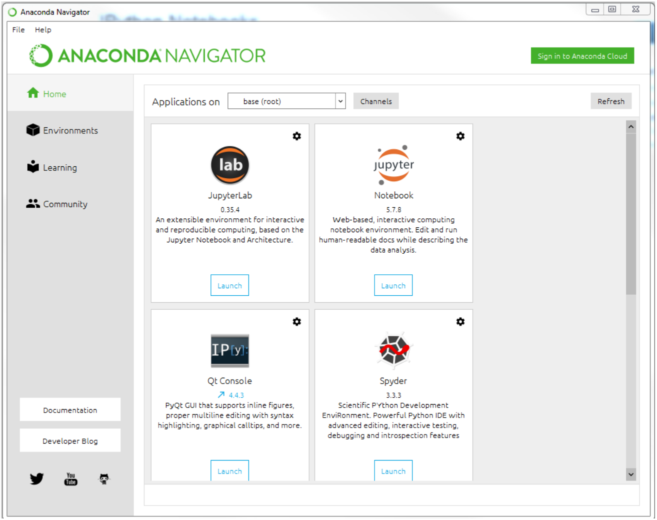
What is Python?

- Python is a general purpose high-level programming language conceived in 1989 by Dutch programmer Guido van Rossum. See his blog post on its history.
- The official Python home page is: `https://www.python.org/`.
- The recommended method to install the Python scientific stack is to use Anaconda.
- Anaconda, a free product of Continuum Analytics (`www.continuum.io`), is a virtually complete scientific stack for Python.
- Anaconda includes both the core Python interpreter and standard libraries as well as most modules required for data analysis.

What is Python?

- The open-source Anaconda Distribution is the easiest way to perform Python data science and machine learning on Linux, Windows, and Mac OS X.
- To install Anaconda, visit Anaconda Distribution web page at `https://www.anaconda.com/distribution/`.
- The recommended settings for installing Anaconda on Windows are
  - Install for all users, which requires admin privileges.
  - Add Anaconda to the System PATH - This is important to ensure that Anaconda commands can be run from the command prompt.
  - Register Anaconda as the system Python unless you have a specific reason not to (unlikely).
- After installation, open Anaconda Navigator.

Anaconda Navigator



Anaconda

Table 1: Selected libraries and packages included in Anacando

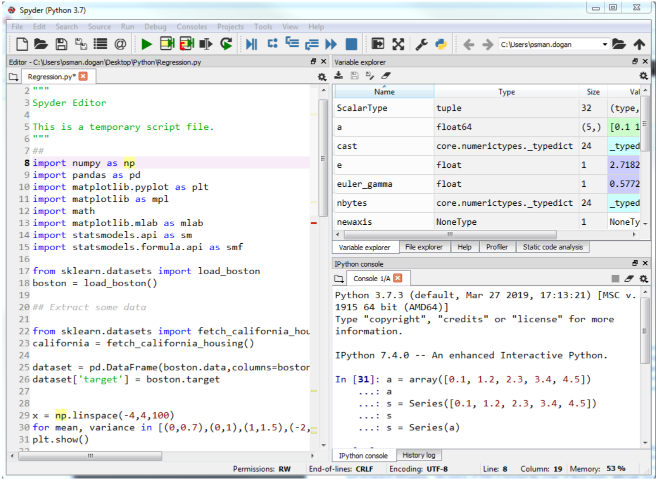
BitArray	Object types for arrays of Booleans
CubesOLAP	Framework for Online Analytical Processing (OLAP) applications
Disco	mapreduce implementation for distributed computing
Gdata	Implementation of Google Data Protocol
h5py	Python wrapper around HDF5 file format
IPython	Interactive Development Environment
lxml	Processing XML and HTML with Python
matplotlib	Standard 2D and 3D plotting library
MPI4Py	Message Parsing Interface (MPI) implementation for parallel computing
MPICH2	Another MPI implementation
NetworkX	Building and analyzing network models and algorithms
numexpr	Optimized execution of numerical expressions
NumPy	Powerful array class and optimized functions on it
pandas	Efficient handling of time series data
PyTables	Hierarchical database using HDF5
SciPy	Collection of scientific functions
Scikit-Learn	Machine learning algorithms
Spyder	Python IDE with syntax checking, debugging, and inspection capabilities
statsmodels	Statistical models
SymPy	Symbolic computation and mathematics
Theano	Mathematical expression compiler

Anaconda Navigator

- The standard Python interactive console is very basic and does not support useful features such as tab completion. The QtConsole version of IPython, transforms the console into a highly productive environment.
- The jupyter notebook is a simple and useful method to share code with others. The primary method for using notebooks is through a web interface, which allows creation, deletion, export and interactive editing of notebooks.
- Spyder is an "integrated development environment" (IDE) includes a built-in python console, code completion features and integrated debugging. It resembles to RStudio . Some other IDE's are Aptana Studio and PyCharm.

Anaconda Navigator

- Our preferred IDE is Spyder.



Modules and Documentation

- Anaconda comes with most of the libraries (modules) that we need. To see all packages available in the latest release of Anaconda, see https://docs.anaconda.com/anaconda/packages/pkg-docs/.

- To see all installed modules type `help(modules)`.

- One of the installed module is `numpy`. To access the squared root function of numpy, one of the following can be used:

```
import numpy # first option for importing numpy
numpy.sqrt(3) # return the squared root of 3

import numpy as np # second option for importing numpy
np.sqrt(3) # return the squared root of 3

from numpy import* # third option for importing numpy
sqrt(3) # return the squared root of 3
```

- In the last case, (`from numpy import*`), all functions in numpy become available.

