Plotting

- For graphics, we need the following modules

```
import matplotlib.pyplot as plt
import seaborn as sns
```

- Commonly used high-level graphic functions are

High-level plot functions	
`plt.plot()`	Line plots
`plt.scatter()`	Scatter plots
`plt.bar()`	Bar charts
`plt.pie()`	Pie charts
`plt.hist()`	Histograms

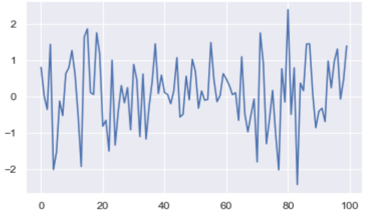
Line plots

- Using `plt.plot()` function, we can produce the following plots.

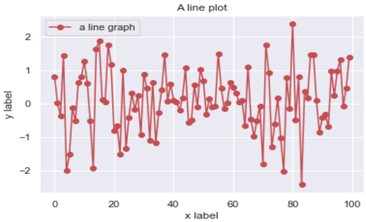
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style='darkgrid') # seaborn theme for the background
# Data
y = np.random.randn(100)
x = np.cumsum(np.random.rand(100))
# Plot 1
%matplotlib inline
plt.plot(y)
# Plot 2
plt.plot(y,'r-o',label='a line graph')
plt.legend()
plt.xlabel('x label')
plt.title('A line plot')
plt.ylabel('y label')
# Plot 3
plt.plot(x,y,'r-d',label='a line graph')
plt.legend()
plt.xlabel('x label')
plt.title('USING PLOT')
plt.ylabel('y label')
```

- In Plot 2, r-o indicates red (r), solid line (-) and circle (o) marker. Similarly, in Plot 3, r-d indicates color red (r), solid line (-) and diamond (d) marker.
- Titles are added with `title()` and legends are added with `legend()`. The legend requires that the line has a `label`.
- The labels for the x and y axis are added by `xlabel()` and `ylabel()`.

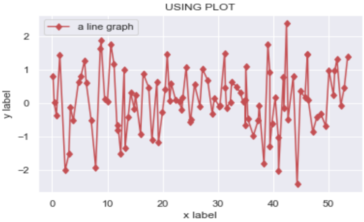
Line plots



(a) Plot 1



(b) Plot 2

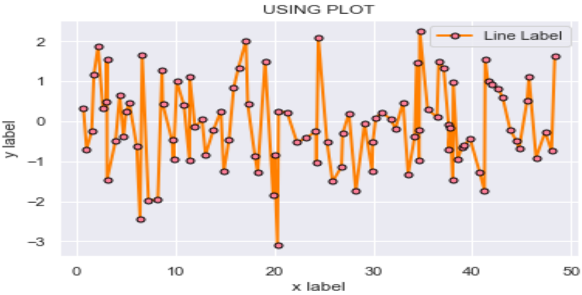


(c) Plot 3

Line plots

- Now we will specify the arguments of `plt.plot()` explicitly in the following example.

```
y = np.random.randn(100)
x = np.cumsum(rand(100))
plt.plot(x,y,alpha = 1, color = '#FF7F00', \
    label = 'Line Label', linestyle = '-', \
    linewidth = 2, marker = 'o', markeredgecolor = '#000000', \
    markeredgewidth = 1, markerfacecolor = '#FF7F99', \
    markersize=5)
plt.legend()
plt.xlabel('x label')
plt.title('USING PLOT')
plt.ylabel('y label')
```



Line plots

- The most useful keyword arguments of `plt.plot()` are listed in the table below.

Table 1: Keyword arguments for `plt.plot()`

`alpha`	Alpha (transparency) of the plot- default is 1 (no transparency)
`color`	Color description for the line
`label`	Label for the line- used when creating legends
`linestyle`	A line style symbol
`linewidth`	A positive integer indicating the width of the line
`marker`	A marker shape symbol or character
`markeredgecolor`	Color of the edge (a line) around the marker
`markeredgewidth`	Width of the edge (a line) around the marker
`markerfacecolor`	Face color of the marker
`markersize`	A positive integer indicating the size of the marker

- The functions `getp()` and `setp()` can be used to get the list of properties for a line (or any matplotlib object), and `setp()` can also be used to set a particular property.

Line plots

- Some options for `color`, `linestyle` and `marker` are given in the following table.

Table 2: Options for `color`, `linestyle` and `marker`

color	linestyle	marker
Blue: b	Solid: -	Point: .
Green: g	Dashed: –	Pixel: ,
Red: r	Dash-dot: -.	Circle: o
Cyan: c	Dotted: :	Square: s
Magenta: m		Diamond: D
Yellow: y		Thin diamond: d
Black: k		Cross: x
White: w		Plus: +
		Star: *
		Hexagon: H
		Alt. Hexagon: h
		Pentagon: p
		Triangles: ^,v,<,>
		Vertical line: |
		Horizontal line: _

Line plots

- The functions getp() and setp() can be used in the following way.

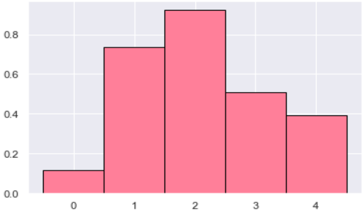
```
# Using setp()
h = plt.plot(np.random.randn(10))
plt.setp(h, alpha=0.5,linestyle='--',linewidth=2,label='Line Label',\
     marker='o',color='red')
plt.legend()
plt.xlabel('x label')
plt.title('USING PLOT')
plt.ylabel('y label')

