

Using Python for Data

Useful Packages

- [astropy](http://www.astropy.org/) (<http://www.astropy.org/>): Includes functions for reading/writing data files (including `.fits`), cosmology calculations, astronomical constants and coordinate systems, image processing, and much more
- [numpy](http://www.numpy.org/) (<http://www.numpy.org/>): Adds ability to deal with multi-dimensional arrays and vectorized math functions
- [scipy](http://www.scipy.org/) (<http://www.scipy.org/>): Extends `numpy` by adding common scientific functions such as ODE integration, statistical analysis, linear algebra, and FFT
- [matplotlib](http://matplotlib.org/) (<http://matplotlib.org/>): A useful plotting package
- [astroML](http://www.astroml.org/) (<http://www.astroml.org/>): Common statistical analysis and machine learning tools used in astronomy
- [scikit-learn](http://scikit-learn.org/stable/) (<http://scikit-learn.org/stable/>): More machine learning tools written in python

Installing python

The easiest way to install python on any OS is to use [anaconda python](https://www.continuum.io/downloads) (<https://www.continuum.io/downloads>). This will install a local version of python on your system so you don't need to worry about needing admin to install new packages. Most of the packages listed above are installed by default with anaconda. For this class we will be using python 3, and I recommend you use this version for your research (unless you have a very good reason to use python 2).

Text editors

Although there are numerous IDEs (e.g. IDLE, Spyder) for python, for most everyday use you will likely be writing python code in a text editor and running your programs via the command line. In this case it is important to have a good text editor that supports syntax highlighting and possibly live linting (syntax and style checking). I use the [atom](https://atom.io/) (<https://atom.io/>) text editor, a 'hackable' text editor that offers a large range of add-ons to support your coding style. If you decide to use atom you will want the following add-ons: `language-python`, `linter`, `linter-python`, and the python packages `pylama` and `pylama-pylint` installed. As a bonus the atom editor has full support for `git` and `git-hub`.

Coding style

When working on code with others, it is helpful to define a coding style for a project. That way the code is written in a predictable way and it is easy to read. Many projects use [PEP 8](https://www.python.org/dev/peps/pep-0008/) (<https://www.python.org/dev/peps/pep-0008/>) as a starting point for a style.

Basic syntax examples

For a general overview of python's syntax head over to [codecademy](https://www.codecademy.com/learn/python) (<https://www.codecademy.com/learn/python>) and take their interactive tutorial. In this class we will only be covering what is necessary for data analysis.

importing packages

Any package or code from another `.py` file can be imported with a simple `import` statement. By default all imported code has its own name space, so you don't have to worry about overwriting existing functions. The final line of this code block is a "magic" Jupyter function needed to make interactive plots inside of Jupyter notebooks.

```
In [3]: import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
%matplotlib notebook
```

data containers

Data inside of python can be stored in several different types of contains. The most basic ones are:

- **list**: an indexed data structure that can hold any objects as an element
- **tuple**: same as a **list** except the data is immutable
- **dictionary**: objects stored as a {key: value} set (note: any immutable object can be used as a key including a tuple)

```
In [1]: example_list = [1, 2, 3]
        example_tuple = (1, 2, 3)
        example_dict = {'key1': 1, 'key2': 2, ('key', 3): 3}
```

Elements in these objects can be accessed using an zero-based index (**list** and **tuple**) or key (**dict**).

```
In [3]: print(example_list[0], example_list[-1])
        print(example_tuple[1])
        print(example_dict['key1'], example_dict[('key', 3)])

1 3
2
1 3
```

Each of these objects have various methods that can be called on them to do various things. To learn what methods can be called you can look at the python documentation (e.g. <https://docs.python.org/3/tutorial/datastructures.html> (<https://docs.python.org/3/tutorial/datastructures.html>)) or you can inspect the object directly and use python's `help` function to get the doc string.

Note: Methods that start with `__` or `_` are private methods that are not designed to be called directly on the object.

```
In [4]: print(dir(example_list))
        help(example_list.pop)

['_add_', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
Help on built-in function pop:

pop(...) method of builtins.list instance
    L.pop([index]) -> item -- remove and return item at index (default last).
    Raises IndexError if list is empty or index is out of range.
```

Slicing lists

Many times it is useful to slice and manipulate lists:

```
In [5]: a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(a)
# print the first 3 elements
print(a[:3])
# print the middle 4 elements
print(a[3:7])
# print the last 3 elements
print(a[7:])
# you can also use neg index
print(a[-3:])
# print only even index
print(a[::2])
# print only odd index
print(a[1::2])
# print the reverse list
print(a[::-1])
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2]
[3, 4, 5, 6]
[7, 8, 9]
[7, 8, 9]
[0, 2, 4, 6, 8]
[1, 3, 5, 7, 9]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Looping over lists and dicts

There are several ways to loop over a list or dict depending on what values you want access to.