- Any name following a dot is called an attribute of the object to the left of the dot:
  - attributes can contain auxiliary information regarding to the object
  - attributes can act like functions, called methods

Modules and Documentation

- If a module is not available in anaconda, then it should be installed. For example to install `quantecon` package, type

```
pip install quantecon
```

- Help is available in IPython sessions using `help(function)`. Some functions (and modules) have very long help files. The help documentation on numpy can be accessed by

```
print(help(np))
```

Module and Documentation

- Python, by default, only has access to a small number of built-in types and functions. The vast majority of functions are located in modules, and before a function can be accessed, the module which contains the function must be imported.
- Some important modules are
  - `numpy`, `scipy`, `matplotlib`, `seaborn`, `pandas` and `statsmodels`.
- Numpy provides a set of array and matrix data types which are essential for statistics, econometrics and data analysis.
- Scipy contains a large number of routines needed for analysis of data. The most important include a wide range of random number generators, linear algebra routines, and optimizers. scipy depends on numpy.
- Matplotlib provides a plotting environment for 2D plots, with limited support for 3D plotting. Seaborn improves the default appearance of matplotlib plots without any additional code.

Module and Documentation

- Pandas provides fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive.
- Statsmodels provides classes and functions for the estimation of many different statistical models, as well as for conducting statistical tests, and statistical data exploration.
- Another useful module is `pylab`.
- Pylab is a collection of common NumPy, SciPy and Matplotlib functions. This module can be easily imported using a single command in an IPython session, `%pylab`, which is equivalent to calling `from pylab import*`.

Built-in data types

- Built-in data types
  - Numeric
  - Boolean
  - String
  - List
  - Tuple
  - Dictionary
  - Set
  - Range

Numeric

- Variable assignment can be done in the following ways.

```
x=3
y='abc'
x, y, z = 1, 3.1415, 'a'
```

- Simple numbers in Python can be either integers, floats or complex.

```
x=3
type(x)

x=3.2
type(x)

x=2+3j
type(x)
```

- Python contains a `math` module providing functions which operate on built-in scalar data types:

addition	+	x+y
subtraction	-	x-y
multiplication	*	x*y
division	/	x/y
integer division	//	x//y=$\lfloor \frac{x}{y} \rfloor$
exponentiation	**	x**y=x^y

where $\lfloor \cdot \rfloor$ is the floor function (returns the largest integer less than or equal to x/y).

Boolean and Strings

- The Boolean data type is used to represent true and false, using the reserved keywords True and False.

```
In: x=True
In: type(x)
Out: bool
```

```
In: bool(1)
Out: True
```

```
In: bool(0)
Out: False
```

- Non-zero, non-empty values generally evaluate to true when evaluated by `bool()`. Zero or empty values such as `bool(0)`, `bool(0.0)`, `bool(0.0j)`, `bool(None)` and `bool([])` are all false.

- Strings are delimited using single quotes (' ') or double quotes (" ").

```
In : x='This is a string'
In : type(x)
Out: str
In : print(x)

This is a string
```

Slicing Strings

- Substrings within a string can be accessed using slicing. Slicing uses [ ] to contain the indices of the characters in a string. Suppose that s has n characters. Python uses $0-$based indices.

Table 2: Strings slicing

`s[:]`	return entire string
`s[i]`	return character i
`s[i:]`	return characters $i, \ldots, n-1$
`s[:i]`	return characters $0, \ldots, i-1$
`s[i:j]`	return characters $i, \ldots, j-1$
`s[i:j:m]`	return characters $i, i+m, i+2m, \ldots, i+m\lfloor\frac{j-i-1}{m}\rfloor$
`s[-i]`	return character $n-i$
`s[-i:]`	return characters $n-i, n-i+1, \ldots, n-1$
`s[:-i]`	return characters $0, 1, 2, \ldots, n-i-1$
`s[-j:-i]`	return characters $n-j, \ldots, n-i-1$ for $-j < -i$
`s[-j:-i:m]`	return characters $n-j, n-j+m, \ldots, n-j+m\lfloor\frac{j-i-1}{m}\rfloor$

- Try the followings:

```
s='Python strings are sliceable.'
print(s)
s[0]
L=len(s)
s[L]
s[:10]
s[10:]
```

Lists

- A list is a collection of other objects-floats, integers, complex numbers, strings or even other lists. Basic lists are constructed using square braces, [ ], and values are separated using commas.

```
In : x=[1,2,3,4]
In : type(x)
Out: list
In : x
Out: [1, 2, 3, 4]
In : y=[[1,2,3,4],[5,6,7]] # a list of lists
In : y
Out: [[1, 2, 3, 4], [5, 6, 7]]
In : z=[1,1.0,1+0j,'one',None,True,False,"abc"]
In : z
Out: [1, 1.0, (1+0j), 'one', None, True, False, 'abc']
```

- Basic list slicing is identical to slicing strings, and operations such as `x[:]`, `x[1:]`, `x[:1]` and `x[3:]` can all be used.

Slicing lists

- Let x be a list with n elements denoted with $x_0, x_1, \ldots, x_{n-1}$.