# If you want to see all the properties that can be set,
# and their possible values, you can do:
getp(h)

#If you want to know the valid types of arguments, you can provide the name of
#the property you want to set without a value:
setp(h, 'alpha')
setp(h, 'color')
setp(h, 'linestyle')
setp(h, 'marker')
```

Scatter, bar, pie and histogram plots

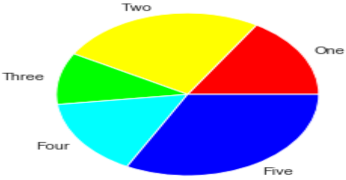
```
# Scatter plots
z =np.random.randn(100,2)
z[:,1] = 0.5*z[:,0] + np.sqrt(0.5)*z[:,1]
x=z[:,0]
y=z[:,1]
plt.scatter(x,y,  c = '#FF7F99', marker='o', \
    alpha = 1, label = 'Scatter Data')

# Bar plots
y = np.random.rand(5)
x = np.arange(5)
b=plt.bar(x,y, width = 1, color = '#FF7F99', \
    edgecolor = '#000000', linewidth = 1)

# Pie  plots
y = np.random.rand(5)
y = y/np.sum(y)
y[y<.05] = .05
labels=['One','Two','Three','Four','Five']
colors = ['#FF0000','#FFFF00','#00FF00','#00FFFF','#0000FF']
plt.pie(y,labels=labels,colors=colors)

#  Histograms
x = np.random.randn(1000)
plt.hist(x, bins = 30)
plt.hist(x, bins = 30, density=True, color='#FF7F00')
```

Scatter, bar, pie and histogram plots

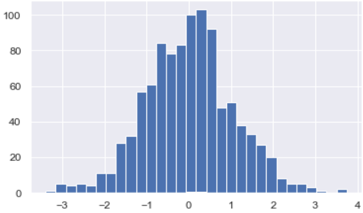
(d) Scatter plot



(e) Bar plot



(f) Pie plot



(g) Histogram

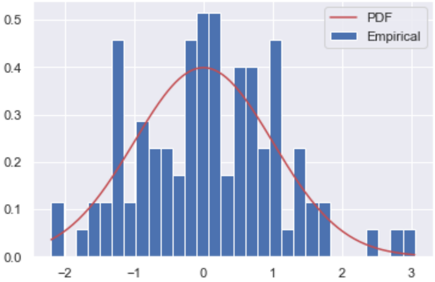
Multiple plots on the same figure

- For this, we need to first initialize the figure window by `figure()`, and then using `add_subplot()`.
- add_subplot(m,n,i), where `m` is the number of rows, `n` is the number of columns and `i` is the index of the subplot, is a method of `figure()`.

```
# Add the subplot to the figure
fig=plt.figure()
# Panel 1
ax = fig.add_subplot(2,2,1)
y = np.random.randn(100)
plt.plot(y)
ax.set_title('Plot 1')
# Panel  2
y = np.random.rand(5)
x = np.arange(5)
ax = fig.add_subplot(2,2,2)
plt.bar(x,y)
ax.set_title('Plot 2')
# Panel 3
y = np.random.rand(5)
y = y/np.sum(y)
y[y<.05] = .05
ax = fig.add_subplot(2,2,3)
plt.pie(y)
ax.set_title('Plot 3')
# Panel 4
z = np.random.randn(100,2)
z[:,1] = 0.5* z[:,0] + np.sqrt(0.5) * z[:,1]
x=z[:,1]; y=z[:,1]
ax = fig.add_subplot(2,2,4)
plt.scatter(x,y)
ax.set_title('Plot 4')
```

Multiple Plots on the Same Axes

```
# Multiple Plots on the Same Axes
import scipy as sp
x=np.random.randn(100)

plt.figure()
plt.hist(x, bins = 30,density=True,label = 'Empirical')
pdfx = np.linspace(x.min(),x.max(),200)
pdfy = sp.stats.norm.pdf(pdfx)
plt.plot(pdfx,pdfy,'r-',label = 'PDF')
plt.legend()
```



Importing and exporting data

- All of the data readers in `pandas` load data into a pandas `DataFrame`.
  - Comma-separated value (CSV) files can be read using `read_csv`.
  - Excel files, both 97/2003 (xls) and 2007/10/13 (xlsx), can be imported using `read_excel`.
  - Stata files can be read using `read_stata`.

`read_excel()`

- `read_excel()` supports reading data from both xls (Excel 2003) and xlsx (Excel 2007/10/13) formats.
- Notable keyword arguments include:
  - `header`, an integer indicating which row to use for the column labels. The default is 0 (top) row.
  - `skiprows`, typically an integer indicating the number of rows at the top of the sheet to skip before reading the file. The default is 0.
  - `skip_footer`, typically an integer indicating the number of rows at the bottom of the sheet to skip when reading the file. The default is 0.
  - `index_col`, an integer or column name indicating the column to use as the index. If not provided, a basic numeric index is generated.

read_csv()

- `read_csv()` reads comma separated value files.
- Notable keyword arguments include:
  - `delimiter`, the delimiter used to separate values. The default is `','`.
  - `delim_whitespace`, Boolean indicating that the delimiter is white space (space or tab). This is preferred to using a regular expression to detect white space.
  - `header`, an integer indicating the row number to use for the column names. The default is 0.
  - `skiprows`, similar to skiprows in `read_excel()`.
  - `skip_footer`, similar to `skip_footer` in `read_excel()`.
  - `index_col`, similar to `index_col` in `read_excel()`.
  - `names`, a list of column names to use in-place of any found in the file must use `header=0` (the default value).
  - `nrows`, an integer, indicates the maximum number of rows to read. This is useful for reading a subset of a file.
  - `usecols`, a list of integers or column names indicating which column to retain.

Import/Export Data

