DATA WRANGLING USING NUMPY

Let's practice Numpy

NUMPY

Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with the arrays. It also has functions for working in domain of linear algebra, fourier transform, and matrices. NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.

NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of non-negative integers. In NumPy dimensions are called axes. For example, the coordinates of a point in 3D space [1, 2, 1] has one axis. That axis has 3 elements in it, so we say it has a length of 3. In the example pictured below, the array has 2 axes. The first axis has a length of 2, the second axis has a length of 3.

```
[[1 2 3]
 [4 5 6]]
```

Arrays

NumPy's array class is called ndarray. It is also known by the alias array. Note that numpy.array is not the same as the Standard Python Library class array.array, which only handles one-dimensional arrays and offers less functionality. A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension. The more important attributes of an ndarray object are:

ndarray.ndim	the number of axes (dimensions) of the array.
ndarray.shape	the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with n rows and m columns, shape will be (n,m). The length of the shape tuple is therefore the number of axes, ndim.
ndarray.size	the total number of elements of the array. This is equal to the product of the elements of shape.
ndarray.dtype	an object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.
ndarray.itemsize	the size in bytes of each element of the array. For example, an array of elements of type float64 has itemsize 8 (=64/8), while one of type complex32 has itemsize 4 (=32/8). It is equivalent to ndarray.dtype.itemsize.
ndarray.data	the buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using

	indexing facilities.

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

```
import numpy as np

x = range(16)
x = np.reshape(x,(4,4))
print(x)

a = np.array([1, 2, 3])   # Create a rank 1 array (Single row)
print(a)                  # Prints "[1, 2, 3]"
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b)
print(b.shape)                     # Prints "(2, 3)" 2 rows 3
columns
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

Numpy also provides many functions to create arrays:

```
import numpy as np

a = np.zeros((2,2))   # Create an array of all zeros
print(a)
b = np.ones((1,2))

c = np.full((2,2), 7)  # Create a constant array
print(c)
d = np.eye(2)         # Create a 2x2 identity matrix
print(d)

e = np.random.random((2,2))
print(e)
```

Output

```
[[0. 0.]

 [0. 0.]]
```

```
[[7 7]
 [7 7]]
[[1. 0.]
 [0. 1.]]
[[0.89403954 0.24898726]
 [0.9919712  0.52524809]]
```

Array indexing

Numpy offers several ways to index into arrays.

Slicing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

```
import numpy as np

# Create a rank 2 array with shape (3, 4)
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(a)
# Use slicing to pull out the subarray consisting of the first
2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying
it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"

#You can also mix integer indexing with slice indexing.
# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower
rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)
print(row_r2, row_r2.shape)

# We can make the same distinction when accessing columns of an
array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)
```

```
print(col_r2, col_r2.shape)
```

Datatypes

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is an example:

```
import numpy as np

# Let numpy choose the datatype
x = np.array([1, 2])
print(x.dtype)  # Prints "int32"
# Force a particular datatype
x = np.array([1, 2], dtype=np.int64)
print(x.dtype)   # Prints "int64"

# Let numpy choose the datatype
x = np.array([1.0, 2.0])
print(x.dtype) # Prints "float64"
```

Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

```
import numpy as np
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce the array
print(x * y)
print(np.multiply(x, y))

# Matrix Multiplication; both produce the array
print(x @ y)
print(np.matmul(x, y))
```

```
#Dot Multiplication
print(x.dot(y))

# Elementwise division; both produce the array
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
print(np.sqrt(x))
```

Dot Product:

For 2-D vectors, it is the equivalent to matrix multiplication. For 1-D arrays, it is the inner product of the vectors. For N-dimensional arrays, it is a sum product over the last axis of a and the second-last axis of b.

Examples to Practice