Table 3: Lists slicing

`x[:]`	return x
`x[i]`	return x_i
`x[i:]`	return $x_i, \ldots, x_{n-1}$
`x[:i]`	return $x_0, \ldots, x_{i-1}$
`x[i:j]`	return $x_i, \ldots, x_{j-1}$
`x[i:j:m]`	return $x_i, x_{i+m}, x_{i+2m}, \ldots, x_{i+m\lfloor\frac{j-i-1}{m}\rfloor}$
`x[-i]`	return x_{n-i}
`x[-i:]`	return $x_{n-i}, x_{n-i+1}, \ldots, x_{n-1}$
`x[:-i]`	return $x_0, x_1, x_2, \ldots, x_{n-i-1}$
`x[-j:-i]`	return $x_{n-j}, \ldots, x_{n-i-1}$
`x[-j:-i:m]`	return $x_{n-j}, x_{n-j+m}, \ldots, x_{n-j+m\lfloor\frac{j-i-1}{m}\rfloor}$

```
In : x = [[1,2,3,4], [5,6,7,8]]
In : x[1]
Out: [5, 6, 7, 8]
In : x[0][0]
Out: 1
In : x[0][1:4]
Out: [2, 3, 4]
In : x[1][-4:-1]
Out: [5, 6, 7]
```

Slicing lists

- A number of functions are available for manipulating lists. The most useful are

Table 4: Lists functions

Function	Method	Description
list.append(x,value)	x.append(value)	Appends value to the end of the list.
len(x)	-	Returns the number of elements in the list.
list.extend(x,list)	x.extend(list)	Appends the values in list to the existing list
list.pop(x,index)	x.pop(index)	Removes the value in position index and returns the value.
list.remove(x,value)	x.remove(value)	Removes the first occurrence of value from the list.
list.count(x,value)	x.count(value)	Counts the number of occurrences of value in the list.
del x[slice]	-	Deletes the elements in slice.

```
In : x = [0,1,2,3,4,5,6,7,8,9]
In : x.append(0)
In : x
Out: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
In : x.extend([11,12,13])
In : x
Out: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 11, 12, 13]
In : x.pop(1)
Out: 1
In : x
Out: [0, 2, 3, 4, 5, 6, 7, 8, 9, 0, 11, 12, 13]
In : x.remove(0)
In : x
Out: [2, 3, 4, 5, 6, 7, 8, 9, 0, 11, 12, 13]
```

Tuples (tuple)

- A tuple is virtually identical to a list with one important difference-tuples are immutable. Immutability means that a tuple cannot be changed once created.
- Tuples are immutable.

```
In : x =(1,2,3,4,5,6,7,'a','abc','ABC')
In : type(x)
Out: tuple

In : x[0]
Out: 1

In : x[-10:-5]
Out: (1, 2, 3, 4, 5)

In : x = list(x) # turning a tuple into a list
In : type(x)
Out: list

In : x = tuple(x) # turning the list back to a tuple
In : type(x)
Out: tuple

In : x= ([1,2],[3,4]) # a tuple of lists
In : type(x)
Out: tuple
In : x[1]
Out: [3, 4]

In : x[1][1] = -10
In : x # Contents can change, elements cannot
Out: ([1, 2], [3, -10])
```

Dictionary (dict)

- Dictionaries in Python are composed of keys (words) and values (definitions). Values are accessed using keys.

```
In : data = {'age': 34, 'children' : [1,2], 1: 'apple'}
In : type(data)
Out: dict

In : data['age']
Out: 34
In : data['children']
Out: [1, 2]
In : data[1]
Out: 'apple'

In : data['age'] = '40' # assign a new value to age
In : data
Out: {'age': '40', 'children': [1, 2], 1: 'apple'}

In : data['name'] = 'Joe' # create a new entry
In : data
Out: {'age': '40', 'children': [1, 2], 1: 'apple', 'name': 'Joe'}

In : del data['age']
In : data
Out: {'children': [1, 2], 1: 'apple', 'name': 'Joe'}
```

Dictionary (dict)

- Sets are collections which contain all unique elements of a collection. set and frozenset only differ in that the latter is immutable.

```
In : x = set(['MSFT','GOOG','AAPL','HPQ','MSFT'])
In : x
Out: {'AAPL', 'GOOG', 'HPQ', 'MSFT'}
In : x.add('CSCO')
In : x
Out: {'AAPL', 'CSCO', 'GOOG', 'HPQ', 'MSFT'}
In : y = set(['XOM', 'GOOG'])
In : x.intersection(y)
Out: {'GOOG'}
In : x = x.union(y)
In : x
Out: {'AAPL', 'CSCO', 'GOOG', 'HPQ', 'MSFT', 'XOM'}
```

Table 5: Sets functions

Function	Method	Description
`set.add(x,element)`	`x.add(element)`	Appends element to a set.
`len(x)`	-	Returns the number of elements in the set.
`set.difference(x,set)`	`x.difference(set)`	Returns the elements in x which are not in set.
`set.intersection(x,set)`	`x.intersection(set)`	Returns the elements of x which are also in set.
`set.remove(x,element)`	`x.remove(value)`	Removes element from the set.
`set.union(x,set)`	`x.union(set)`	Returns the set containing all elements of x and set.

range

- `range(a,b,i)` creates the sequences that follows the pattern $a, a+i, a+2i, \ldots, a+(m-1)i$ where $m = \lceil \frac{b-a}{i} \rceil$ and $\lceil \cdot \rceil$ is the ceiling function (returns the smallest integer greater than or equal to x/y).

```
In : x = range(10)
In : type(x)
Out: range

In : print(x)
range(0, 10)

In : list(x)
Out: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In : x = range(3,10,3)
In : type(x)
Out: range
In : list(x)
Out: [3, 6, 9]
```

Arrays

- Arrays are the base data type in NumPy, are in similar to lists or tuples since they both contain collections of elements.
- Arrays are initialized from lists (or tuples) using `array()`. **Higher dimensional** arrays can be initialized by **nesting lists or tuples**.

```
In : import numpy as np
In : x = [0.0, 1, 2, 3, 4]
In : y = np.array(x)
In : y
Out: array([0., 1., 2., 3., 4.])
In : type(y)
Out: numpy.ndarray

In : y = np.array([[0.0, 1, 2, 3, 4], [5, 6, 7, 8, 9]])
In : y
Out:
array([[0., 1., 2., 3., 4.],
       [5., 6., 7., 8., 9.]])
In : np.shape(y)
Out: (2, 5)

In : y = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
In : y
Out:
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])
In : np.shape(y)
Out: (2, 2, 2)
```