```
In [6]: # loop over values in a list
for i in example_list:
    print(i)
print('=====')

# loop over values in a list with index
for idx, i in enumerate(example_list):
    print('{0}: {1}'.format(idx, i))
print('=====')

# loop over keys in dict
for i in example_dict:
    print(i)
print('=====')

# loop over values in dict
for i in example_dict.values():
    print(i)
print('=====')

# loop over keys and values in dict
for key, value in example_dict.items():
    print('{0}: {1}'.format(key, value))
```

```
1
2
3
=====
0: 1
1: 2
2: 3
=====
('key', 3)
key2
key1
=====
3
2
1
=====
('key', 3): 3
key2: 2
key1: 1
```

list/dict comprehension

If you need to make a list or dict as the result of a loop you can use comprehension. **Note** comprehension is faster than a normal loop since the iteration uses the map function that is compiled in C.

```
In [7]: # slower method
list_loop = []
dict_loop = {}
for i in a:
    list_loop.append(i**2)
    dict_loop['key{0}'.format(i)] = i
print(list_loop)
print(dict_loop)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
{'key7': 7, 'key2': 2, 'key1': 1, 'key4': 4, 'key3': 3, 'key8': 8, 'key5': 5,
'key9': 9, 'key6': 6, 'key0': 0}
```

```
In [10]: # faster method
list_comp = [i**2 for i in a]
dict_comp = {'key{0}'.format(i): i for i in a}
print(list_comp)
print(dict_comp)

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
{'key7': 7, 'key2': 2, 'key1': 1, 'key4': 4, 'key3': 3, 'key8': 8, 'key5': 5,
'key9': 9, 'key6': 6, 'key0': 0}
```

Writing reusable code

It is always best to keep your code DRY (don't repeat yourself). If you find yourself writing the same block of code more than 2 times you should think about extracting it to a function. If you need to create a custom object that has its own methods assigned to it you should create a custom class.

functions

In python functions use a local name space, so don't worry about reusing variable names. Only if a variable is not in the local name space will the function look to the global name space. If the function argument is immutable it will be local in scope, otherwise it will not.

```
In [11]: def alpha(x):
          x = x + 1
          return x

x = 1
print(alpha(x))
print(x)

def beta(x):
    x[0] = x[0] + 1
    return x

x = [1]
print(beta(x))
print(x)

2
1
[2]
[2]
```

classes

Classes are useful when you will have multiple instances of an object type:

```
In [12]: class Shape:
    def __init__(self, x, y, cx=0.0, cy=0.0):
        self.name = 'rectangle'
        self.x = x
        self.y = y
        self.cx = cx
        self.cy = cy

    def area(self):
        return self.x * self.y

    def move(self, dx, dy):
        self.cx += dx
        self.cy += dy

    def get_position(self):
        return '[x: {0}, y: {1}]'.format(self.cx, self.cy)

class Square(Shape):
    def __init__(self, x, cx=0.0, cy=0.0):
        self.name = 'square'
        self.x = x
        self.y = x
        self.cx = cx
        self.cy = cy

class Circle(Shape):
    def __init__(self, r, cx=0.0, cy=0.0):
        self.name = 'circle'
        self.r = r
        self.cx = cx
        self.cy = cy

    def area(self):
        '''Return the area of the circle'''
        return np.pi * self.r**2

shape_list = [Shape(1, 2), Square(3), Circle(5)]
for sdx, s in enumerate(shape_list):
    s.move(sdx, sdx)
    print('{0} area: {1}, position: {2}'.format(s.name, s.area(), s.get_position()))

rectangle area: 2, position: [x: 0.0, y: 0.0]
square area: 9, position: [x: 1.0, y: 1.0]
circle area: 78.53981633974483, position: [x: 2.0, y: 2.0]
```

As demonstrated before, you can show all the methods available to a class by using the `dir` function. If a docstring is defined (triple quote comment on the first line of a function) it will be displayed if `help` is called on the function.

