1 Useful Python Commands

1.1 Functions

Command	Description
*	multiplication operation: `2*3` return 6.
**	power operation: `2**3` returns 8.
lambda	Used for create anonymous one line functions of the form: `f = lambda x,y: 5*x+y` The code after the lambda but before variables specifies the parameters. The code after the colon tells python what object to return.

def	The def command is used to create functions of more than one line:
	
	`def g(x,y):`
	`"""`
	`Docstring`
	`"""`
	`    ret = sin(x)`
	`    return ret + y`
	
	The code immediately following `def` names the function, in this example `g`. The variables in the parenthesis are the parameters of the function. The remaining lines of the function are denoted by tab indents. The return statement specifies the object to be returned.

`len(iterable)`	`len` is a function that takes an iterable, such as a list, tuple or numpy array and returns the number of items in that object. For a numpy array, `len` returns the length of the outermost dimension `len( np.zeros( (5,4) ) )` returns 5.
`l = [a1,a2,...,an]`	Constructs a list containing the objects a1,a2,...,an. You can append to the list using l.append(). The ith element of l can be accessed using `l[i]`
`t =(a1,a2,...,an)`	Constructs a tuple containing the objects $a1, a2, ..., an$. The ith element of t can be accessed using `t[i]`

`for a in iterable:`	For loop used to perform a sequence of commands (denoted using tabs) for each element in an iterable object such as a list, tuple, or numpy array. An example code is
	
	`l  = []`
	`for i in [1,2,3]:`
	`     l.append(i**2)`
	`print(l)`
	
	prints `[1,4,9]`

`if condition:`	Performs code if a condition is met (using tabs). For example
	
	`if x == 5:`
	`     x = x**2`
	`else:`
	`     x = x**3`
	
	squares x if x is 5, otherwise cubes it.

`plt.plot(x,y,s =None)`	The plot command is included in `matplotlib.pyplot`. The plot command is used to plot x versus y where x and y are iterables of the same length. By default the plot command draws a line, using the s argument you can specify type of line and color. For example '-','- -',':','o','x', and '-o' reprent line, dashed line, dotted line, circles, x's, and circle with line through it respectively. Color can be changed by appending 'b','k','g' or 'r', to get a blue, black, green or red plot respectively. For example,
	
	`import numpy as np`
	`import matplotlib.pyplot as plt`
	`x=np.linspace(0,10,100)`
	`N=len(x)`
	`v= np.cos(x)`
	`plt.figure(1)`
	`plt.plot(x,v,'-og')`
	`plt.show()`
	`plt.savefig('tom_test.eps')`
	
	plots the cosine function on the domain (0,10) with a green line with circles at the points x, v

1.2 Numpy Arrays

Command	Description

`np.array(object,dtype = None)`	`np.array` constructs a numpy array from an object, such as a list or a list of lists. `dtype` allows you to specify the type of object the array is holding. You will generally note need to specify the `dtype`. Examples:
	
	`np.array([1, 2, 3]) #creates 1 dim array of ints`
	`np.array( [1,2,3.0] )#creates 1 dim array of floats`
	`np.array( [ [1,2],`
	`            [3,4] ]) #creates a 2 dim array`

`A[i1,i2,...,in]`	Access a the element in numpy array A in with index i1 in dimension 1, i2 in dimension 2, etc. Can use : to access a range of indices, where `imin`:imax represents all i such that $imin \leq i < imax$. Always returns an object of minimal dimension. For example,
	
	`A[:,2]`
	
	returns the 2nd column (counting from 0) of A as a 1 dimensional array and
	
	`A[0:2,:]`
	
	returns the 0th and 1st rows in a 2 dimensional array.

`np.zeros( shape )`	Constructs numpy array of shape shape. Here shape is an integer of sequence of integers. Such as 3, (1,2),(2,1), or (5,5). Thus
	
	`np.zeros( (5,5) )`
	
	Constructs an 5×5 array while
	
	`np.zeros(5,5)`
	
	will throw an error.