Arrays

- If an input contains all integers, it will have a dtype of int32. If an input contains integers, floats, or a mix of the two, the array's data type will be float64. If the input contains a mix of integers, floats and complex types, the array will be initialized to hold complex data.

```
In : x = [0, 1, 2, 3, 4] # Integers
In : y = np.array(x)
In : y.dtype
Out: dtype('int32')

In : x = [0.0, 1, 2, 3, 4] # 0.0 is a float
In : y = np.array(x)
In : y.dtype
Out: dtype('float64')

In : x = [0.0 + 1j, 1, 2, 3, 4] # (0.0 + 1j) is a complex
In : y = np.array(x)
In : y
Out: array([0.+1.j, 1.+0.j, 2.+0.j, 3.+0.j, 4.+0.j])
In : y.dtype
Out: dtype('complex128')
```

Matrices

- Matrices are essentially a subset of arrays and behave in a virtually identical manner.
- We can use `matrix()` on an array-like objects to create matrices. Alternatively, `mat()` or `asmatrix()` can be used to coerce an array to behave like a matrix without copying any data.

```
In : x=np.matrix([1,2,3,4])
In : x
Out: matrix([[1, 2, 3, 4]])
In : type(x)
Out: numpy.matrix
In : np.shape(x)
Out: (1, 4)
```

```
In : y=np.matrix((1,2,3,4))
In : y
Out: matrix([[1, 2, 3, 4]])
In : type(y)
Out: numpy.matrix
In : np.shape(y)
Out: (1, 4)
```

```
In : x = [0.0,1, 2, 3, 4] # 1 Float makes all float
In : z = np.asmatrix(x)
In : z
Out: matrix([[0., 1., 2., 3., 4.]])
In : type(z)
Out: numpy.matrix
```

Arrays and Matrices

- Consider the followings:

$$x = \begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix}, \quad y = \begin{bmatrix} 2 & 1 & 0 \\ 3 & 2 & 1 \\ 5 & 4 & 3 \end{bmatrix} \tag{1}$$

- Try the followings:

```
In : x=np.array([[2],[3],[5]])
In : x
In : np.ndim(x)

In : x=np.matrix([[2],[3],[5]])
In : x
In : np.ndim(x)

In : x=np.array([[2],[3],[5]])
In : x=np.asmatrix(x)
In : x
In : np.ndim(x)

In : y=np.array([[2,1,0],[3,2,1],[5,4,3]])
In : y
In : np.ndim(y)

In : y=np.asmatrix(y)
In : y
Out:
matrix([[2, 1, 0],
        [3, 2, 1],
        [5, 4, 3]])
In : np.ndim(y)
Out: 2
```

Arrays and Matrices

- Concatenation is the process by which one vector or matrix is appended to another.
- Try the followings:

```
In : x=np.array([[1.0,2.0],[3.0,4.0]])
In : y=np.array([[5.0,6.0],[7.0,8.0]])

In : z=np.concatenate((x,y),axis = 0)
In : z
Out:
array([[1., 2.],
       [3., 4.],
       [5., 6.],
       [7., 8.]])

In : z=np.concatenate((x,y),axis = 1)
In : z
Out:
array([[1., 2., 5., 6.],
       [3., 4., 7., 8.]])

z=np.vstack((x,y)) # Same as z = np.concatenate((x,y),axis = 0)
z=np.hstack((x,y)) # Same as z = np.concatenate((x,y),axis = 1)
```

Accessing elements of an array

- Four methods are available for accessing elements contained within an array: scalar selection, slicing, numerical indexing and logical (or Boolean) indexing

```python
# scalar selection
x = np.array([1.0,2.0,3.0,4.0,5.0])
x[0]
x = np.array([[1.0,2,3],[4,5,6]])
x[1,2]
type(x[1,2])
x = np.array([1.0,2.0,3.0,4.0,5.0])
x[0] = -5

# slicing
x = np.array([1.0,2.0,3.0,4.0,5.0])
y = x[:]
y = x[:2]
y = x[1::2]

# 2 dimensional array
y = np.array([[0.0, 1, 2, 3, 4],[5, 6, 7, 8, 9]])
y[:1,:] # Row 0, all columns
y[:,:1] # all rows, column 0
y[:1,0:3] # Row 0, columns 0 to 2
y[:1][:,0:3] # Same as previous
y[:,3:] # All rows, columns 3 and 4

# 3 dimensional array
y = np.array([[[1.0,2],[3,4]],[[5,6],[7,8]]])
y[0,:,:] # Panel 0
y[1,:,:] # Panel 1
y[0,0,0] # Row 0 and column 0 of Panel 0
y[1,1,0] # Row 1 and column 0 of Panel 1
```

QUEENS COLLEGE

Basic math for arrays and matrices

- Addition and subtraction operate element-by-element. For arrays * performs element -by-element multiplication. For the matrix multiplication @ is used.

```
In : x=np.array([[1.0, 2],[ 3, 2], [3, 4]])
In : y=np.array([[9.0, 8],[7, 6]])

In : x@y
Out:
array([[23., 20.],
       [41., 36.],
       [55., 48.]])

In : x.dot(y)
Out:
array([[23., 20.],
       [41., 36.],
       [55., 48.]])

In : x@2 # Return an error
In : x*2

In : x=np.asmatrix(x)
In : np.shape(x)
Out: (3, 2)

In : y=asmatrix(y)
In : shape(y)
Out: (2, 2)

In : x+y # return an error
In : x@y
Out:
matrix([[23., 20.],
        [41., 36.],
        [55., 48.]])

In : x@2 # return an error
In : x*2
```

Basic math for arrays and matrices

- Division is always element-by-element.
- Array exponentiation operates element-by-element. Exponentiation of matrices differs from array exponentiation, and can only be used on square matrices. When x is a square matrix and y is a positive integer, `x**y` produces `x*x*...*x` (y times).
- Matrix transpose is expressed using either `.T` or the `transpose()` function.

```
z=np.matrix([[1,2,4],[5,7,10]])
np.transpose(z) # using transpose() function
z.T
z.transpose()
```

Special Arrays

- Functions are available to construct a number of useful, frequently encountered arrays.