```
In [13]: print(dir(Circle))
print(help(shape_list[2].area))

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'area', 'get_position', 'move']
Help on method area in module __main__:

area() method of __main__.Circle instance
    Return the area of the circle

None
```

```
if __name__ == '__main__':
```

Sometimes you want a file to run a bit of code when called directly from the command line, but not call that code if it is imported into another file. This can be done by checking the value of the global variable `__name__`, when a bit of code is directly run `__name__` will be `'__main__'`, when imported it will not.

```
In [14]: if __name__ == '__main__':  
        # code that is only run when this file is directly called from the command  
        line  
        # This is a good place to put example code for the functions and classes de  
        fined in the file  
        print('An example')
```

An example

with blocks

When working with objects that have `__enter__` and `__exit__` methods defined, you can use a `with` block to automatically call `__enter__` at the start and `__exit__` at the end. A typical use case is automatically closing files after you are done reading/writing data:

```
In [15]: with open('data.csv', 'r') as file:  
        print(file.readline())  
  
        print(file.readline())
```

ID, x, y, sy, sx, pxy

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-15-be2d04b70708> in <module>()  
      2     print(file.readline())  
      3  
----> 4 print(file.readline())  
  
ValueError: I/O operation on closed file.
```

Numpy

NumPy extends Python to provide n-dimensional arrays along with a wealth of statistical and mathematical functions.

```
In [16]: b = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  
        print(b)
```

```
[[1 2 3]  
 [4 5 6]  
 [7 8 9]]
```

There are several ways to create arrays of a given size:

```
In [17]: zero = np.zeros((2, 2, 3))
print(zero)
one = np.ones((2, 4))
print(one)
empty = np.empty((3, 3))
print(empty)
```

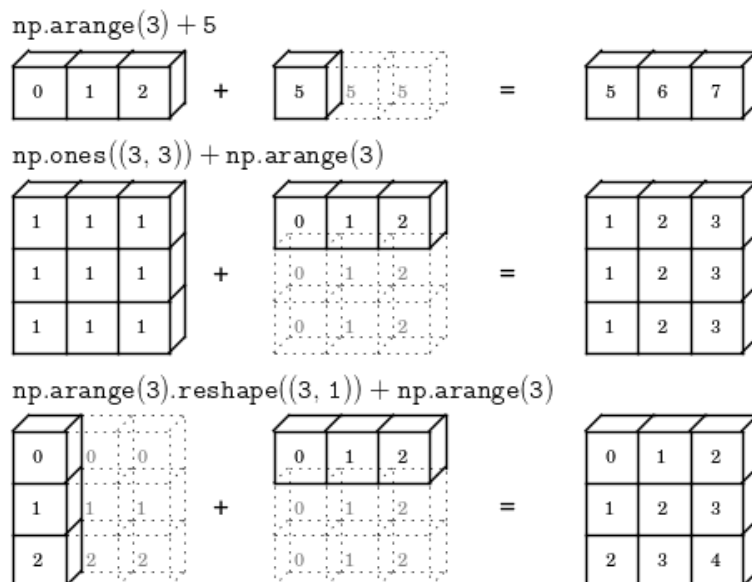
```
[[[ 0.  0.  0.]
  [ 0.  0.  0.]
```

```
[[ 0.  0.  0.]
 [ 0.  0.  0.]]]
[[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]
```

Note: empty fills the array with whatever happened to be in that bit of memory earlier!

Basic operations

Arrays typically act element by element or try to cast the operations in "obvious" ways:



-image ref: <http://www.astroML.org> (<http://www.astroML.org>)


```
In [18]: print(b)
print('=====')

print (b + b)
print('=====')

print (3 * b)
print('=====')

d = np.array([1, 2, 3])
print(d)
print (b + d)
print('=====')

e = np.array([[1], [2], [3]])
print(e)
print (b + e)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
=====
[[ 2  4  6]
 [ 8 10 12]
 [14 16 18]]
=====
[[ 3  6  9]
 [12 15 18]
 [21 24 27]]
=====
[1 2 3]
[[ 2  4  6]
 [ 5  7  9]
 [ 8 10 12]]
=====
[[1]
 [2]
 [3]]
[[ 2  3  4]
 [ 6  7  8]
 [10 11 12]]
```

Methods

Arrays also have methods such as `sum()`, `min()`, `max()` and these also take axis arguments to operate just over one index.

```
In [19]: print(b.sum())
print(b.sum(axis=0))
print(b.sum(axis=1))
```

```
45
[12 15 18]
[ 6 15 24]
```

