code

Demitri Muna
OSU

2 August 2016

Common Code Pattern

A common pattern in people's code goes something like this:

Script to do some analysis...

```
[set some parameters for the script]
[read "datafile.dat"]
[loop over the data file]
[read into some structure]
[analyze code for x]
[analyze code for y]
[write results to output file]
```

Then another script...

```
[set other parameters for the script]
[read "datafile.dat"]
[loop over the data file]
[read into some structure]
[analyze code for x]
[analyze code for z]
[write results to output file]
```

- Much of code becomes a template
- Code is repeated
- Changes/fixes to one script need to be made to the others
- Code can't be reused in new scripts or by others
- Scripts are essentially one-offs
- Analysis code mixed in with bookkeeping code

Example: Analyzing SDSS Data

```
[go to data directory]
[loop over plates (one per directory)]
  [open FITS file]
  [read header]
  [read table]
  [select spectra that match "galaxy" and "0.1 < z < 0.2"]
  ["bookmark" that spectrum]
[loop over found spectra]
  [open FITS file]
  [read spectrum from correct HDU]
  [read ra/dec from correct header]
  [analyze spectrum]
  [write results out to file]
[generate plots]
```

← mostly bookkeeping,
would be duplicated
for other scripts

← parameters
buried in script

← code depends on file format –
if this changes, many scripts
need to be updated

Aims of Code

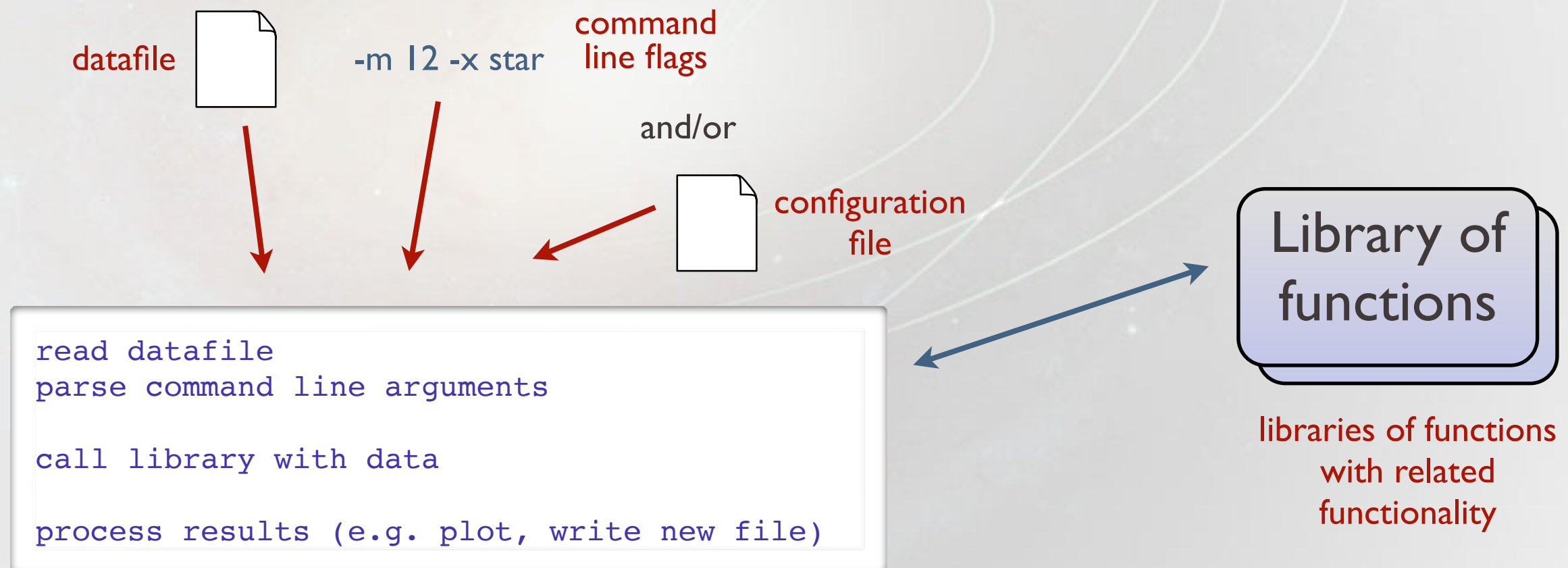
- Functions that do a particular job should be written once and used by other pieces of code.
- Data files should be separate from functionality.
- Input parameters should be separate from functionality.