```
# Importing data
import pandas as pd

# Use read_excel() to import data
state_gdp = pd.read_excel('US_state_GDP.xls',sheet_name='Sheet1')
type(state_gdp)
state_gdp.head()

# Use read_csv() to import data
csv_data=pd.read_csv('US_state_GDP.csv')
type(csv_data)
csv_data.head()

# Use read_stata() to import data
stata_data=pd.read_stata('US_state.dta')
type(stata_data)
stata_data.head()

##
# Exporting data
state_gdp.to_excel('State_GDP_from_DataFrame.xls')
state_gdp.to_excel('State_GDP_from_DataFrame.xls', sheet_name='State GDP')
state_gdp.to_excel('State_GDP_from_DataFrame.xlsx')
state_gdp.to_csv('State_GDP_from_DataFrame.csv',index=False)
state_gdp.to_stata('State_GDP_from_DataFrame.dta',write_index=False)
```

Pandas

- The module `pandas` is a high-performance package that provides a comprehensive set of structures for working with data.
- `pandas` provides a set of data structures which include `Series` and `DataFrames`.
- `Series` are the equivalent of 1-dimensional arrays. `DataFrames` are collections of Series and so are 2-dimensional.
- `Series` are the primary building block of the data structures in pandas, and in many ways a `Series` behaves similarly to a `NumPy` `array`.
- A `Series` is initialized using a `list`, `tuple`, `array` or using a `dictionary`.

Series

```
# Series
In : s=pd.Series([0.1, 1.2, 2.3, 3.4, 4.5])
In : s
Out:
0     0.1
1     1.2
2     2.3
3     3.4
4     4.5
dtype: float64

In : type(s)
Out: pandas.core.series.Series

In : # From array
In : a=pd.array([0.1, 1.2, 2.3, 3.4, 4.5])
In : s=pd.Series(a)
In : s
Out:
0     0.1
1     1.2
2     2.3
3     3.4
4     4.5
dtype: float64

#From tuple
In : a=(1,2,3,4,'abs',NaN)
In : s=pd.Series(a)
In : s
Out:
0       1
1       2
2       3
3       4
4     abs
5     NaN
dtype: object
```

Series

- Series, like arrays, are sliceable. However, unlike a 1-dimensional array, a Series has an additional column-an index- which is a set of values which are associated with the rows of the Series.

```
In : s = pd.Series([0.1, 1.2, 2.3, 3.4, 4.5], index = ['a','b','c','d','e'])

In : s['a']
Out: 0.1

In : s[0]
Out: 0.1

In : s[['a','c']]
Out:
a     0.1
c     2.3
dtype: float64

In : s[[0,2]]
Out:
a     0.1
c     2.3
dtype: float64

In : s[:2]
Out:
a     0.1
b     1.2
dtype: float64

In : s[s>2]
Out:
c     2.3
d     3.4
e     4.5
dtype: float64
```

Series

- Series can also be initialized directly from `dictionaries`.

```
In : s=pd.Series({'a': 0.1, 'b': 1.2, 'c': 2.3})
In : s
Out:
a     0.1
b     1.2
c     2.3
dtype: float64
```

- Series are subject to math operations element-wise.

```
In : s * 2.0
Out:
a     0.2
b     2.4
c     4.6
dtype: float64
```

```
In : s - 1.0
Out:
a    -0.9
b     0.2
c     1.3
dtype: float64
```

QUEENS COLLEGE

Series

- In mathematical operations, indices that do not match are given the value NaN (not a number).

```
In : s1 = pd.Series({'a': 0.1, 'b': 1.2, 'c': 2.3})

In : s2 = pd.Series({'a': 1.0, 'b': 2.0, 'c': 3.0})

In : s3 = pd.Series({'c': 0.1, 'd': 1.2, 'e': 2.3})

In : s1 + s2
Out:
a     1.1
b     3.2
c     5.3
dtype: float64

In : s1 * s2
Out:
a     0.1
b     2.4
c     6.9
dtype: float64

In : s1 + s3
Out:
a     NaN
b     NaN
c     2.4
d     NaN
e     NaN
dtype: float64
```

Series

- The notable methods of series are listed in the table below.

Table 3: Some methods for `Series`

`values/index`	returns series as an array/returns index
`head()/tail()`	shows the first/last 5 rows of a series
`isnull()/notnull()`	returns a boolean same-sized object indicating if the values are NA/not NA
`loc[ ]/iloc[ ]`	`iloc[ ]` allows access using position `loc[ ]` allows access using index value or logical arrays.
`describe()`	returns a simple set of summary statistics.
`unique() and nunique()`	`unique()` returns unique values of Series object. `nunique()` returns number of unique elements in the object.
`drop and dropna`	`drop` returns series with specified index labels removed. `dropna` return a new series with missing values removed.
`fillna()`	fills all null values in a series with a specific value.
`append()`	appends one series to another.
`replace()`	`replace(list,values)` replaces a set of values in a series with a new value.
`update()`	`update(series)` replaces values in a series with those in another series, matching on the index.

Series