Slices

Works the same as lists, just provide a slice for each dimension:

```
In [20]: print(b[0, 0:2])
print(b[:, 0:2])
print(b[0:2, 2:])
```

```
[1 2]
[[1 2]
 [4 5]
 [7 8]]
[[3]
 [6]]
```

Iterating

When using an array as an iterator it will loop over the first index of the array (e.g. for a 2d array it loops row-by-row). Loop over the resulting object to loop over the second index, etc...

```
In [21]: for row in b:
print(row)
    for col in row:
print(col)
```

```
[1 2 3]
1
2
3
[4 5 6]
4
5
6
[7 8 9]
7
8
9
```

Masking arrays

Many times you want to find the values in an array to pass a particular condition (e.g. $B-V < 0.3$). This can be done with array masks:

```
In [22]: mask = b >= 5
print(mask)
print(b[mask])
```

```
[[False False False]
 [False  True  True]
 [ True  True  True]]
[5 6 7 8 9]
```

You can also combine multiple masks with the *bitwise* comparison operators (&, |, ~, ^):

```
In [23]: mask2 = b <= 7
print(mask2)
print(b[mask & mask2])
print(b[mask | mask2])
print(b[~mask | mask2])

[[ True  True  True]
 [ True  True  True]
 [ True False False]]
[5 6 7]
[1 2 3 4 5 6 7 8 9]
[1 2 3 4 5 6 7]
```

You can also create masks based on parts of an array (e.g. the first column) and apply it to other parts of the array (e.g. the second column):

```
In [24]: mask3 = b[:, 0] <= 4
print(mask3)
print(b[:, 0][mask3])
print(b[:, 1][mask3])
print(b[:, 2][mask3])

[ True  True False]
[1 4]
[2 5]
[3 6]
```

Looking at source code

Numpy also has a function that lets you take a look at source code:

```
In [25]: np.source(plt.figure)
```

In file: /Users/coleman/anaconda/envs/python3/lib/python3.5/site-packages/matplotlib/pyplot.py

```
def figure(num=None, # autoincrement if None, else integer from 1-N
           figsize=None, # defaults to rc figure.figsize
           dpi=None, # defaults to rc figure.dpi
           facecolor=None, # defaults to rc figure.facecolor
           edgecolor=None, # defaults to rc figure.edgecolor
           frameon=True,
           FigureClass=Figure,
           **kwargs
           ):
    """
    Creates a new figure.

    Parameters
    -----

    num : integer or string, optional, default: none
        If not provided, a new figure will be created, and the figure number
        will be incremented. The figure object holds this number in a `number`
        attribute.
        If num is provided, and a figure with this id already exists, make
        it active, and returns a reference to it. If this figure does not
        exist, create it and returns it.
        If num is a string, the window title will be set to this figure's
        `num`.

    figsize : tuple of integers, optional, default: None
        width, height in inches. If not provided, defaults to rc
        figure.figsize.

    dpi : integer, optional, default: None
        resolution of the figure. If not provided, defaults to rc figure.dpi.

    facecolor :
        the background color. If not provided, defaults to rc figure.facecolor

    edgecolor :
        the border color. If not provided, defaults to rc figure.edgecolor

    Returns
    -----
    figure : Figure
        The Figure instance returned will also be passed to new_figure_manager
        in the backends, which allows to hook custom Figure classes into the
        pylab interface. Additional kwargs will be passed to the figure init
        function.

    Notes
    -----
    If you are creating many figures, make sure you explicitly call "close"
    on the figures you are not using, because this will enable pylab
    to properly clean up the memory.

    rcParams defines the default values, which can be modified in the
    matplotlibrc file

    """

    if figsize is None:
        figsize = rcParams['figure.figsize']
    if dpi is None:
        dpi = rcParams['figure.dpi']
    if facecolor is None:
        facecolor = rcParams['figure.facecolor']
    if edgecolor is None:
        edgecolor = rcParams['figure.edgecolor']
```

```
In [26]: a = 1
         b = 2
         print(a / b)
         print(a // b)
```

```
0.5
0
```

Astropy

The package is the magic that will make your astronomy code easier to write. There are already functions for many of the things you would want to do, e.g. `.fits` reading/writing, data table reading/writing, sky coordinate transformations, cosmology calculations, and more.