```
import numpy as np

A = np.arange(3)
print("A: \n",A)
A = np.arange(10)**3
print(A)
print("Slicing: ",A[2:5])
#Similar to String or List slicing
#Iterating through the Array
print("Calculating cube root of the values:")
for i in A:
    print(i**(1/3))

c = np.array(
    [[[0, 1, 2],  # a 3D array (two stacked 2D arrays)
    [10, 12, 13]],
    [[100, 101, 102],
    [110, 112, 113]]])
print("Shape of the matrix is: ", np.shape(c))
print(c[1,...])
print("Matrix C: \n",c.reshape(6,2))

#Iterating in Flat format
print("Flat Format:")
for element in c.flat:
    print(element)
```

More Operations using Numpy

```
import numpy as np
```

```
#Splitting one array into several smaller ones
A = np.arange(9)**3
A.resize(3,3)
print("Original Matrix: \n",A)
# Split a into 3
print("Horizontal Split into 3: ",np.hsplit(A,3))
print("Vertical Split into 3: ",np.vsplit(A,3))
```

Output:

```
Original Matrix:
 [[  0   1   8]
 [ 27  64 125]
 [216 343 512]]
Horizontal Split into 3:  [array([[  0],
       [ 27],
       [216]], dtype=int32), array([[  1],
       [ 64],
       [343]], dtype=int32), array([[  8],
       [125],
       [512]], dtype=int32)]
Vertical Split into 3:  [array([[0, 1, 8]], dtype=int32), array([[ 27,  64, 125]], dtype=int32), array([[216, 343, 512]], dtype=int32)]
```

Copy and View in Numpy

```
import numpy as np

a = np.array([[ 0,  1,  2,  3],
              [ 4,  5,  6,  7],
              [ 8,  9, 10, 11]])
b = a    # no new object is created
print(b is a)  # a and b are two names for the same ndarray object
print("ID of a: ",id(a))
print("ID of b: ",id(b))
#Value you will get may vary but you will see both has same value
c = a.view()
print(c is a)
# output: False as c is a view of the data owned by a
print("Checking base: ",c.base is a)
# Output: True as c is a view of the data owned by a
#Deep Copy using copy
d = a.copy()
print(c is a)
# output: False as d is a new array object
print("Checking base: ",c.base is a)
# Output: False as d doesn't share anything with a
```

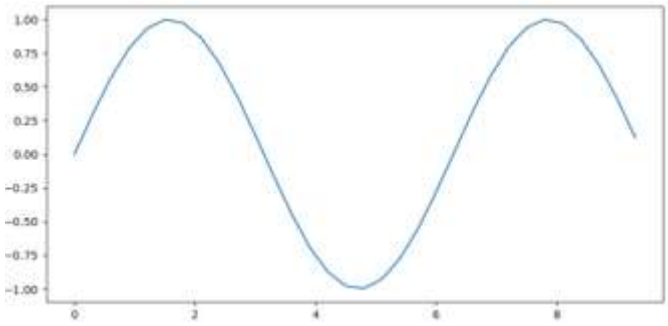
Output:

```
True
ID of a:  50935512
ID of b:  50935512
False
Checking base:  True
False
Checking base:  True
```

Special Function

Plot a Sin graph using Numpy:

```
import numpy as np
import matplotlib.pyplot as plt
x= np.arange(0, 3*np.pi,0.3)
y = np.sin(x)
plt.plot(x,y)
plt.show()
```



LINEAR ALGEBRA: MATRIX APPLICATIONS USING NUMPY

```
#Program to print Identity Matrix of order 'n', where n is an
integer
import numpy as np
# 2x2 matrix with 1's on main diagnol
b = np.identity(2, dtype = float)
print("Matrix b : \n", b)
a = np.identity(4)
print("\n Matrix a : \n", a)
```

- An Identity Matrix is a **square matrix** whose main diagonal elements are **1** and all the other elements are **0**.
- Identity Matrix is also called as **Unit Matrix.**

Example: $A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

$B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Python Program to Find the Transpose of a Matrix

Transpose of a matrix is the interchanging of rows and columns. It is denoted as X'. The element at ith row and jth column in X will be placed at jth row and ith column in X'. So if X is a 3x2 matrix, X' will be a 2x3 matrix.