`np.ones(shape)`	Same as `np.zeros` but produces an array of ones

<table>
<tr><td><code>np.linspace(a,b,n)</code></td><td>Returns a numpy array with n linearly spaced points between a and b. For example

<code>np.linspace(1,2,10)</code>

returns

<pre>array([1.   , 1.11111111, 1.22222222, 1.33333333,
        1.44444444, 1.55555556, 1.66666667, 1.77777778,
        1.88888889, 2.   ])</pre></td></tr>
<tr><td><code>np.eye(N)</code></td><td>Constructs the identity matrix of size N. For example

<code>np.eye(3)</code>

returns the 3×3 identity matrix:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$</td></tr>
<tr><td><code>np.diag(a)</code></td><td><code>np.diag</code> has 2 uses. First if <code>a</code> is a 2 dimensional array then <code>np.diag</code> returns the principle diagonal of the matrix. Thus

<pre>np.diag([[1,3],
           [5,6]])</pre>
returns <code>[1,6]</code> . If a is a 1 dimensional array then <code>np.diag</code> constructs an array with a as the principle diagonal. Thus,

<code>np.diag([1,2])</code>

returns
$$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$$</td></tr>
<tr><td><code>np.random.rand(d0, d1, ..., dn)</code></td><td>Constructs a numpy array of shape <code>(d0,d1,...,dn)</code> filled with random numbers drawn from a uniform distribution between $(0, 1)$. For example, <code>np.random.rand(2,3)</code> returns

<pre>array([[0.69060674, 0.38943021, 0.19128955],
       [0.5419038 , 0.66963507, 0.78687237]])</pre></td></tr>
<tr><td><code>np.random.randn(d0, d1, ..., dn)</code></td><td>Same as <code>np.random.rand(d0, d1, ..., dn)</code> except that it draws from the standard normal distribution $\mathcal{N}(0, 1)$ rather than the uniform distribution.</td></tr>
<tr><td><code>A.T</code></td><td>Reverses the dimensions of an array (transpose). For example, if $x = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ then <code>x.T</code> returns $\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$</td></tr>
</table>

`np.hstack( tuple )`	Take a sequence of arrays and stack them horizontally to make a single array. For example
	
	`a = np.array(( [1,2,3] )`
	`b = np.array( [2,3,4] )`
	`np.hstack( (a,b) )`
	
	returns $[1, 2, 3, 2, 3, 4]$ while
	
	`a = np.array( [[1],[2],[3]] )`
	`b = np.array( [[2],[3],[4]] )`
	`np.hstack((a,b))`
	
	returns $\begin{pmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \end{pmatrix}$

`np.vstack( tuple )`	Like `np.hstack`. Takes a sequence of arrays and stack them vertically to make a single array. For example
	
	`a = np.array( [1,2,3] )`
	`b = np.array( [2,3,4] )`
	`np.hstack( (a,b) )`
	
	returns
	
	`array( [ [1,2,3],`
	`          [2,3,4] ] )`

`np.amax(a, axis = None)`	By default `np.amax(a)` finds the maximum of all elements in the array a. Can specify maximization along a particular dimension with axis. If
	
	`a = np.array( [ [2,1],`
	`                 [3,4] ]) #creates a 2 dim array`
	
	then
	
	`np.amax(a,axis = 0) #maximization along row (dim 0)`
	
	returns `array([3,4])` and
	
	`np.amax(a, axis = 1) #maximization along column (dim 1)`
	
	returns `array([2,4])`

`np.amin(a, axis = None)`	Same as `np.amax` except returns minimum element.

`np.argmax(a, axis = None)`	Performs similar function to np.amax except returns index of maximal element. By default gives index of flattened array, otherwise can use axis to specify dimension. From the example for np.amax
	
	`np.amax(a,axis = 0) #maximization along row (dim 0)`
	
	returns `array([1,1])` and
	
	`np.amax(a, axis = 1) #maximization along column (dim 1)`
	
	returns `array([0,1])`

`np.argmin(a, axis =None)`	Same as `np.argmax` except finds minimal index.
`np.dot(a,b)` or `a.dot(b)`	Returns an array equal to the dot product of a and b. For this operation to work the innermost dimension of a must be equal to the outermost dimension of b. If a is a $(3, 2)$ array and b is a (2) array then `np.dot(a,b)` is valid. If b is a $(1, 2)$ array then the operation will return an error.

1.3 numpy.linalg commands

Command	Description

`np.linalg.inv(A)`	For a 2-dimensional array A. `np.linalg.inv` returns the inverse of A. For example, for a (2,2) array A
	
	`np.linalg.inv(A).dot(A)`
	
	returns
	
	`np.array( [1,0],`
	`          [0,1] ])`

`np.linalg.eig(A)`	Returns a 1-dimensional array with all the eigenvalues of A as well as a 2-dimensional array with the eigenvectors as columns. For example, `eigvals,eigvecs = np.linalg.eig(A)` returns the eigenvalues in `eigvals` and the eigenvectors in `eigvecs`. `eigvecs[:,i]` is the eigenvector of A with eigenvalue of `eigval[i]`.

`np.linalg.solve(A,b)`	Constructs array x such that `A.dot(x)` is equal to b. Theoretically should give the same answer as
	
	`Ainv = np.linalg.inv(A)`
	`x = Ainv.dot(b)`
	
	but numerically more stable.