Reading tables

You won't want to type most data directly into your python code, instead you can use `astropy.table` (<http://docs.astropy.org/en/stable/io/unified.html>) (see also: <http://docs.astropy.org/en/stable/table/> (<http://docs.astropy.org/en/stable/table/>)) to read the data in from a file. The following data types are directly supported:

- fits
- ascii
- aastex
- basic
- cds
- daophot
- ecsv
- fixed_width
- html
- ipac
- latex
- rdb
- sextractor
- tab
- csv
- votable

For other formats you can extend the existing `table` class to support it.

```
In [28]: import astropy
         print(astropy.__version__)
```

```
1.2.1
```

```
In [27]: from astropy.table import Table
t = Table.read('data.csv', format='ascii.csv')
print(t)
print(t.info)
print(t.colnames)
```

```

ID   x    y   sy  sx  pxy
--- --- --- --- ---
  1 201 592   61   9 -0.84
  2 244 401   25   4  0.31
  3  47 583   38  11  0.64
  4 287 402   15   7 -0.27
  5 203 495   21   5 -0.33
  6  58 173   15   9  0.67
  7 202 479   27   4 -0.02
  8 202 504   14   4 -0.05
  9 198 510   30  11 -0.84
 10 158 416   16   7 -0.69
 11 165 393   14   5  0.3
 12 201 442   25   5 -0.46
 13 157 317   52   5 -0.03
 14 131 311   16   6  0.5
 15 166 400   34   6  0.73
 16 160 337   31   5 -0.52
 17 186 423   42   9  0.9
 18 125 334   26   8  0.4
 19 218 533   16   6 -0.78
 20 146 344   22   5 -0.56
```

```
<Table length=20>
```

```
name dtype
```

```

-----
ID   int64
x    int64
y    int64
sy   int64
sx   int64
pxy  float64
```

```
['ID', 'x', 'y', 'sy', 'sx', 'pxy']
```

```

/Users/coleman/anaconda/envs/python3/lib/python3.5/site-packages/astropy/table
/column.py:263: FutureWarning: elementwise comparison failed; returning scalar
instead, but in the future will perform elementwise comparison
  return self.data.__eq__(other)
```

The columns of `t` can be accessed by name:

```
In [29]: print(t['ID', 'pxy'])
```

```
ID  pxy
---  ----
1  -0.84
2   0.31
3   0.64
4  -0.27
5  -0.33
6   0.67
7  -0.02
8  -0.05
9  -0.84
10 -0.69
11   0.3
12 -0.46
13 -0.03
14   0.5
15   0.73
16 -0.52
17   0.9
18   0.4
19 -0.78
20 -0.56
```

And math can be applied:

```
In [30]: print(np.sqrt(t['sx']**2 + t['sy']**2))
```

```
      sx
-----
61.6603600379
25.3179778023
39.5600808897
16.5529453572
21.5870331449
17.4928556845
27.2946881279
14.5602197786
31.9530906173
17.4642491966
14.8660687473
25.495097568
52.2398315464
17.0880074906
34.5253530033
31.4006369362
42.9534631898
27.2029410175
17.0880074906
22.5610283454
```

If you have multiple data tables you can also stack them (vertically or horizontally) or join them (see <http://docs.astropy.org/en/stable/table/operations.html> (<http://docs.astropy.org/en/stable/table/operations.html>))

Constants and Units

Many of the constants you would need can be found in `astropy.constants` (<http://docs.astropy.org/en/stable/constants/>). You can also assign units to your values using `astropy.units` (<http://docs.astropy.org/en/stable/units/>).


```
In [31]: from astropy import constants as const  
print(const.c)
```

```
Name      = Speed of light in vacuum  
Value     = 299792458.0  
Uncertainty = 0.0  
Unit      = m / s  
Reference = CODATA 2010
```

```
In [32]: from astropy import units as u  
wavelength = [1000., 2000., 3000.] * u.nm  
print(wavelength)  
# convert to meters  
print(wavelength.to(u.m))  
# convert to frequency  
freq = wavelength.to(u.Hz, equivalencies=u.spectral())  
print(freq)  
# convert to velocity from a rest wavelength of 2000 nm  
freq_to_vel = u.doppler_optical(2000 * u.nm)  
vel = freq.to(u.km / u.s, equivalencies=freq_to_vel)  
print(vel)
```

```
[ 1000.  2000.  3000.] nm  
[ 1.000000000e-06  2.000000000e-06  3.000000000e-06] m  
[ 2.99792458e+14  1.49896229e+14  9.99308193e+13] Hz  
[-149896.229      0.      149896.229] km / s
```

```
In [ ]:
```