```
import numpy
matrix=[[1,2,3],[4,5,6]]
print(matrix)
print("\n")
print(numpy.transpose(matrix))
```

Matrix Multiplication

- Two matrices A and B may be multiplied only if matrix A and matrix B obey following rule:
  - The number of columns of A **equals** the number of rows of B.
- In general, if A is an m × n matrix and B is an n × p matrix, then A × B is possible and the order of the resulting product matrix AB will be **m × p**.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} = \begin{bmatrix} (aj+bm+cp) & (ak+bm+cq) & (al+bo+cr) \\ (dj+em+fp) & (dk+en+fq) & (dl+eo+fr) \\ (gj+hm+ip) & (gk+hn+iq) & (gl+ho+ir) \end{bmatrix}$$

```
# Python code to demonstrate matrix operations
# mul()
# importing numpy for matrix operations
import numpy
# initializing matrices
x = numpy.array([[10, 20, 50], [5, 8, 8]])
y = numpy.array([[9, 8, 9], [4, 3, 2]])
# using mul()  multiply matrices
print ("The element wise multiplication of matrix is : ")
print (numpy.multiply(x,y))
```

Inverse of a Matrix

```
# Python Program to Compute Determinant of a Matrix
import numpy as np
b = np.array([[6,1,1], [4, -2, 5], [2,8,7]])
print(b)
print(np.linalg.det(b))
#alternative: manually calculating
print(6*(-2*7 - 5*8) - 1*(4*7 - 5*2) + 1*(4*8 - -2*2))
```

- Determinant of a Matrix is a special number that is defined only for **Square Matrix**.
- The **symbol** for determinant is two vertical lines either side.

  Example: |**A**| means the determinant of the matrix **A**

Determinant of 2 X 2 Matrix

$$M = \begin{bmatrix} p & q \\ r & s \end{bmatrix}$$

$$\det(M) = (p*s)-(q*r)$$

Inverse of a Matrix

- For a square matrix **A**, the inverse is written $\mathbf{A^{-1}}$. When **A** is multiplied by $\mathbf{A^{-1}}$ the result is the **Identity matrix 'I'**.
- Non-square matrices do not have inverses.

$$\mathbf{A\,A^{-1} = A^{-1}\,A = I}$$

For Example, Inverse of **2 X 2** matrix, $A = \begin{bmatrix} p & q \\ r & s \end{bmatrix}$

$$A^{-1} = \frac{1}{|A|}\begin{bmatrix} p & q \\ r & s \end{bmatrix} = \frac{1}{|ad-bc|}\begin{bmatrix} s & -q \\ -r & p \end{bmatrix}$$

```
#Printing inverse of a matrix
import numpy as np
x = np.array([[1,2],[3,4]])
y = np.linalg.inv(x)
print(x)
print(y)
print(np.dot(x,y))
```

Solving Linear Equation Problem using Numpy

- A system of Linear Equations can be represented in matrix form using a coefficient matrix, a variable matrix, and a constant matrix.
- Consider the system,

  1a + 1b = 35

  2a + 4b = 94

  Coefficient matrix , $A = \begin{bmatrix} 1 & 1 \\ 2 & 4 \end{bmatrix}$

  Variable Matrix, $X = \begin{bmatrix} a \\ b \end{bmatrix}$

  Constant Matrix, $B = \begin{bmatrix} 35 \\ 94 \end{bmatrix}$

Therefore in short linear equations represented as : **A.X = B**

Variables are calculated by finding product of A-1 and B: **$X = A^{-1} . B$**