```
In : s1 = pd.Series([1.0,2,3])
In : s1.values # access to values
Out: array([1., 2., 3.])
In : s1.index
Out: RangeIndex(start=0, stop=3, step=1)
In : s1.index = ['cat','dog','elephant'] # set index labels
In : s1.index
Out: Index(['cat', 'dog', 'elephant'], dtype='object')
```

```
# Try the followings
s1=pd.Series([1,3,5,6,NaN,'cat','abc',10,12,5])
s1.index=['a','b','c','d','e','f','g','h','i','k']
s1.head()
s1.tail()
s1.isnull()
s1.notnull()
s1.loc['e']
s1.iloc[4]
s1.drop('e')
s1.dropna()
s1.fillna(-99)
```

```
s1 = pd.Series(arange(10.0,20.0))
s1.describe()
summ = s1.describe()
summ
summ['mean']
```

```
s1=pd.Series([1, 2, 3])
s2=pd.Series([4, 5, 6])
s1.append(s2)
s1.append(s2, ignore_index=True)
```

```
s1=pd.Series([1, 2, 3])
s2=pd.Series([4, 5, 6])
s1.replace(1,-99)
s1.update(s2)
s1
```

DataFrame

- DataFrames collect multiple series in the same way that a spreadsheet collects multiple columns of data.

```
In : import numpy as np
In : import pandas as pd
In : df = pd.DataFrame(np.array([[1,2],[3,4]]), columns=['dogs','cats'], \
   ...:                     index=['Alice','Bob'])
In : df
Out:
       dogs  cats
Alice     1     2
Bob       3     4

In : s1 = pd.Series(arange(0,5.0))
In : s2 = pd.Series(arange(1.0,6.0))
In : pd.DataFrame({'one': s1, 'two': s2})
Out:
   one  two
0  0.0  1.0
1  1.0  2.0
2  2.0  3.0
3  3.0  4.0
4  4.0  5.0

In : s3 = pd.Series(np.arange(0,3.0))
In : pd.DataFrame({'one': s1, 'two': s2, 'three': s3})
Out:
   one  two  three
0  0.0  1.0    0.0
1  1.0  2.0    1.0
2  2.0  3.0    2.0
3  3.0  4.0    NaN
4  4.0  5.0    NaN
```

Manipulating DataFrame: Column selection

- The use of DataFrame will be demonstrated using a data set containing a mix of data types using state level GDP data from the US.
- The data is loaded directly into a DataFrame using read_excel.

```
In : state_gdp = pd.read_excel('US_state_GDP.xls','Sheet1')
In : state_gdp.head() # print first 5 rows
Out:
  state_code        state  gdp_2009  ...  gdp_growth_2011  gdp_growth_2012  region
0         AK       Alaska     44215  ...              1.7              1.1      FW
1         AL      Alabama    149843  ...              1.0              1.2      SE
2         AR     Arkansas     89776  ...              0.7              1.3      SE
3         AZ      Arizona    221405  ...              1.7              2.6      SW
4         CA   California   1667152  ...              1.2              3.5      FW

[5 rows x 11 columns]
```

- Columns can be selected using a list of column names as in `state_gdp[['state_code', 'state']]`.

```
In : state_gdp.columns # print all column names
Out:
Index(['state_code', 'state', 'gdp_2009', 'gdp_2010', 'gdp_2011', 'gdp_2012',
       'gdp_growth_2009', 'gdp_growth_2010', 'gdp_growth_2011',
       'gdp_growth_2012', 'region'],
      dtype='object')

state_gdp[['state_code','state']].head() # select state_code and state columns
state_gdp.state_code.head() # select state_code column
state_gdp.region.head() # first five observation in region column
```

Manipulating DataFrame: Row slicing and column selection

- Rows can be selected using standard numerical slices.

```
In : state_gdp[1:3]
Out:
  state_code      state  gdp_2009  ...  gdp_growth_2011  gdp_growth_2012  region
1         AL    Alabama    149843  ...              1.0              1.2      SE
2         AR   Arkansas     89776  ...              0.7              1.3      SE

[2 rows x 11 columns]
```

```
state_gdp.region[0:5] # first five observation in region column
state_gdp['region'][0:5] # first five observation in region column
```

```
state_gdp[['state','gdp_2009']][0:5] # the first five rows of state and gdp_2009
```

- Finally, rows can also be selected using logical selection using a Boolean array with the same number of elements as the number of rows as the DataFrame.

```
state_long_recession = state_gdp['gdp_growth_2010']<0
state_gdp[state_long_recession] # returns states for which gdp_growth_2010 is negative
```

- It is not possible to use standard slicing to select both rows and columns. But we can use `loc[rowselector, coloumnselector]`.

