Using Python for Data

Useful Packages

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

Installing python

The easiest way to install python on any OS is to use anaconda python. 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 you research (unless you have a very good reason to use python 2).

Note

As of October 2019 python 2.7 is officially depreciated and will only receive security updates and in December 2021 python 3.6 will be offically depreciated as well. Many of the major packages listed above have already dropped python 2 support are are startting to drop support of python 3.6 and lower.

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 have used the atom text editor in the past, 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` .

Recently I have switched to VScode. Much the same as atom, it offers many add-ons that make writing code easier.

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 as a starting point for a style. Many linters will let you adjust what rules from PEP 8 you want to use. I use flake8 for my projects.

Basic syntax examples

For a general overview of python's syntax head over to codecademy 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 [1]:

```
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 containers. 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 [2]:

```
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) 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))
print('\n\n')
help(example_list.pop)
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir_
_', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getite
m__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subcla
ss__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new_
_', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__
setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'appen
d', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'r
everse', 'sort']



Help on built-in function pop:

pop(index=-1, /) method of builtins.list instance
    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 `list`s and `dict`s

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

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]
{'key0': 0, 'key1': 1, 'key2': 2, 'key3': 3, 'key4': 4, 'key5': 5, 'key6': 6,
'key7': 7, 'key8': 8, 'key9': 9}
```

In [8]:

```
# 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]
{'key0': 0, 'key1': 1, 'key2': 2, 'key3': 3, 'key4': 4, 'key5': 5, 'key6': 6,
'key7': 7, 'key8': 8, 'key9': 9}
```

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 [9]:

```
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 [10]:

```
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_posit
```

```
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 [11]:

```
print(dir(Circle))
print('\n\n')
print(help(shape_list[2].area))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__fo
rmat__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__in
it_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__redu
ce__', '__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 form 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 it directly run `__name__` will be `'__main__'`, when imported it will not.

In [12]:

```
if __name__ == '__main__':
    # code that is only run when this file is directly called from the comman
    # This is a good place to put example code for the functions and classes 
    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 [13]:

```
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)
/tmp/ipykernel_20630/1880574709.py 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 [14]:

```
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 [15]:

```
zero = np.zeros((2, 2, 3))
print(zero)
one = np.ones((2, 4))
print(one)
empty = np.empty((3, 3))
print(empty)
```

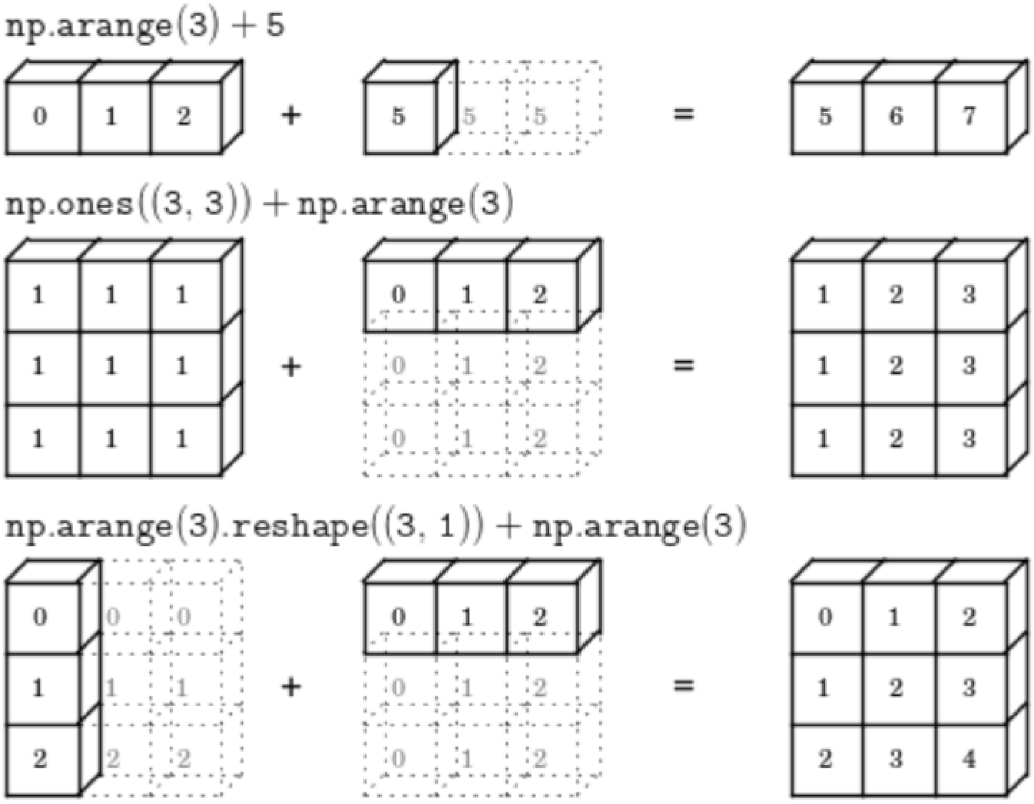
```
[[[0. 0. 0.]
  [0. 0. 0.]]