```
import numpy as np
#A: Coefficient Matrix
A=[[1,1],[2,4]]
A=np.array(A)
#print(type(A))
print(A)
#B: Constant Matrix
B=[[35],[94]]
B=np.array(B)
#print(type(B))
print(B)

detA=np.linalg.det(A)
if detA==0:
    print("Solution is not possible!")
else:
    InvA = np.linalg.inv(A)
    C= np.matmul(InvA,B)
    print("Solution: \n",C)
```

Find the solution to the following system of equations.

$$2x + 5y + 2z = -38$$
$$3x - 2y + 4z = 17$$
$$-6x + y - 7z = -12$$

Solving using Numpy:

```
import numpy as np
#A: Coefficient Matrix
A=[[2,5,2],[3,-2,4],[-6,1,-7]]
A=np.array(A)
#print(type(A))
print(A)
#B: Constant Matrix based on Solution
b=[[-38],[17],[-12]]
b=np.array(b)
#print(type(B))
print(b)
detA=np.linalg.det(A)
if detA==0:
    print("Solution is not possible!")
else:
    InvA = np.linalg.inv(A)
    C= np.matmul(InvA,b)
    print("Solution: x = {}, y= {},
z={}".format(C[[0]],C[[1]],C[[2]]))
```

Output:

```
[[ 2  5  2]
 [ 3 -2  4]
 [-6  1 -7]]
[[-38]
 [ 17]
 [-12]]
Solution: x = [[3.]], y= [[-8.]], z=[[-2.]]
```

DATA WRANGLING USING SCIPY

SciPy is a collection of mathematical algorithms and convenience functions built on the NumPy extension of Python. It adds significant power to the interactive Python session by providing the user with high-level commands and classes for manipulating and visualizing data. With SciPy, an interactive Python session becomes a data-processing and system-prototyping environment rivaling systems, such as MATLAB, IDL, Octave, R-Lab, and SciLab.

The additional benefit of basing SciPy on Python is that this also makes a powerful programming language available for use in developing sophisticated programs and specialized applications. Scientific applications using SciPy benefit from the development of additional modules in numerous niches of the software landscape by developers across the world. Everything from parallel programming to web and data-base subroutines and classes have been made available to the Python programmer. All of this power is available in addition to the mathematical libraries in SciPy.

- SciPy is built in top of the NumPy
- SciPy is a fully-featured version of Linear Algebra while Numpy contains only a few features.
- Most new Data Science features are available in Scipy rather than Numpy.

You can also install SciPy in Windows via pip

```
Python3 -m pip install --user numpy scipy
```

Install Scipy on Linux

```
sudo apt-get install  python-scipy python-numpy
```

Install SciPy in Mac

```
sudo port install py35-scipy py35-numpy
```

Different subpackages of SciPy

Subpackage	Description
cluster	Clustering algorithms
constants	Physical and mathematical constants
fftpack	Fast Fourier Transform routines
integrate	Integration and ordinary differential equation solvers
interpolate	Interpolation and smoothing splines
io	Input and Output

linalg	**Linear algebra**
ndimage	**N-dimensional image processing**
odr	**Orthogonal distance regression**
optimize	**Optimization and root-finding routines**
signal	**Signal processing**
sparse	**Sparse matrices and associated routines**
spatial	**Spatial data structures and algorithms**
special	**Special functions**
stats	**Statistical distributions and functions**

In this chapter, we will cover important sub-packages relevant to us.

File IO (scipy.io)

Scipy, I/O package, has a wide range of functions for work with different files format which are Matlab, Arff, Wave, Matrix Market, IDL, NetCDF, TXT, CSV and binary format.