```
state_gdp.loc[10:15,'state']
state_gdp.loc[state_long_recession,'state']
state_gdp.loc[state_long_recession,['state','gdp_growth_2010']]
state_gdp.loc[state_long_recession,['state','gdp_growth_2009','gdp_growth_2010']]
```

Manipulating DataFrame: Adding columns

- Adding columns to dataframes can be done in the following ways.

```
# Adding columns
# Create a new dataframe: state_gdp_2012
state_gdp_2012=state_gdp[['state','gdp_2012']].copy()
state_gdp_2012.head()
# Add column "gdp_growth_2012" to "state_gdp_2012".
state_gdp_2012['gdp_growth_2012'] = state_gdp['gdp_growth_2012']
state_gdp_2012.head()
```

- `insert(location,column_name,series)` inserts a Series at a specific location:
  - location uses 0-based indexing (i.e. 0 places the column first, 1 places it second, etc.),
  - `column_name` is the name of the column to be added
  - series is the series data.

```
state_gdp_2012 = state_gdp[['state','gdp_2012']]
state_gdp_2012.insert(1,'gdp_growth_2012',state_gdp['gdp_growth_2012'])
state_gdp_2012.head()
```

Manipulating DataFrame: Deleting columns

- Columns can be deleted by (i) the `del` keyword, (ii) `pop(column)` and (iii) `drop(list of columns,axis=1)`.
  - `del` will simply delete the Series from the DataFrame,
  - `pop(column)` will both delete the Series and return the Series as an output,
  - `drop()` will return a DataFrame with the Series dropped without modify the original DataFrame.

```
# Deleting a column
state_gdp_copy = state_gdp.copy()
state_gdp_copy.index = state_gdp['state_code'] # replace index with state_code
# Keep only 'gdp_2009','gdp_growth_2011' and 'gdp_growth_2012'
state_gdp_copy = state_gdp_copy[['gdp_2009','gdp_growth_2011','gdp_growth_2012']]
state_gdp_copy.head()
# Drop 'gdp_2009'
state_gdp_copy=state_gdp_copy.drop('gdp_2009',axis=1)
state_gdp_copy.head()
# Delete 'gdp_growth_2012'
gdp_growth_2012 = state_gdp_copy.pop('gdp_growth_2012')
gdp_growth_2012.head()
state_gdp_copy.head()
# Delete 'gdp_growth_2011'
del state_gdp_copy['gdp_growth_2011']
state_gdp_copy.head()
```

Functions and methods for DataFrame

- Some useful functions and methods are listed in the table below.

Table 4: Functions and methods for DataFrame

drop()/drapna()	drops specified labels from rows or columns.
	dropna() remove missing values (NaN values).
drop_duplicates	removes rows which are duplicates or other rows
values/index	values retrieves a the NumPy array.
	index returns the index of the DataFrame.
fillna	fills NA/NaN or other null values with other values.
T/transpose	both swap rows and columns of a DataFrame.
sort_values()/sort_index()	sort_values() sorts by the values along either axis.
	sort_index() will sort a DataFrame by the values in the index.
count()	counts non-NA cells for each column or row.
describe()	generates descriptive statistics.
value_counts()	returns a series containing counts of unique values..

Functions and methods for DataFrame

```
# Using insert()
state_gdp_2012 = state_gdp[['state','gdp_2012']] #create a new DataFrame:state_gdp_2012
state_gdp_2012.insert(1,'gdp_growth_2012',state_gdp['gdp_growth_2012'])
state_gdp_2012.head()

# Using drop(), dropna() and drop_duplicates()
df=pd.DataFrame(array([[1,nan,3,8],[nan,2,3,5],[10,2,3,nan],
                      [10,2,3,nan],[10,2,3,11]]))

df.columns = ['one','two','three','four'] # assign names to columns
df.index=['a','b','c','d','e'] # assign labels to index

df.drop('a',axis=0) # removes row 'a'
df.drop(['a','c'],axis=0) # removes row 'a' and 'c'
df.drop_duplicates() # removes row 'd'
df.drop('one',axis=1) # removes column 'one'

# Using values and index
df.values # returns values as an array
df.index # returns the index of the dataframe

#Using fillna()
df.fillna(0) # Replace all NaN elements with 0s.
replacements={'one':-99, 'two':-999}
df.fillna(value=replacements) # replace NaN values in column one and two

# Using T and transpose
df.T
np.transpose(df)
```

Statistical Distributions

- NumPy and SciPy contain important functions for simulation, probability distributions and statistics.
- NumPy random number generators are all stored in the module `numpy.random`.

Table 5: Statistical functions of `numpy.random`

`rand()/random_sample()`	generates uniform random numbers from $[0, 1)$.
`randn()/standard_normal`	generates random numbers from standard normal distribution.
`randint()/random_integers`	generates random integer from `[low,high)`.
`shuffle()`	randomly reorders the elements of an array in place.
`permutation()`	returns randomly reordered elements of an array.
`binomial()`	draw samples from a binomial distribution.
`chisquare()`	generates draws from chi-squared distribution.
	`sort_index()` will sort a DataFrame by the values in the index.
`exponential()`	generates a draw from the Exponential distribution.
`f(v_1,v_2)`	generates draws from F_{v_1,v_2} distribution.
`gamma()`	generates from gamma distribution.
`laplace()`	generates draws from the Laplace (Double Exponential) distribution.
`lognormal()`	generates draws from Log-Normal distribution.
`multinomial()`	generates draws from multinomial distribution.
`multivariate_normal()`	generates from multivariate Normal distribution.
`normal()`	generates from Normal distribution.
`poisson()`	generates from poisson distribution.
`standard_t()`	generates a draw from a Student's t distribution.
`uniform()`	generates a uniformrandom variable on (0,1).

Statistical functions from numpy.random