 [[0. 0. 0.]
  [0. 0. 0.]]]
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]]
[[ 6.89856097e-310  9.57802776e-317  6.89856188e-310]
 [-1.50303451e-066  6.89844398e-310  6.89856188e-310]
 [-1.01383835e-067  6.89844299e-310  3.95252517e-322]]
```

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

In [16]:

```
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 [17]:

```
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 [18]:

```
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 [19]:

```
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 [20]:

```
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 [21]:

```
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 [22]:

```
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 as a function that lets you take a look at source code:

In [24]:
```python
np.source(plt.figure)
```

```
In file: /mnt/lustre/shared_python_environment/DataLanguages/lib/python3.8/sit
e-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,
           clear=False,
           **kwargs
           ):
    """
    Create a new figure, or activate an existing figure.

    Parameters
    ----------
    num : int or str or `.Figure`, optional
        A unique identifier for the figure.

        If a figure with that identifier already exists, this figure is made
        active and returned. An integer refers to the ``Figure.number``
        attribute, a string refers to the figure label.

        If there is no figure with the identifier or *num* is not given, a new
        figure is created, made active and returned.  If *num* is an int, it
        will be used for the ``Figure.number`` attribute, otherwise, an
        auto-generated integer value is used (starting at 1 and incremented
        for each new figure). If *num* is a string, the figure label and the
        window title is set to this value.

    figsize : (float, float), default: :rc:`figure.figsize`
        Width, height in inches.

    dpi : float, default: :rc:`figure.dpi`
        The resolution of the figure in dots-per-inch.

    facecolor : color, default: :rc:`figure.facecolor`
        The background color.

    edgecolor : color, default: :rc:`figure.edgecolor`
        The border color.

    frameon : bool, default: True
        If False, suppress drawing the figure frame.

    FigureClass : subclass of `~matplotlib.figure.Figure`
        Optionally use a custom `.Figure` instance.

    clear : bool, default: False
        If True and the figure already exists, then it is cleared.

    tight_layout : bool or dict, default: :rc:`figure.autolayout`
        If ``False`` use *subplotpars*. If ``True`` adjust subplot
        parameters using `.tight_layout` with default padding.
        When providing a dict containing the keys ``pad``, ``w_pad``,
        ``h_pad``, and ``rect``, the default `.tight_layout` paddings
        will be overridden.

    constrained_layout : bool, default: :rc:`figure.constrained_layout.use`
        If ``True`` use constrained layout to adjust positioning of plot
        elements.  Like ``tight_layout``, but designed to be more
        flexible.  See
        :doc:`/tutorials/intermediate/constrainedlayout_guide`
        for examples.  (Note: does not work with `add_subplot` or
        `~.pyplot.subplot2grid`.)
```

```
    **kwargs : optional
        See `~.matplotlib.figure.Figure` for other possible arguments.

    Returns
    -------
    `~matplotlib.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 pyplot interface. Additional kwargs will be
        passed to the `.Figure` init function.

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

    `~matplotlib.rcParams` defines the default values, which can be modified
    in the matplotlibrc file.
    """
    if isinstance(num, Figure):
        if num.canvas.manager is None:
            raise ValueError("The passed figure is not managed by pyplot")
        _pylab_helpers.Gcf.set_active(num.canvas.manager)
        return num

    allnums = get_fignums()
    next_num = max(allnums) + 1 if allnums else 1
    fig_label = ''
    if num is None:
        num = next_num
    elif isinstance(num, str):
        fig_label = num
        all_labels = get_figlabels()
        if fig_label not in all_labels:
            if fig_label == 'all':
                _api.warn_external("close('all') closes all existing figure
s.")
            num = next_num
        else:
            inum = all_labels.index(fig_label)
            num = allnums[inum]
    else:
        num = int(num)  # crude validation of num argument

    manager = _pylab_helpers.Gcf.get_fig_manager(num)
    if manager is None:
        max_open_warning = rcParams['figure.max_open_warning']
        if len(allnums) == max_open_warning >= 1:
            _api.warn_external(
                f"More than {max_open_warning} figures have been opened. "
                f"Figures created through the pyplot interface "
                f"(`matplotlib.pyplot.figure`) are retained until explicitly "
                f"closed and may consume too much memory. (To control this "
                f"warning, see the rcParam `figure.max_open_warning`).",
                RuntimeWarning)

        manager = new_figure_manager(
            num, figsize=figsize, dpi=dpi,
            facecolor=facecolor, edgecolor=edgecolor, frameon=frameon,
            FigureClass=FigureClass, **kwargs)
        fig = manager.canvas.figure
        if fig_label:
            fig.set_label(fig_label)

        _pylab_helpers.Gcf._set_new_active_manager(manager)
```

```
            # make sure backends (inline) that we don't ship that expect this
            # to be called in plotting commands to make the figure call show
            # still work.  There is probably a better way to do this in the
            # FigureManager base class.
            draw_if_interactive()

            if _INSTALL_FIG_OBSERVER:
                fig.stale_callback = _auto_draw_if_interactive

        if clear:
            manager.canvas.figure.clear()

        return manager.canvas.figure
```