Table 6: Special arrays

`ones()`	generates an array of 1s and is generally called with one argument, a tuple, containing the size of each dimension.
`ones_like()`	creates an array with the same shape and data type as the input `ones_like(x)` is equivalent to calling `ones(x.shape,x.dtype)`.
`zeros()`	produces an array of 0s in the same way ones.
`zeros_like()`	creates an array with the same size and shape as the input. `zeros_like(x)` is equivalent to calling `zeros(x.shape,x.dtype)`.
`empty()`	produces an empty (uninitialized) array
`empty_like()`	creates an array with the same size and shape as the input. `empty_like(x)` is equivalent to calling `empty(x.shape,x.dtype)`.
`full()`	produces a full (input size) array of the input
`eye(), identity()`	generates an identity array.

```
M, N = 5, 5
x=np.ones((M,N)) # M by N array of 1s
x=np.ones((M,M,N)) # 3D array
x=np.ones((M,N), dtype='int32') # 32-bit integers

x=np.zeros((M,N)) # M by N array of 0s
x=np.zeros((M,M,N)) # 3D array of 0s
x=np.zeros((M,N),dtype='int64') # 64 bit integers

x=np.empty((M,N)) # M by N empty array
x=np.empty((N,N,N,N)) # 4D empty array
x=np.empty((M,N),dtype='float32') # 32-bit floats (single precision)
x=np.full((M,N), c) # dtype of c

In = np.eye(N) # N by N identity array
```

Array and Matrix Functions

- Some useful array functions are listed in the following table

Table 7: Array and matrix functions

`view()`	produce a representation of an array and matrix as another type without copying the data.
`asarray()`	work in a similar matter as `asmatrix`
`shape(x)`	returns the size of all dimensions or an array or matrix as a tuple.
`reshape()`	gives a new shape to an array without changing its data.
`size()`	returns the total number of elements in an array or matrix.
`ndim`	returns the size of all dimensions or an array or matrix as a tuple.
`tile()`	replicates an array according to a specified size vector.
`ravel()`	returns a flattened view (1-dimensional) of an array or matrix.
`flatten`	works like `ravel` except that it copies the array when producing the flattened version.
`flat()`	produces a `numpy.flatiter` object (flat iterator) which is an iterator over a flattened view of an array.
`vstack()/hstack()`	stacks compatible arrays and matrices vertically/horizontally.
`concatenate()`	allows concatenation along any given axis.
`split(), vsplit(), hsplit()`	`vsplit` and `hsplit` split arrays and matrices vertically and horizontally. `split`, which can split along an arbitrary axis.
`delete()`	returns a new array with sub-arrays along an axis deleted.
`squeeze()`	removes singleton dimensions from an array.
`fliplr(), flipud()`	flip arrays in a left-to-right and up-to-down directions, respectively.
`diag(x)`	returns the diagonal elements of a squared array as a column vector.
`triu(), tril()`	produce upper and lower triangular arrays, respectively.

Array and Matrix Functions

```
In : x=np.random.randn(4,3)
In : x.shape
Out: (4, 3)
In : np.shape(x)
Out: (4, 3)
In : M,N=np.shape(x) # set M=4 and N=3

In : x=np.array([[1,2],[3,4]])
In : y=np.reshape(x,(4,1)) # return 4 by 1 array
In : y
Out:
array([[1],
       [2],
       [3],
       [4]])

In : x=np.random.randn(4,3)
In : size(x)
Out: 12

In : np.ndim(x)
Out: 2
In : x.ndim
Out: 2

In : x=np.array([[1,2],[3,4]])
In : x.ravel()
Out: array([1, 2, 3, 4])
In : x.T.ravel()
Out: array([1, 3, 2, 4])

In : x=np.reshape(np.arange(6),(2,3))
In : y=x
In : np.hstack((x,y))
Out:
array([[0, 1, 2, 0, 1, 2],
       [3, 4, 5, 3, 4, 5]])
```

Linear Algebra Functions

Table 8: Linear algebra functions

`matrix_power()`	raises a square array or matrix to an integer power.
`svd()`	computes the singular value decomposition of a matrix.
`cond()`	computes the condition number of a matrix.
`slogdet()`	computes the sign and log of the absolute value of the determinant.
`solve()`	solves $X\beta = y$ for β when X is invertible.
`lstsq()`	returns the least squared solution of $y = X\beta$.
`cholesky(A)`	returns the Cholesky decomposition of A.
`det(A)`	returns the determinant of A.
`eig(A)`	computes the eigenvalues and eigenvectors of A.
`inv(A)`	computes the inverse of A.
`kron(x,y)`	computes $x \otimes y$.
`trace(A)`	computes the trace of A.
`matrix_rank(A)`	computes the rank of A using SVD.

