```python
import numpy as np

# basic  concept
#One of the key features of NumPy is its N-dimensional array object, or ndarray,
# which is a fast, flexible container for large data sets in Python.

arr = np.array([1, 2, 3, 4, 5])

print(arr.dtype)
print(arr)

float_arr = arr.astype(np.float64)

print(float_arr.dtype)
print(float_arr)

# In addition to np.array, there are a number of other functions for creating new arrays.
# As examples, zeros and ones create arrays of 0’s or 1’s, respectively, with a given length or shape.
# empty creates an array without initializing its values to any particular value.

arr = np.ones(10)
print(arr)

arr = np.zeros(10)
print(arr)

arr = np.zeros((3,6))
print(arr)

arr = np.empty((2,4,2))
print(arr)


int_array = np.arange(10)
print(int_array)

# array vectorization
# Arrays are important because they enable you to express batch operations on data without writing any for loops.
# This is usually called vectorization.
# Any arithmetic operations between equal-size arrays applies the operation elementwise:

arr = np.array([[1, 2, 3], [4, 5, 6]])

print(arr)
print(arr * arr)
```

```
print(arr - arr)

print(1 / arr)
print("arr*0.5=", arr * 0.5)

# it is different with broadcasting which applied to the operations between different arrays.
# Arithmetic operations with scalars are as you would expect, propagating the value to each element

print(1 / arr)
print("arr*0.5=", arr * 0.5)

# Numpy Array operation

# NumPy array indexing is a rich topic, as there are many ways you may want to select a subset of your data
# or individual elements.
# One-dimensional arrays are simple; on the surface they act similarly to Python lists:

arr = np.arange(10)
print(arr[5])
print(arr[5:8])

# As you can see, if you assign a scalar value to a slice, as in arr[5:8] = 12,
# the value is propagated (or broadcasted henceforth) to the entire selection.
#  any modifications  will be reflected in the source array:

alist = list(range(10))
alist[5] = 10
# alist[5:8] = 10 #error
print(alist)

arr[5:8] = 10
print(arr)

new_arr = arr[5:8].copy()
print(new_arr)

# slice
arr2d = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])
```

```

print(arr2d)
print(arr2d[:2])
print(arr2d[:2, 1:])
print(arr2d[1, :2])
print(arr2d[2, 0])
print(arr2d[2, :1])



arr2d[:2, 1:] = 0
print(arr2d)


# Note that a colon by itself means to take the entire axis,
# so you can slice only higher dimensional axes by doing:

print(arr2d[:, :1])


# boolean Index

# Let's consider an example where we have some data in an array and an array of names with duplicates.
# I'm going to use here the randn function in numpy.random to generate some random normally distributed data:

names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
data = np.random.randn(7, 4)

print(names)
print(data)

#names == "Bob"
#  If we wanted to select all the rows with corresponding name 'Bob'.
#  Like arithmetic operations, comparisons (such as ==) with arrays are also vectorized.
print("Bob data =", data[names == 'Bob'])

print("!Bob data =", data[names != 'Bob'])

print("Partial Bob data =", data[names == 'Bob', 2:])

print("data < 0", data[data < 0])


# data[data < 0]=0
# print("data < 0 is 0", data)

```

```
data [names != 'Bob'] = 0
print(data)

# fancy indexing
# Fancy indexing is a term adopted by NumPy to describe indexing using integer arrays.

arr = np.empty((8,4))
for i in range(8):
    arr[i] = i

print(arr)

# To select out a subset of the rows in a particular order,
# you can simply pass a list or ndarray of integers specifying the desired order:

print(arr[[4,3,0,6]])

print(arr[[2,3]])

# Using negative indices select rows from the end:
print(arr[[-2,-1]])



arr = np.arange(32)
print(arr)

arr =  arr.reshape(8,4)
print(arr)

#arr = np.arange(32).reshape(8,4)

# Passing multiple index arrays does something slightly different;
# it selects a 1D array of elements corresponding to each tuple of indices:
print("intersection elements= ", arr[[1,5,7,2],[0,3,1,2]])

# Take a moment to understand what just happened: the elements (1, 0), (5, 3), (7, 1), and(2, 2) wereselected.
# The behavior of fancy indexing in this case is a bit different from what some users might have expected
# (myself included), which is the rectangular region formed by selecting a subset of the matrix' s rows and columns

print(arr[np.ix_([1,5,7,2],[0,3,1,2])]) # generate new array, different with slice
print("arr=",arr)

#transpose

```

```
# Transposing is a special form of reshaping which similarly returns a view on the un- derlying data
# without copying anything.

arr = np.arange(15).reshape((3,5))

print("arr = ", arr)

print(arr.T)

# When doing matrix computations, you will do this very often,
# like for example computing the inner matrix product XTX using np.dot:

arrDot = np.dot(arr.T, arr)
print("arrDot", arrDot)