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` (see also: 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 [25]:
```
import astropy
print(astropy.__version__)
```

```
4.3.1
```

In [26]:
```
from astropy.table import Table
```

```
t = Table.read('data.csv', format='ascii.csv')
display(t)
print(t.info)
print(t.colnames)
```

Table length=20

ID	x	y	sy	sx	pxy
int64	**int64**	**int64**	**int64**	**int64**	**float64**
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']
```

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

In [27]:

```
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 [28]:
```
print(np.sqrt(t['sx']**2 + t['sy']**2))
```

```
        sx
------------------
 61.66036003787198
25.317977802344327
 39.56008088970496
 16.55294535724685
21.587033144922902
  17.4928556845359
27.294688127912362
14.560219778561036
31.953090617340916
 17.46424919657298
14.866068747318506
25.495097567963924
 52.23983154643591
 17.08800749063506
 34.52535300326414
31.400636936215164
 42.95346318982906
27.202941017470888
 17.08800749063506
22.561028345356956
```

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)

Constants and Units

Many of the constants you would need can be found in `astropy.constants`. You can also assign units to your values using `astropy.units`.

In [29]:
```
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 2018
```

In [30]:
```
from astropy import units as u
wavelength = [1000., 2000., 3000.] * u.nm
```

```
print(wavelength)
# convert to meters
print(wavelength.to(u.m))
# convert to frequncy
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.e-06 2.e-06 3.e-06] m
[2.99792458e+14 1.49896229e+14 9.99308193e+13] Hz
[-149896.229       0.      149896.229] km / s
```

Pandas

Data tables can also be read in with pandas:

In [31]:

```
import pandas
data = pandas.read_csv('data.csv')
display(data)
print(data.columns)
```

	ID	x	y	sy	sx	pxy
0	1	201	592	61	9	-0.84
1	2	244	401	25	4	0.31
2	3	47	583	38	11	0.64
3	4	287	402	15	7	-0.27
4	5	203	495	21	5	-0.33
5	6	58	173	15	9	0.67
6	7	202	479	27	4	-0.02
7	8	202	504	14	4	-0.05
8	9	198	510	30	11	-0.84
9	10	158	416	16	7	-0.69
10	11	165	393	14	5	0.30
11	12	201	442	25	5	-0.46
12	13	157	317	52	5	-0.03
13	14	131	311	16	6	0.50
14	15	166	400	34	6	0.73
15	16	160	337	31	5	-0.52
16	17	186	423	42	9	0.90
17	18	125	334	26	8	0.40
18	19	218	533	16	6	-0.78
19	20	146	344	22	5	-0.56

```
Index(['ID', 'x', 'y', 'sy', 'sx', 'pxy'], dtype='object')
```

The columns can be accessed with 'dot' notation or name

In [32]:

```
print(data.x)
print(data[['x', 'y']])
```

```
0     201
1     244
2      47
3     287
4     203
5      58
6     202
7     202
8     198
9     158
10    165
11    201
12    157
13    131
14    166
15    160
16    186
17    125
18    218
19    146
Name: x, dtype: int64
      x    y
0   201  592
1   244  401
2    47  583
3   287  402
4   203  495
5    58  173
6   202  479
7   202  504
8   198  510
9   158  416
10  165  393
11  201  442
12  157  317
13  131  311
14  166  400
15  160  337
16  186  423
17  125  334
18  218  533
19  146  344
```

As before math can be done directly on the columns

In [33]:

```
print(np.sqrt(data.sx**2 + data.sy**2))
```

```
0     61.660360
1     25.317978
2     39.560081
3     16.552945
4     21.587033
5     17.492856
6     27.294688
7     14.560220
8     31.953091
9     17.464249
10    14.866069
11    25.495098
12    52.239832
13    17.088007
14    34.525353
```

```
15    31.400637
16    42.953463
17    27.202941
18    17.088007
19    22.561028
dtype: float64
```

Pandas teats these `DataFrames` like databases, so most database operations (e.g. join, merge, groupby, etc...) can be done on a data table.

```
In [ ]:
```