Unix Model of Code

Unix has a particular design philosophy for code.

- All functionality is written in a library.
- The library can be called by programs or other libraries
- Programs call the library to perform tasks.
- Programs are “thin” – they:
 - read files
 - set parameters
 - produce output

Unix Model of Code



Calling method examples:

```
my_script -m 12 -x star datafile
my_script configfile.cfg
```

sample config file

```
datafile = mydata.fits
m = 12
iterations = 100
type = star
```

Example Revisited: Analyzing SDSS Data

```
import SDSSData

[argparse config file]

spectra = SDSSData.SpectrumSearch(z=[zmin, zmax],
                                   kind='galaxy',
                                   mag=[mag_min, mag_max])

for spectrum in spectra:
    ra = spectrum.ra
    dec = spectrum.dec
    sigma = spectrum.standard_deviation
    x = SDSSData.find_lyman_alpha_break(spectrum)
```

all analysis in
SDSSData module

parameters read from
configuration file

Extracting the “ra” requires detailed knowledge of the file, but it’s basically bookkeeping (open FITS file, read HDU x, extract header with keyword “xxx”, convert to float, etc.).

Create an object for each “thing” you work with – a spectrum, a data file, etc. Put functions into those objects that do the bookkeeping, hiding it from your analysis code.

Application Programming Interface (API)

- An application programming interface defines functions for certain tasks or data.
- The implementation is hidden in the library.
- Changes to how the task is performed can be implemented, but the API stays the same.

“ra” is an API call



```
from SDSSData import Spectrum  
  
s = Spectrum(file="spectrum_file.fits")  
print s.ra()
```

The details of how “ra” is looked up don’t matter – they are in the library.
If the file format changes, change the library. Any code that uses the API still works.

\$PYTHONPATH

When you import a package, where does Python look?

```
>>> import mycustompackage
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'mycustompackage'
>>>
```

First, Python looks inside the Python installation. If not found, it checks each directory defined in the environment variable \$PYTHONPATH. There is no default \$PYTHONPATH; you create it yourself.

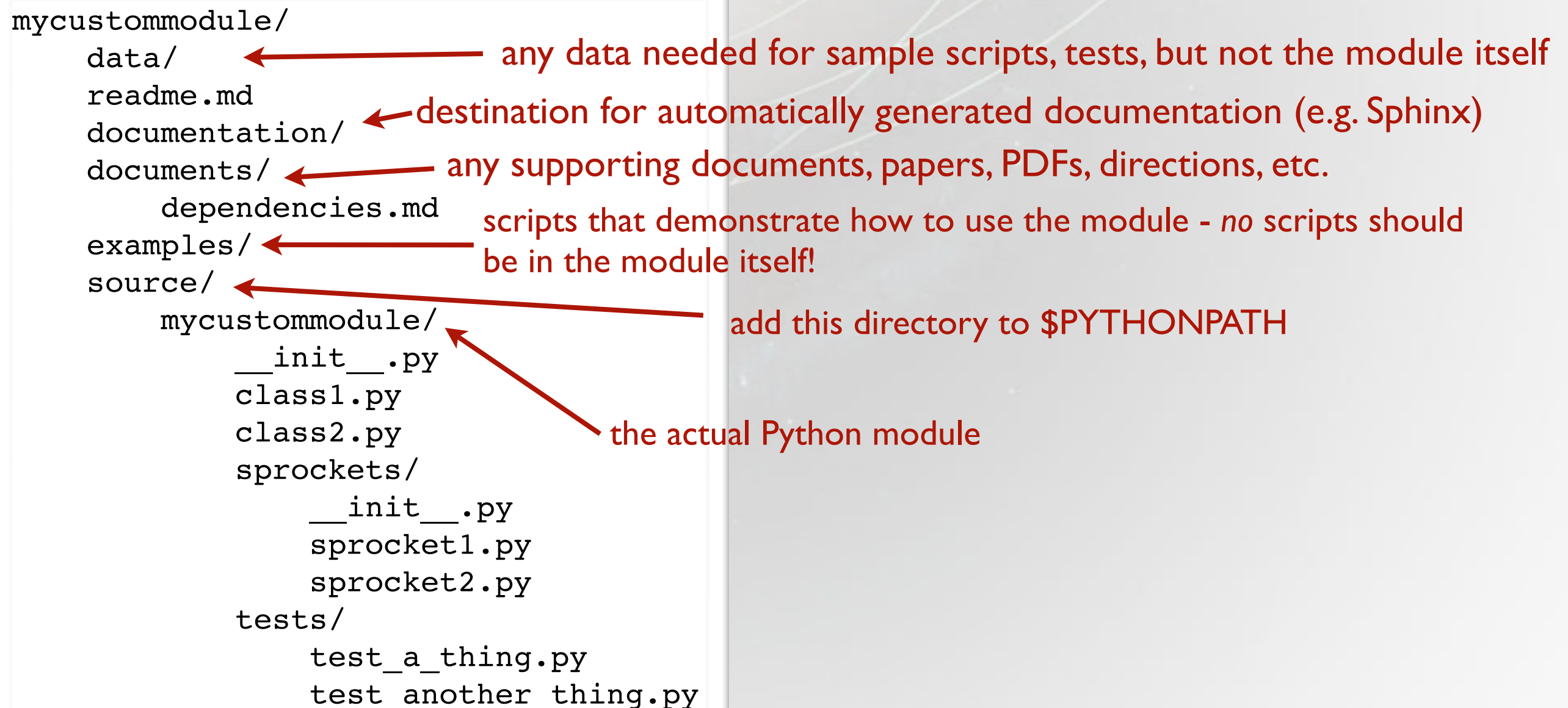
```
% export PYTHONPATH="/Users/demitri/Documents/Repositories/mycustompackage/python"
```