# universal function
# A universal function, or ufunc, is a function that performs elementwise operations on data in ndarrays.
# You can think of them as fast vectorized wrappers for simple functions that take one or more scalar values
# and produce one or more scalar results.

arr = np.arange(8)+1
print(arr)

print(np.square(arr))
print(np.exp(arr))
print(np.log(arr))


# case study 1 : mesh computing

# import matplotlib.pyplot as plt
# import numpy as np
#
# points = np.arange(-5,5,0.1) # 1000 points with the same distance
# print(points)
# xs, ys = np.meshgrid(points, points)
#
# print("ys=",ys)
# print("xs=",xs)
#
# z = np.sqrt(xs**2+ys**2)
# print(z)
#
# plt.imshow(z)
# plt.colorbar()
# plt.title('Image plot of $\sqrt{X^2+y^2}$ for a grid of values')
```

```
#


# logic expression
#x if condition else y


arr = np.random.randn(4,4)
print(arr)

# The numpy.where function is a vectorized version of the ternary expression x if condi tion else y.
# print(np.where(arr > 0.5, 2, -2))
print(np.where(arr > 1, 0, arr))


# result = [(2 if c > 0.5 else -2) for c in arr]
# print(result)

alist = list(range(10))
print(alist)

result = [(0 if c >5 else 1) for c in alist]
print(result)

# A set of mathematical functions which compute statistics about an entire array or about the data
# along an axis are accessible as array methods.
# Aggregations (often called reductions) like sum, mean, and standard deviation std can either be used by
# calling the array instance method or using the top level NumPy function

# sum, mean, std, var, min, max , argmin, argmax, cumsum, cumprod
arr = np.arange(9).reshape(3,3)

print(arr)

print(arr.sum())
print(np.sum(arr))

print(arr.mean())
print(arr.std())
print(arr.var())
print(arr.min())
print(arr.max())
print(arr.argmin())
print(arr.argmax())

# Cumulative sum of elements starting from 0
```

```
a = np.array([[1,2,3], [4,5,6]])
print(np.cumsum(a))

col =  np.cumsum(a,axis=0)      # sum over rows for each of the 3 columns
print(col)

row = np.cumsum(a, axis=1)  # sum over columns for each of the 2 rows
print(row)

arr = np.random.randn(8)
# result = arr.sort()
# print(result)
print(np.sort(arr))

#unique
# NumPy has some basic set operations for one-dimensional ndarrays.
# Probably the most commonly used one is np.unique, which returns the sorted unique values in an array:

names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
print(np.unique(names))
print(sorted(set(names)))

#######################################################
#
#                    Pandas - Series
#
#######################################################

from pandas import Series, DataFrame
import pandas as pd

# Thus, whenever you see pd. in code, it’ s referring to pandas.
# Series and DataFrame are used so much that I find it easier to import them into the local namespace.


# A Series is a one-dimensional array-like object containing an array of data (of any NumPy data type)
#  and an associated array of data labels, called its index.
#  The simplest Series is formed from only an array of data:

obj = Series([4,7,-5,3])
print(obj)
print(obj.values)
print(obj.index)

obj2 = Series([4,7,5,3],index=['d','b','a','c'])
print(obj2)
```

```python
print(obj2['a'])
print(obj2[['a','c']])  # double [[]]
print(obj2[obj2 > 0])
print(obj2*2)
print(np.exp(obj2))

# Should you have data contained in a Python dict, you can create a Series from it by passing the dict
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
obj3 = Series(sdata)
print(obj3)

# extract partial data
states = ['California', 'Ohio', 'Oregon', 'Texas']
obj4 = Series(sdata, index=states)
print(obj4)

# check non value

print(pd.isnull(obj4))
print(pd.notnull(obj4))
print(obj4.isnull())

# Both the Series object itself and its index have a name attribute
obj4.name = 'population'
obj4.index.name = 'state'
print(obj4)

#  Series's index can be altered in place by assignment:
print(obj)
obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
print(obj)

######################################################
#
#                    Pandas -  DataFrame
#
######################################################

data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'], 'year': [2000, 2001, 2002, 2001, 2002],
'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}

frame = DataFrame(data)

print(frame)


print(DataFrame(data, columns=['year','state','pop']))
```

```
print(DataFrame(data, columns=['year','state','pop','debt']))

# A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:
print(frame.columns)
print(frame['state'])
print(frame.state)

print(frame)

#Rows can also be retrieved by position or name by a couple of methods, such as the ix indexing field
print(frame.ix[3])

frame['debt'] = 16.5
print(frame)

# For example, the empty 'debt' column could be assigned a scalar value or an array of values
frame['debt'] = np.arange(5.)
print(frame)

# When assigning lists or arrays to a column, the value's length must match the length of the DataFrame.
# If you assign a Series, it will be instead conformed exactly to the DataFrame's index, inserting missing values in any holes:

val = Series([-1.2, -1.5, -1.7], index=[2, 4, 5])
frame['debt'] = val
print(frame)

#Assigning a column that doesn't exist will create a new column.