Linear Algebra Functions

```
In : X=np.array([[1.0,2.0,3.0],[3.0,3.0,4.0],[1.0,1.0,4.0]])
In : y=np.array([[1.0],[2.0],[3.0]])
In : np.linalg.solve(X,y)
Out:
array([[ 0.625],
       [-1.125],
       [ 0.875]])

In : x=np.matrix([[1,.5],[.5,1]])
In : C=np.linalg.cholesky(x)
In : C*C.T - x
Out:
matrix([[ 0.00000000e+00,  0.00000000e+00],
        [ 0.00000000e+00, -1.11022302e-16]])

In : x=np.matrix([[1,.5],[.5,1]])
In : np.linalg.det(x)
Out: 0.75

In : x=np.array([[1,.5],[.5,1]])
In : xInv=np.linalg.inv(x)
In : np.dot(x,xInv)
Out:
array([[1., 0.],
       [0., 1.]])

In : x=np.asmatrix(x)
In : x**(-1)*x
Out:
matrix([[1., 0.],
        [0., 1.]])

In : x=np.array([[1,.5],[1,.5]])
In : np.linalg.matrix_rank(x)
Out: 1
```

Basic functions

- Some functions from numpy are given in the following tables.

Table 9: Basic functions, Part I

Arrays and Matrices	
`linspace(l,u,n)`	generates a set of n points uniformly spaced between l and u
`logspace(l,u,n)`	produces a set of logarithmically spaced points between 10^l and 10^u
`arange(l,u,s)`	produces a set of points spaced by s between l (inclusive) and u (exclusive).
`meshgrid()`	broadcasts two vectors to produce two 2-dimensional arrays.
`r_[start:end:step/count]`	generates 1-dimensional arrays from slice notation.
`c_`	identical to r_ except that column arrays are generated.
Rounding	
`around()`	rounds to the nearest integer
`floor()`	rounds to the next smallest integer
`ceil()`	rounds to the next largest integer
Math	
`sum()/cumsum()`	sums elements in an array/produces the cumulative sum of the values in the array
`prod()/cumprod()`	product and cumulative product are returned
`diff()`	calculate the n-th discrete difference along the given axis.
`exp()`	returns the element-by-element exponential (e^x) for an array.
`log()`	returns the element-by-element natural logarithm ($\ln(x)$) for an array.
`log10()`	returns the element-by-element base-10 logarithm ($\log 10(x)$) for an array.
`sqrt()`	returns the element-by-element square root ($\sqrt{x}$) for an array.
`square()`	returns the element-by-element square (x^2) for an array.
`absolute(), abs()`	returns the element-by-element absolute value for an array.
`sign()`	returns the element-by-element sign function.
`real()/imag()`	returns the real/complex elements of a complex array.
`conj(), conjugate()`	returns the element-by-element complex conjugate for a complex array

Basic functions

- Some functions from `numpy` are given in the following tables.

Table 10: Basic functions, Part II

Set	
`unique()`	returns the unique elements in an array.
`in1d()`	test whether each element of a 1-D array is also present in a second array.
`intersect1d()`	similar to in1d, except that it returns the elements rather than a Boolean array
`union1d`	returns the unique set of elements in 2 arrays.
`setdiff1d`	returns the set of the elements which are only in the first array but not in the second array
`setxor1d`	returns the set of elements which are in one (and only one) of two arrays.
Sorting	
`sort()`	sorts the elements of an array.
`ndarray.sort()`	a method for ndarrays which performs an in-place sort.
`argsort()`	returns the indices necessary to produce a sorted array.
`ndarray.max()/ndarray.min()`	methods for ndarrays for returning max/min.
`amax()/amin()`	returns max/min.
`argmax()/argmin()`	return the index or indices of the maximum/minimum element(s).
`maximum()/minimum()`	return the maximum/minimum of two arrays.
Nan	
`nansum()`	identical sum, except that NaNs are ignored.
`nanmax(), nanmin`	identical to their non-NaN counterparts, except that NaNs are ignored.
`nanargmax(), nanargmin()`	identical to their non-NaN counterparts, except that NaNs are ignored.

Basic math for arrays and matrices

```
In : x = np.linspace(0, 10, 11) # generates 11 points between 0 and 10
In : x
Out: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])

In : x = np.arange(4, 10, 1.25) # points between 4 and 10 (exclusive), step size 1.25
In : x
Out: array([4.  , 5.25, 6.5 , 7.75, 9.  ])

In : np.r_[0:10:1] # equivalent to arange(0:10:1) or arange(10)
Out: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In : np.around(x, 2) # 2 decimal places to round
Out: array([ 0.42, -0.28, -1.39])

In : x = np.random.randn(4,2)
In : x
Out:
array([[ 0.26049728, -0.15057116],
       [-1.12153343, -0.86933689],
       [ 2.11189621, -1.9812482 ],
       [-0.23327226, -1.27066752]])

In : np.sort(x) # sort across columns
Out:
array([[-0.15057116,  0.26049728],
       [-1.12153343, -0.86933689],
       [-1.9812482 ,  2.11189621],
       [-1.27066752, -0.23327226]])

In : np.sort(x, 0) # sort across rows
Out:
array([[-1.12153343, -1.9812482 ],
       [-0.23327226, -1.27066752],
       [ 0.26049728, -0.86933689],
       [ 2.11189621, -0.15057116]])
```

Logical operators

- Logical operators can be used on scalars, arrays or matrices. All comparisons are done element-by-element and return either `True` or `False`.

Table 11: Logical operators: Part I

`>, greater()`	greater than
`>=, greater_equal()`	greater than or equal to
`<, less()`	less than
`<=, less_equal()`	less than or equal to.
`==, equal()`	equal to
`!=, not_equal()`	not equal to.
`&, and, logical_and()`	True if both True
`or, logical_or()`	True if either or both True
`~, not, logical_not()`	True if not True
`^, logical_xor()`	True if one True and one False.
`all()`	returns True if all logical elements in an array are 1.
`any()`	returns True if any element of an array is True.
`allclose()`	compares two arrays for near equality.
`array_equal()`	tests if two arrays have the same shape and elements.