```
import numpy as np
x=np.random.rand(3,4,5)
y=np.random.random_sample((3,4,5))
x=np.random.randn(3,4,5)

y=np.random.standard_normal((3,4,5))
x=np.random.randint(0,10,(100))

x=np.arange(10)
np.random.shuffle(x)

x=np.arange(10)
np.random.permutation(x)

mu,sigma = 2, 1.5 # mean and standard deviation
s=np.random.normal(mu, sigma, 10)

n,p = 10,0.5  # number of trials, probability of each trial
s=np.random.binomial(n, p, 20)

nu,n=2,4 # degrees of freedom and sampel size
np.random.chisquare(nu,n)

v1,v2,n=2,30,3 # degrees of freedoms and sample size
np.random.f(v1,v2,n)

mean = [0, 0]
cov = [[10, 0], [0, 50]]  # diagonal covariance
import matplotlib.pyplot as plt
x, y = np.random.multivariate_normal(mean, cov, 1000).T
plt.plot(x, y, 'o')
plt.axis('equal')
plt.xlabel('x')
plt.ylabel('y')

np.random.standard_t(df=10, size=5)
```

Statistical Distributions

- Computer simulated random numbers are not actually random, they are generally described to as pseudo-random numbers.
- All pseudo-random numbers in `NumPy` use one core random number generator based on the Mersenne Twister.
- `numpy.random.seed` is a useful function for initializing the random number generator. To generate the same random numbers, we need to set seed.

```
In : import numpy as np

In : np.random.seed(0)
In : np.random.randn()
Out: 1.764052345967664

In : np.random.seed(0)

In : np.random.randn()
Out: 1.764052345967664
```

Statistical Distributions

- SciPy provides an extended range of random number generators, probability distributions and statistical tests.
- SciPy statistical functions are stored in the module `scipy.stats`. We import this module in the following way

```
import scipy.stats as stats
```

- Important distribution functions in `scipy.stats` are listed in the following table.

Table 6: Important distribution functions in `scipy.stats`

`norm`	normal distribution.
`beta`	beta distribution.
`cauchy`	cauchy distribution.
`chi2`	chi-squared distribution.
`expon`	exponential distribution.
`exponpow`	exponential power distribution
`f`	F distribution.
`Gamma`	gamma distribution.
`laplace`	laplace, double exponential distribution.
`lognorm`	lognormal distribution.
`t`	student's t distribution.

Statistical Distributions

- Important methods for distribution functions in Table 8 are listed in the following table.

Table 7: Methods for distribution functions in `scipy.stats`

`rvs()`	generates pseudo-random numbers.
`pdf()`	returns probability density function.
`logpdf()`	returns log probability density function.
`cdf()`	returns cumulative distribution function.
`ppf()`	inverse CDF evaluation for an array of values between 0 and 1.
`fit()`	estimates shape, location, and scale parameters from data by maximum likelihood using an array of data.
`median()/mean()`	returns median/mean of the distribution.
`var()/std()`	returns variance/standard deviation of the distribution.
`moment()`	returns nth non-central moment of the distribution.

- The documentation on these methods is given at `https://docs.scipy.org/doc/scipy/reference/stats.html`.

Statistical functions from scipy.stats

```
In : import scipy as sp
In : sp.stats.norm.rvs(loc=2,scale=3,size=10) # generates 10 rvs from N(2,9)
Out:
array([ 0.31613221, -1.05744118,  2.28474865,  5.43251686, -1.97227871,
        2.06680403,  2.18448145,  5.38146375,  2.71106676,  1.60296263])

In : sp.stats.norm.pdf(1.96, loc=0, scale=1) # evaluate normal pdf at 1.96
Out: 0.058440944333451476
In : sp.stats.norm.cdf(-1.96, loc=0, scale=1)#evaluate normal cdf at -1.96
Out: 0.024997895148220435
In : sp.stats.norm.ppf(0.95,loc=0,scale=1)#return quantile at the lower tail prob 0.95
Out: 1.6448536269514722

In : x=sp.stats.norm.rvs(loc=1,scale=5,size=1000)
In : location,scale=sp.stats.norm.fit(x)
In : location,scale=sp.stats.norm.fit(x,input=(1,3))#The search starts at input=(1,3)
In : print('(location, scale)=',(location,scale))
(location, scale)= (1.2632581286252547, 4.8100742790320625)

In : sp.stats.norm.median(loc=3,scale=1) # returns median of N(3,1)
Out: 3.0
In : sp.stats.norm.mean(loc=3,scale=2) # returns mean of N(3,4)
Out: 3.0
In : sp.stats.norm.var(loc=3,scale=2) # returns variance of N(3,4)
Out: 4.0
In : sp.stats.norm.std(loc=3,scale=2) # return std of N(3,4)
Out: 2.0
In : sp.stats.norm.moment(2,loc=0,scale=1) # the second non-central moment of N(0,1)
Out: 1.0
```

Regression analysis

- The statsmodels module provides a large range of cross-sectional models as well as sometime-series models.
- The documentation is available at http://www.statsmodels.org/stable/index.html.
- We will use the statsmodels.api and statsmodels.formula.api module to run regressions.