Above, I add the directory from my repository that *contains* the Python package. As with \$PATH, you can add as many as you like, separated by ":". Now this works without having to install the module or put it anywhere special. This is useful for development; eventually you'll want to make an installable package.

```
>>> import mycustompackage
>>>
```

Repository Directory Structure

Each Python module you create should probably have its own repository. This is how I'd recommend creating the directory structure:



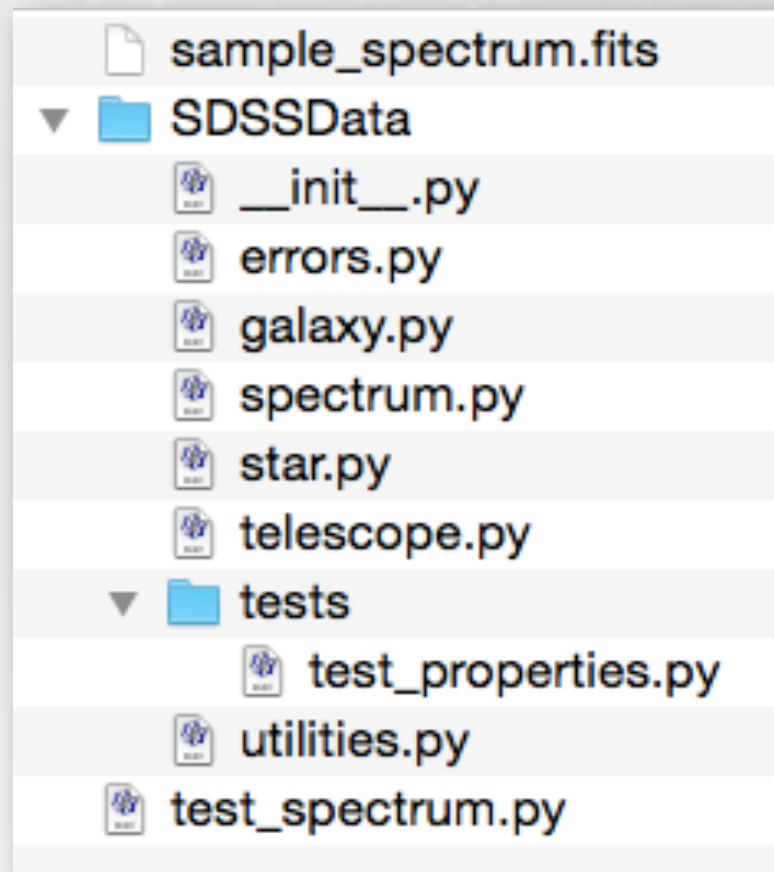
Our Own Python Modules

test_spectrum.py

```
from SDSSData.spectrum import Spectrum  
  
s = Spectrum(file="spectrum_file.fits")  
print s.ra
```

← “SDSSData” is a module we write


directory structure for our module
(while in development)



- If a file called `__init__.py` is present in a folder, Python will see the folder as a module.
- The file can be empty, or can contain initialization for the module.
- `import SDSSData` executes `__init__.py`.
- Add the enclosing folder to your `$PYTHONPATH`.