Logical operators

- Logical operators can be used on scalars, arrays or matrices. All comparisons are done element-by-element and return either `True` or `False`.

Table 12: Logical operators: Part II

`isnan()`	True if nan
`isinf()`	True if inf
`isfinite()`	True if not inf and not nan
`isposfin(), isnegfin()`	True for positive or negative inf
`isreal()`	True if real valued
`iscomplex()`	True if complex valued
`is_string_like()`	True if argument is a string
`is_numlike()`	True if argument is a numeric type
`isvector()`	True if argument is a vector

Logical operators

```
In : x = np.array([[1,2],[-3,-4]])
In : x > 0
Out:
array([[ True,  True],
       [False, False]])
In : x == -3
Out:
array([[False, False],
       [ True, False]])

In : x=np.arange(2.0,4)
In : y=x>= 0

In : y
Out: array([ True,  True])

In : z=x<2
In : z
Out: array([False, False])

In : np.logical_and(y,z)
Out: array([False, False])

In : y&z
Out: array([False, False])

In : ~(y&z)
Out: array([ True,  True])
In : np.logical_not(y&z)
Out: array([ True,  True])

In : import math
In : x=np.array([4,math.pi,math.inf,math.inf/math.inf])
In : x
Out: array([4.        , 3.14159265,        inf,        nan])
In : np.isnan(x)
Out: array([False, False, False,  True])
In : np.isinf(x)
Out: array([False, False,  True, False])
```

Control Structures

- Python uses white space changes to indicate the start and end of flow control blocks, and so indentation (4 spaces) matters.
  - `if...elif...else`
  - `for`
  - `while`
  - `try...except`
  - List, tuple, dictionary and set comprehension

if...elif...else

- The structure of `if...elif...else` is in the following form:

```
if logical_1:
    Code to run if logical_1
elif logical_2:
    Code to run if logical_2 and not logical_1
elif logical_3:
    Code to run if logical_3 and not logical_1 or logical_2
    ...
    ...
else:
    Code to run if all previous logicals are false
```

- Some examples

```
In : y = 5
In : if y<5:
...:        y += 1 # this is equivalent to y=y+1
...: else:
...:        y -= 1 # this is equivalent to y=y-1

In : y
Out: 4

In : x = 5
In : if x<5:
...:      x = x + 1
...: elif x>5:
...:      x = x - 1
...: else:
...:      x = x * 2

In : x
Out: 10
```

for loop

- The the generic structure of a for loop is

```
for item in iterable:
    Code to run
```

- item is an element from iterable, and iterable can be anything that is iterable in Python. The most common examples are range,list, tuple,array or matrix.
- Some examples

```
In : count = 0
...: for i in range(100):
...:     count += i # Equivalent to count=count+i
...:
...: count
Out: 4950
```

```
In : count = 0
...: x=np.linspace(0,500,50)
...: for i in x:
...:     count += i # Equivalent to count=count+i
...:
...: count
Out: 12499.999999999998
```

```
In : count = 0
...: x = list(np.arange(-20,21))
...: for i in x:
...:     count += i
...:
...: count
Out: 0
```

for loop

- Some examples

```
In : x = np.zeros((5,5))
...: for i in range(np.size(x,0)):
...:     for j in range(np.size(x,1)): # nested for loop
...:         if i<j:
...:             x[i,j]=i+j
...:         else:
...:             x[i,j]=i-j
...:
...: x
Out:
array([[0., 1., 2., 3., 4.],
       [1., 0., 3., 4., 5.],
       [2., 1., 0., 5., 6.],
       [3., 2., 1., 0., 7.],
       [4., 3., 2., 1., 0.]])
```

- Finally, `for` loops can be used with 2 items when the `iterable` is wrapped in `enumerate`.

```
In : x=np.linspace(0,5,3)
...: for i,y in enumerate(x):
...:     print('i is :', i)
...:     print('y is :', y)
...:
...:
i is : 0
y is : 0.0
i is : 1
y is : 2.5
i is : 2
y is : 5.0
```

while loop

- The the generic structure of a while loop is

```
while logical:
    Code to run
    Update logical
```

- Some examples

```
In : count = 0
...: i = 1
...: while i<10:
...:     count += i
...:     i += 1 # equivalent to i=i+1
...:
...: count
Out: 45
```

```
In : mu = np.abs(100*np.random.randn(1))
...: index = 1
...: while np.abs(mu) > .0001:
...:     mu = (mu+np.random.randn(1))/index
...:     index=index+1
...:
In : mu
Out: array([-7.44593523e-05])
In : index
Out: 162
```

- break can be used in a while loop to immediately terminate execution.

```
In : condition = True
...: i = 1
...: x = np.random.randn(1000000)
...: while condition:
...:     if x[i] > 1.0:
...:         break # No printing if x[i] > 3 or reached end of array
...:     print(x[i])
...:     i += 1
```

List comprehension

- A simple list can be used to convert a `for` loop which includes an append into a single line statement.

```
In : x=np.arange(3.0)
...: y=[]
...: for i in range(len(x)):
...:     y.append(np.exp(x[i]))
...:
   ...: y
Out: [1.0, 2.718281828459045, 7.38905609893065]
In : z = [np.exp(x[i]) for i in range(len(x))] #list comprehension saves 2 lines.
...: z
Out: [1.0, 2.718281828459045, 7.38905609893065]


In : x = np.arange(5.0)
...: y = []
...: for i in range(len(x)):
...:    if np.floor(i/2)==i/2:
...:        y.append(x[i]**2)
...:
...: y
Out: [0.0, 4.0, 16.0]
In : z=[x[i]**2 for i in range(len(x)) if np.floor(i/2)==i/2]
...: z
Out: [0.0, 4.0, 16.0]
```