```
import numpy as np
import pandas as pd
import statsmodels.api as sm
import statsmodels.stats.api as sms
import statsmodels.formula.api as smf
from statsmodels.iolib.summary2 import summary_col
```

- There are two options for running OLS regressions: (i) smf.ols() and (ii) sm.OLS(). In the first option, statsmodels allows users to fit statistical models using R-style formulas.

Wage Regression

- The data set is wage1000.csv with headers.

```
# Importing data
# Use read_csv() to import data
wage_data = pd.read_csv('wage1000.csv')
type(wage_data)
wage_data.head()
```

```
In : wage_data.columns
Out:
Index(['wage', 'female', 'nonwhite', 'unionmember', 'education', 'experience', 'age'],
      dtype='object')
```

- It is wage data with 1000 observations from the US Bureau of Census Current Population survey, March 1995.
- The underlying population is the employed labor force, age 18-65. The variables are as follows:
  1. hourly wage
  2. female (1= worker = female)
  3. non-white (1= worker = non-white)
  4. unionmember (1 = worker = unionized)
  5. education (years of education)
  6. experience (years of work experience)
  7. age

Wage Regression: using `smf.ols()`

- The `smf` module hosts many of the same functions found in the `sm` module (e.g. OLS, GLM). Use `dir(smf)` to list available models.

```
model1 = smf.ols(formula='wage~female+nonwhite+unionmember+education+\
                 experience',data=wage_data)
# We need to use .fit() to obtain parameter estimates
result1 = model1.fit()
# We now have the fitted regression model stored in result1
# To view the OLS regression results, we can call the .summary() method
result1.summary()
```

- Note that we do not need to specify an intercept term `smf.ols()`. The function will include an intercept term by default.

Wage Regression: using `smf.ols()`

- The `result1.summary()` function prints the following output.

```
In : print(result1.summary())
                            OLS Regression Results
==============================================================================
Dep. Variable:                   wage   R-squared:                       0.348
Model:                            OLS   Adj. R-squared:                  0.345
Method:                 Least Squares   F-statistic:                     106.0
Date:                Wed, 19 Jun 2019   Prob (F-statistic):           9.45e-90
Time:                        15:34:18   Log-Likelihood:                -3314.2
No. Observations:                1000   AIC:                             6640.
Df Residuals:                     994   BIC:                             6670.
Df Model:                           5
Covariance Type:            nonrobust
================================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
Intercept     -8.5786      1.161     -7.388      0.000     -10.857      -6.300
female        -3.0985      0.424     -7.313      0.000      -3.930      -2.267
nonwhite      -1.6072      0.603     -2.664      0.008      -2.791      -0.423
unionmember    0.8212      0.583      1.408      0.159      -0.323       1.966
education      1.4983      0.075     19.948      0.000       1.351       1.646
experience     0.1697      0.018      9.197      0.000       0.133       0.206
==============================================================================
Omnibus:                      370.409   Durbin-Watson:                   1.899
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             2099.721
Skew:                           1.598   Prob(JB):                         0.00
Kurtosis:                       9.339   Cond. No.                         141.
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specifi
```

Wage Regression: using `smf.ols()`

- We can use `print(dir(result1))` and `print(dir(result1.model))` to see available attributes.

```
In [178]: print(dir(result1))
['HC0_se', 'HC1_se', 'HC2_se', 'HC3_se', '_HCCM', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', '_cache', '_data_attr', '_get_robustcov_results', '_is_nested', '_wexog_singular_values',
'aic', 'bic', 'bse', 'centered_tss', 'compare_f_test', 'compare_lm_test', 'compare_lr_test', 'condition_number', 'conf_int',
'conf_int_el', 'cov_HC0', 'cov_HC1', 'cov_HC2', 'cov_HC3', 'cov_kwds', 'cov_params', 'cov_type', 'df_model', 'df_resid',
'diagn', 'eigenvals', 'el_test', 'ess', 'f_pvalue', 'f_test', 'fittedvalues', 'fvalue', 'get_influence', 'get_prediction',
'get_robustcov_results', 'het_scale', 'initialize', 'k_constant', 'llf', 'load', 'model', 'mse_model', 'mse_resid', 'mse_total',
'nobs', 'normalized_cov_params', 'outlier_test', 'params', 'predict', 'pvalues', 'remove_data', 'resid', 'resid_pearson',
'rsquared', 'rsquared_adj', 'save', 'scale', 'ssr', 'summary', 'summary2', 't_test', 't_test_pairwise', 'tvalues',
'uncentered_tss', 'use_t', 'wald_test', 'wald_test_terms', 'wresid']