Example Spectrum.py File

“.” says to look for “errors.py” in this directory. This is a *relative import*; you will also see “..” for two directories up. Our custom errors are defined there.



```
from .errors import SDSSFileNotSpecified

class Spectrum(object):

    def __init__(self, filepath=None):
        if filepath is None:
            raise SDSSFileNotSpecified("A spectrum file must "
                                       "be specified to create a spectrum.")

        self.filepath = filepath
        self.datafile = open(self.filepath)
        self._ra = None

    @property
    def ra(self):
        ''' Returns the RA of this spectrum in degrees. '''
        if self._ra == None:
            # open the FITS file
            # read the right HDU
            self._ra = hdu.header["ra"]
        return self._ra
```


Python Properties

Let's look at our custom property 'ra'.

```
@property
def ra(self):
    if self._ra == None:
        # open the FITS file
        # read the right HDU
        self._ra = hdu.header["ra"]
    return self._ra
```

Python method decorator

Note we have two methods with the same name. The decorator distinguishes them.

This is ok if it's a read-only value. But what if we allow RA to be updated in the file? Since the value is actually in the FITS header, we again need a function to set it. In this case, we again use a method decorator to accomplish this.

This hides the implementation details from the user and makes the object more natural to work with.

Getting the RA value requires a function, so technically it would need to be called as:

```
ra = spectrum.ra()
```

But we don't think of RA as a function, we want to treat it as a property. The `@property` decorator on the function hides the fact that it's a method and allows us to access it as a property.

```
ra = spectrum.ra
```

The name of the property is part of the decorator.

```
@ra.setter
def ra(self, new_ra):
    # code to update the FITS header
```

Now this works:

```
ra = spectrum.ra
spectrum.ra = 12.34 * u.hour
```


Command Line Programs

We're familiar with command line programs and how to print the help that shows how to use the program:

```
% gzip --help
Apple gzip 242
usage: gzip [-123456789acdfhklLNnqrtVv] [-S .suffix] [<file> [<file> ...]]
  -1 --fast           fastest (worst) compression
  -2 .. -8           set compression level
  -9 --best          best (slowest) compression
  -c --stdout         write to stdout, keep original files
                    --to-stdout
  -d --decompress     uncompress files
                    --uncompress
  -f --force          force overwriting & compress links
  ...
```

standard way to get help

all available flags

short description of parameter

short flag (one character), more verbose (descriptive!) alternative

It would be nice if our programs could have this kind of output (not to mention parsing any flags we want!), but that seems like a lot of work. Luckily it's not!

The argparse Package

`argparse` is a Python package that parses command line arguments for your command line programs, and automatically generates the help output we saw before. I keep the following code handy as a template to paste into my new programs and just customize it.

```
import argparse

parser = argparse.ArgumentParser(description="This program does stuff",
                                usage="name_of_script ...",
                                epilog="this is text added to the end of the help")

# argument: files to process
parser.add_argument("-f", "--files",
                    dest="files",
                    default=None,
                    help="the list of files to process",
                    nargs="+")

args = parser.parse_args()

plateruns = args.files
```

Annotations for the code above:

- `specify the long and short flag names` (points to `"-f", "--files"`)
- `the name of the variable that will contain the result` (points to `dest="files"`)
- `any default value if the flag is not given` (points to `default=None`)
- `the short help string` (points to `help="the list of files to process"`)
- `all parameters are now contained in "args"` (points to `args = parser.parse_args()`)

A long arrow points from the `args` variable to `args.files` in the final line.

That's it! You can add as many parameters as needed.