List comprehension

- Set and dictionary comprehensions use {} while tuple comprehensions require an explicit call to `tuple`.

```
In : x = np.arange(-5.0,5.0)
...: z_set={x[i]**2.0 for i in range(len(x))}
...: z_set
Out: {0.0, 1.0, 4.0, 9.0, 16.0, 25.0}

In : z_dict={i:np.exp(i) for i in x}
...: z_dict
Out:
{-5.0: 0.006737946999085467,
 -4.0: 0.01831563888873418,
 -3.0: 0.049787068367863944,
 -2.0: 0.1353352832366127,
 -1.0: 0.36787944117144233,
 0.0: 1.0,
 1.0: 2.718281828459045,
 2.0: 7.38905609893065,
 3.0: 20.085536923187668,
 4.0: 54.598150033144236}

In : z_tuple=tuple(i**3 for i in x)
...: z_tuple
Out: (-125.0, -64.0, -27.0, -8.0, -1.0, 0.0, 1.0, 8.0, 27.0, 64.0)
```

Functions

- To declare a function:

```
def function_name(arguments):
    """docstring"""
    statement(s)
    return expression_list
```

- Functions are declared using the `def` keyword, and the value produced is returned using the `return` keyword.
- `docstring` is use to describe what function does. It can be accessed by calling `help(function_name)`.
- Consider the L_p distance between two vectors $x = (x_1, \ldots, x_n)'$ and $y = (y_1, \ldots, y_n)'$ defined by $d_p(x, y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{1/p}$.

```
def lp_norm(x,y,p):
    """
    This function calculates L_p distance between two vectors
    """
    d = (np.sum(np.abs(x-y)**p))**(1/p)
    return d
```

Functions

- The function can be called in the following way.

```
In : x = np.random.randn(10)
...: y = np.random.randn(10)
...: z1 = lp_norm(x,y,2)
...: z2 = lp_norm(p=2,x=x,y=y)
...: print("The Lp distances are ",z1, "and", z2)
The Lp distances are  3.88268940626486 and 3.88268940626486
```

- The multiple outputs can be returned in tuple form as in the following example.

```
In : def l1_l2_norm(x,y):
...:     d1 = np.sum(np.abs(x-y))
...:     d2= (np.sum(np.abs(x-y)**2))**(1/2)
...:     return (d1, d2)
In : x = np.random.randn(10)
In : y = np.random.randn(10)
In : # Using 1 output returns a tuple
In : z=l1_l2_norm(x,y)
In : print("The L1 distance is ",z[0])
The L1 distance is  5.83427433121689
In : print("The L2 distance is ",z[1])
The L2 distance is  2.137922607625371
```

Default and variable inputs

- Default values are set in the function declaration using the syntax `input=default`.

```
def lp_norm(x,y,p = 2):
    d=(np.sum(np.abs(x-y)**p))**(1/p)
    return d
# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
# Inputs with default values can be ignored
l2 = lp_norm(x,y)
l1 = lp_norm(x,y,1)
print("The l1 and l2 distances are ",l1,l2)
print("Is the default value overridden?", sum(abs(x-y))==l1)
The l1 and l2 distances are  10.201240563996286 3.911381835158071
Is the default value overridden? True
```

- Variable inputs can be handled using the `*args` (arguments) or `**kwargs` (keyword arguments) syntax. The `*args` syntax will generate tuple a containing all inputs past the required input list.

```
def lp_norm(x,y,p = 2, *args):
   d = (np.sum(np.abs(x-y)**p))**(1/p)
   print('The L' + str(p) + ' distance is :', d)
   out = [d]
   for p in args:
       d = (np.sum(np.abs(x-y)**p))**(1/p)
       print('The L' + str(p) + ' distance is :', d)
       out.append(d)
   return tuple(out)
```

Default and variable inputs

- Calling this new function yields

```
In : x = np.random.randn(10)
In : y = np.random.randn(10)

In : lp = lp_norm(x,y,2)
The L2 distance is : 3.4795676471815

In : lp = lp_norm(x,y,1,2,3,4,1.5,2.5,0.5)
The L1 distance is : 8.462614772098016
The L2 distance is : 3.4795676471815
The L3 distance is : 2.767201896918236
The L4 distance is : 2.5476765728380926
The L1.5 distance is : 4.579054294123239
The L2.5 distance is : 3.009213482305885
The L0.5 distance is : 65.6716939112997
```

Default and variable inputs

- The alternative syntax, `**kwargs`, generates a dictionary with all keyword inputs which are not in the function signature.

```
# Using kwargs
def lp_norm(x,y,p = 2, **kwargs):
    d = (np.sum(np.abs(x-y)**p))**(1/p)
    for key,value in kwargs.items():
        print('The value of', key, 'is', value)
    return d
# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
# Inputs with default values can be ignored
lp = lp_norm(x,y,kword1=1,kword2=3.2)
# The p keyword is in the function def, so not in **keywords
lp = lp_norm(x,y,kword1=1,kword2=3.2,p=0)
```

```
In : lp = lp_norm(x,y,kword1=1,kword2=3.2)
The value of kword1 is 1
The value of kword2 is 3.2

In : lp
Out: 4.123858213836834

In : lp = lp_norm(x,y,kword1=1,kword2=3.2,p=1)
The value of kword1 is 1
The value of kword2 is 3.2

In : lp
Out: 9.216016436616611
```

Anonymous function: `lambda`

- Python supports anonymous functions using the keyword `lambda`.
- The syntax for this function is `lambda arguments: expression`.
- Lambda functions can have any number of arguments but only one expression.

```
lp_norm=lambda x,y,p:(np.sum(np.abs(x-y)**p))**(1/p)

x = np.random.randn(10)
y = np.random.randn(10)
lp_norm(x,y,p=2)
Out: 5.712372067111222
```