frame['eastern'] = 1
print(frame)


frame['marks'] = frame.state == 'Ohio'
del frame['eastern']
print(frame)

# Index Objects
obj = Series(range(3), index=['a', 'b', 'c'])
print(obj)

# Index objects are immutable index[1] = 'd'

# Reindexing
# Calling reindex on this Series rearranges the data according to the new index,
```

```
# introducing missing values if any index values were not already present:

obj2 = obj.reindex([’a’, ’b’, ’c’, ’d’, ’e’])
print(obj2)

obj2 = obj.reindex([’a’, ’b’, ’c’, ’d’, ’e’], fill_value=0)
print(obj2)

# For ordered data like time series, it may be desirable to do some interpolation or filling of values when reindexing.
# The method option allows us to do this, using a method such as ffill which forward fills the values:

obj3 = Series([’blue’, ’purple’, ’yellow’], index=[0, 2, 4])
print (obj3)
obj3 = obj3.reindex(range(6), method=’ffill’)
print(obj3)

# ffill or pad : Fill (or carry) values forward, bfill or backfill : Fill (or carry) values backward

# With DataFrame, reindex can alter either the (row) index, columns, or both.

frame = DataFrame(np.arange(9).reshape((3, 3)), index=[’a’, ’c’, ’d’], columns=[’Ohio’, ’Texas’, ’California’])
print(frame)


# When passed just a sequence,  the rows are reindexed in the result:
frame2 = frame.reindex([’a’, ’b’, ’c’, ’d’])
print(frame2)

# The columns can be reindexed using the columns keyword:
states = [’Texas’, ’Utah’, ’California’]
frame = frame.reindex(columns=states)

print(frame)

# Both can be reindexed in one shot, though interpolation will only apply row-wise(axis 0)
frame = frame.reindex(index=[’a’, ’b’, ’c’, ’d’], method=’ffill’, columns=states)
print(frame)


# Dropping entries from an axis

obj = Series(np.arange(5.), index=[’a’, ’b’, ’c’, ’d’, ’e’])
new_obj = obj.drop(’c’)
print(new_obj)
```

```

# With DataFrame, index values can be deleted from either axis:

data = DataFrame(np.arange(16).reshape((4, 4)), index=['Ohio', 'Colorado', 'Utah', 'New York'], columns=['one', 'two', 'three', 'four'])

data.drop(['Colorado', 'Ohio'])

print(data)
data.drop('two', axis=1)
print(data)
# Summarizing and Computing Descriptive Statistics
print(data.describe())

print(data.sum())
print(data.sum(axis =1 ))

data.ix["ohio"] = None
print(data)
data1 = data.mean(axis=0, skipna=True)
print(data1)

#like idxmin and idxmax, return indirect statistics like the index value where the minimum or maximum values are attained:
print(data.idxmax())

#import pandas.io.data as web

# from pandas import Series, DataFrame
# import pandas as pd

import pandas_datareader.data as web
import datetime

# start = datetime.datetime(2013, 1, 1)
# end = datetime.datetime(2017, 3, 7)
# df = web.DataReader("GOOGL", 'yahoo', start, end)
#
# print("df=",df)
#
# price = df['Adj Close']
#
# # price = DataFrame({tic: df['Adj Close'] for tic, data in df.iteritems()})
#
# print("price =", price)
#
# returns = price.pct_change()
# print(returns.tail())

```

```python
start = datetime.datetime(2017, 1, 1)
end = datetime.datetime(2017, 3, 7)
all_data = {}

for ticker in ['AAPL','IBM','GOOG']:
    all_data[ticker] = web.DataReader(ticker, 'yahoo', start, end)



# draw graph

import matplotlib as mpl
import matplotlib.pyplot as plt
#matplotlib inline

np.random.seed(1000)
y = np.random.standard_normal(20)
x = range(len(y))

plt.plot(x,y)

price = DataFrame({tic: data['Adj Close'] for tic, data in all_data.items()})
volume = DataFrame({tic: data['Volume'] for tic, data in all_data.items()})

print("price = \n", price.tail(5))
print("price AAPL= \n", price['AAPL'])
print("volume = \n", volume.tail(5))

#plt.plot(price['AAPL'])

returns = price.pct_change()
print(returns.tail(10))

# The corr method of Series computes the correlation of the overlapping, non-NA,
# aligned-by-index values in two Series. Relatedly, cov computes the covariance:
cov = returns.AAPL.corr(returns.IBM)
print(cov)

# DataFrame's corr and cov methods, on the other hand,
# return a full correlation or covariance matrix as a DataFrame, respectively:

print(returns.corr())
print(returns.cov())

# Using DataFrame's corrwith method, you can compute pairwise correlations between a DataFrame's columns or rows with another Series or DataFrame.
# Passing a Series returns a Series with the correlation value computed for each column:

```

```python
print("IBM\n",returns.corrwith(returns.IBM))

#Passing a DataFrame computes the correlations of matching column names.
# Here I compute correlations of percent changes with volume:
print("returns with volume\n",returns.corrwith(volume))

#dates =[]

# for x in range(len(df)):
#     #print(str(df.index[x]))
#     newdate = str(df.index[x])
#     newdate = newdate[0:10]
#     #print("newdata =",newdate)
#     dates.append(newdate)
#
# df['dates'] = dates

#print (df.head(5))
#print (df.tail(10))

# obj = Series([1,1.01,1.01,1.01])
# print(obj.pct_change())
```