Adding Parameters

```
# simplest option, read as args.flag
parser.add_argument("--flag")

# default values can be specified
parser.add_argument("--flag", default="semaphore")

# simple argument, read as args.flag
parser.add_argument("-f", "--flag", help="this is a kind of flag")

# "-" automatically converted to "_", read as "args.long_name"
parser.add_argument("-l", "--long-name", help="this is a kind of flag")

# put the value into your own variable, read as "some_variable"
parser.add_argument("-f", "--flag", dest="some_variable", help="I need somebody")

# required option (keyword can be T/F)
parse.add_argument("-f", "--flag", help="this is a kind of flag", required=True)
```

Adding Parameters

```
# accepts more than one argument, e.g. % script.py -f file1 file2
parser.add_argument("-f", "--files", help="this is a kind of flag", nargs="+")

# convert the inputs to a given type
parser.add_argument("-c", "--count", type=int)
parser.add_argument("-f", "--filename", type=file)

# limit input to a list of choices
parser.add_argument("-c", "--count", choices=range(10))
parser.add_argument("-l", "--letters", choices=["a", "b", "c"])

# boolean option
parser.add_argument("-f", "--flag", dest="variable_name", action="store_true")
```

Handling Errors

How do we handle problems in a library class? Printing a statement to the command line is not a good idea (what if there is no command line?) Quitting the program is the worst idea!

```
class SprocketFileNotValidException(Exception):  
    pass
```

← custom exception class
(always use "Extension" as a suffix)

```
class Sprocket(object):  
    ''' This class represents a data file. '''  
    def __init__(self, file=None):  
        if file is None:  
            raise FileNotFoundError("The file {0} was not found!".format(file))  
        self._isValid(file)  
  
    def _isValid(self, file=None):  
        # check if file is valid  
        raise SprocketFileNotValidException("The file {0} is not valid \\  
because...".format(file))
```

← uses standard exception from Python

← custom exception

```
try:  
    sprocket = Sprocket(file="sprocket_file.fits")  
except FileNotFoundError:  
    # handle exception  
except SprocketFileNotValidException:  
    # handle exception
```

The library returns errors in such a way that it allows the user to decide how to handle them.

Testing Your Code

- How do we know our code works?
- We check the code as we're writing it.
- As code base grows, how do we know all of it is being run?
- When we make changes, how do we make sure we haven't broken anything that worked before? (This is called *regression*.)
- Testing my code takes time – I'm trying to publish!

Unit Testing

- The `pytest` module provides a framework to make testing easy.
- Write small tests to check each piece of functionality, not a huge program to test everything.
- Run tests regularly as you write code to avoid regression.
- Best practice: write the tests *before* you write your code, then write your code until the tests pass.

Sample Test File

known file kept in testing directory to check against



```
import numpy as np
from ..Spectrum import Spectrum

sample_spectrum_filename = "sample_spectrum.fits"

def test_spectrum_read():
    ''' Check that we can read a spectrum file. '''
    s = Spectrum("sample_spectrum_filename")
    assert len(s.hdu_list) == 3, "Unexpected number of HDUs found in file."

def test_ra():
    ''' Check that we can read the RA from an SDSS file. '''
    s = Spectrum("sample_spectrum_filename")
    np.testing.assert_approx_equal(s.ra, 12.3432)
```

Use multiple asserts to check any functionality to be tested. Use them liberally!

The numpy testing.assert_approx_equal method is useful to check floating point values.

Running Tests

On the command line, go to the testing directory in your module and run “`py.test`” (installed when you install the `pytest` module).

```
% py.test
===== test session starts =====
platform darwin -- Python 2.7.5 -- pytest-2.3.4
collected 14 items

test_spectrum.py .. ← two tests found, two tests passed
test_galaxy.py ..FF ← four tests found; two passed, two failed
```

When tests fail, `py.test` produces output to tell you which ones failed and where.

Write your tests first and keep writing code until they all pass. Run tests regularly to make sure nothing you do breaks existing code.

Exercise: fizzbuzz Unit Test

Create a function called "fizzbuzz" in a file called "fizzbuzz.py".

```
#!/usr/bin/env python  
  
def fizzbuzz(x):  
    pass
```

In another file in the same directory, create another file called "test_fizzbuzz.py".

```
#!/usr/bin/env python  
  
from fizzbuzz import fizzbuzz
```

Implement the function until it satisfies the given inputs and outputs.
Write multiple test cases in the unit test file.