In [179]: print(dir(result1.model))
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_data_attr',
'_df_model', '_df_resid', '_fit_ridge', '_get_init_kwds', '_handle_data', '_init_keys', '_setup_score_hess', 'data', 'df_model',
'df_resid', 'endog', 'endog_names', 'exog', 'exog_names', 'fit', 'fit_regularized', 'formula', 'from_formula',
'get_distribution', 'hessian', 'hessian_factor', 'information', 'initialize', 'k_constant', 'loglike', 'nobs',
'normalized_cov_params', 'pinv_wexog', 'predict', 'rank', 'score', 'weights', 'wendog', 'wexog', 'wexog_singular_values',
'whiten']
```

Wage Regression: attributes

- Some important attributes are listed in the following table.

Table 8: Some important attributes

nobs	returns the number of observations used in the estimation.
params	returns the estimated parameters in list.
resid	returns the residuals in list.
predict	returns predicted values in array.
model.exog	returns exogenous variables in array.
model.exog_names	returns the names of exogenous variables in a list.
model.endog/model.endog_names	returns the endogenous variable values/name.
model.loglike	returns the likelihood function evaluated at params.
rsquared/rsquared_adj	returns unadjusted/adjusted R^2.

- Try the following:

```
# Some attributes
result1.nobs
result1.params
result1.resid
result1.model.endog_names
result1.model.exog_names
result1.rsquared
```

Wage Regression: using `smf.ols()`

- Run some alternative models.

```
## Run some alternative models
# Add squared-experience as an exogenous variable
model2=smf.ols(formula='wage~female+nonwhite+unionmember+education+\
                experience+I(experience**2)',data=wage_data)

result2=model2.fit()
result2.summary()

# Normality of the residuals
JB,JBpv,Skew,Kurtosis= sms.jarque_bera(result2.resid)
print('Test statistic is',JB, 'with a p-value of',JBpv)

# Heteroskedasticity tests
In : test = sms.het_breuschpagan(result2.resid, result2.model.exog)
     ...: print('LM statistic is',np.round(test[0],3), 'with a p-value of',\
     ...:       np.round(test[1],3))
LM statistic is 46.249 with a p-value of 0.0

#Using the option cov_type='HC0': White's heteroskedasticity consistent
#covariance estimator.
result3=model2.fit(cov_type='HC0')
print(result3.summary())
# Compare standard errors
In : sde = pd.concat([result2.bse, result3.bse],1)
In : sde.columns = ['No option', 'HC0']
In : print(sde)
                    No option       HC0
Intercept            1.184103  1.286034
female               0.419360  0.418680
nonwhite             0.596781  0.488253
unionmember          0.577060  0.490091
education            0.075091  0.095138
experience           0.061721  0.060487
I(experience ** 2)   0.001389  0.001464
```

Wage Regression: using `sm.OLS()`

- Next, we describe how to use `sm.OLS()` for regression analysis

```
## An alternative approach based on sm.OLS
# Define endogenous and exogenous variables
wage_data['const']=1 # add a constant column to wage_data
y=wage_data['wage']
X=wage_data[['const','female', 'nonwhite', 'unionmember', 'education',\
               'experience']]
model1=sm.OLS(endog=y,exog=X)
type(model1)
# We need to use .fit() to obtain parameter estimates
result1=model1.fit()
type(result1)
# We now have the fitted regression model stored in result1
# To view the OLS regression results, we can call the .summary() method
result1.summary()

# Add experience**2 to X and form a new model
X['I(experience**2)']=wage_data['experience']**2
model2=sm.OLS(endog=y,exog=X)
result2=model2.fit()
result2.summary()

# Use cov_type='HC0'
result3=model2.fit(cov_type='HC0')
result3.summary()
```

- Note that we need to explicitly specify the intercept term: `wage_data['const']=1` adds a column of ones to the dataframe `wage_data`.

Using summary_col() for reporting regression results

- Next, we show how to use summary_col() to report regression results.

```
# Print results in table form
models=['Model 1','Model 2','Robust Model']
info_dict={'R-squared': lambda x: "{:.2f}".format(x.rsquared),\
           'No. observations':lambda x: "{0:d}".format(int(x.nobs))}
regressors=['const','female','nonwhite','unionmember','education','experience',\
            'I(experience**2)']
results_table=summary_col(results=[result1,result2,result3],\
                          float_format='%0.3f', stars=True,\
                          model_names=models,info_dict=info_dict,\
                         regressor_order=regressors)
results_table.add_title('Table 1: Wage Regressions') # add a title
```

Using summary_col() for reporting regression results

- The resulting table is in the following form.

```
In : results_table
Out:
<class 'statsmodels.iolib.summary2.Summary'>
"""
              Table 1: Wage Regressions
=================================================
                  Model 1   Model 2   Robust Model
-------------------------------------------------
const             -8.579*** -9.960*** -9.960***
                  (1.161)   (1.184)   (1.286)
female            -3.099*** -3.026*** -3.026***
                  (0.424)   (0.419)   (0.419)
nonwhite          -1.607*** -1.553*** -1.553***
                  (0.603)   (0.597)   (0.488)
unionmember       0.821     0.741     0.741
                  (0.583)   (0.577)   (0.490)
education         1.498***  1.446***  1.446***
                  (0.075)   (0.075)   (0.095)
experience        0.170***  0.452***  0.452***
                  (0.018)   (0.062)   (0.060)
I(experience**2)            -0.007*** -0.007***
                            (0.001)   (0.001)
R-squared         0.35      0.36       0.36
No. observations  1000      1000       1000
=================================================
Standard errors in parentheses.
* p<.1, ** p<.05, ***p<.01
"""
```