```
import numpy as np
from scipy import io as sio
#creating an array using Numpy
array = np.ones((4, 4))
#Saving to file
sio.savemat('array.mat', {'ar': array})
#reading from the file
data = sio.loadmat('array.mat', struct_as_record=True)
print("Print the content: \n",data)
#printing only the array part
print("Print the array: \n",data['ar'])
```

Special Package (scipy.special)

SciPy also gives functionality to calculate Permutations and Combinations. In this section, we will only cover sample program. We have covered this topic in detail in our book "*Essential Probability & Statistics for Machine Learning (Implementation using Python) By Swapnil Saurav*".

Permutation and combination are the ways to represent a group of objects by selecting them in a set and forming subsets. It defines the various ways to arrange a certain group of data. When we select the data or objects from a certain group it is said to be permutations whereas the order in which they are represented is called combination.

Permutation Formula

A permutation is the choice of r things from a set of n things without replacement and where the order matters.

^nP_r = (n!) / (n-r)!

Combination Formula

A combination is the choice of r things from a set of n things without replacement and where order does not matter.

$$ {}_nC_r = \binom{n}{r} = \frac{{}_nP_r}{r!} = \frac{n!}{r!(n-r)!} $$

Problem 1: In a group of 6 boys and 4 girls, four children are to be selected. In how many different ways can they be selected such that at least one boy should be there?

Solution: We have 4 options as given below:

- We can select 4 boys: 6C4
- We can select 3 boys and 1 girl: 6C3 × 4C1
- We can select 2 boys and 2 girls: 6C2 × 4C2
- We can select 1 boy and 3 girls: 6C1 × 4C3

```
from scipy.special import comb

sum=0
#find combinations of 5, 2 values using comb(N, k)
#We can select 4 boys: 6C4
com = comb(6, 4, exact = False, repetition=False)
sum+=com
#We can select 3 boys and 1 girl: 6C3 × 4C1
com = comb(6, 3) * comb(4, 1)
sum+=com
#We can select 2 boys and 2 girls: 6C2 × 4C2
com = comb(6, 2) * comb(4,2)
sum+=com
#We can select 1 boy and 3 girls: 6C1 × 4C3
com = comb(6, 1) * comb(4, 3)
sum+=com
print("Total combination possible: ",sum)
```

Output:

```
Total combination possible:  209.0
```

The number of combinations of N things taken k at a time. This is often expressed as “N choose k”:

```
scipy.special.comb(N, k, exact=False, repetition=False)
```

Parameters

- Nint, ndarray - Number of things.
- kint, ndarray - Number of elements taken.
- exactbool, optional - If exact is False, then floating point precision is used, otherwise exact long integer is computed.
- repetitionbool, optional - If repetition is True, then the number of combinations with repetition is computed.

Problem 2: Three men have 4 coats, 5 waist coats, and 6 caps. In how many ways can they wear them?

Solution:

Number of ways in which 3 Men can wear 4 coats = 4!/1! = 4! = 24

Number of ways in which 3 Men can wear 5 waist coats = 5 X 4 X 3 = 60

Number of ways in which 3 Men can wear 6 caps = 6 X 5 X 4 = 120

By Fundamental Principle of Multiplication

Arrangement of 4 Coats AND Arrangement of 5 Waist Coats AND Arrangement of 6 Caps

Total number of ways = 24 X 60 X 120 = 172,800

```
from scipy.special import perm

product=1
#find permutation of 4, 3 using perm (N, k) function
#Number of ways in which 3 Men can wear 4 coats
per = perm(4, 3)
product*=per
#Number of ways in which 3 Men can wear 5 waist coats
per = perm(5, 3)
product*=per
#Number of ways in which 3 Men can wear 6 caps
per = perm(6, 3)
product*=per

print("Total Permutation possible: ",product)
```

Output:

```
Total Permutation possible:  172800.0
```

Permutations of N things taken k at a time, i.e., k-permutations of N. It’s also known as “partial permutations”.

```
scipy.special.perm(N, k, exact=False)
```

Parameters

- Nint, ndarray - Number of things.
- kint, ndarray - Number of elements taken.
- exactbool, optional - If exact is False, then floating point precision is used, otherwise exact long integer is computed.

Bessel functions of real order(jv, jn_zeros)

Bessel functions are a family of solutions to Bessel's differential equation with real or complex order alpha:

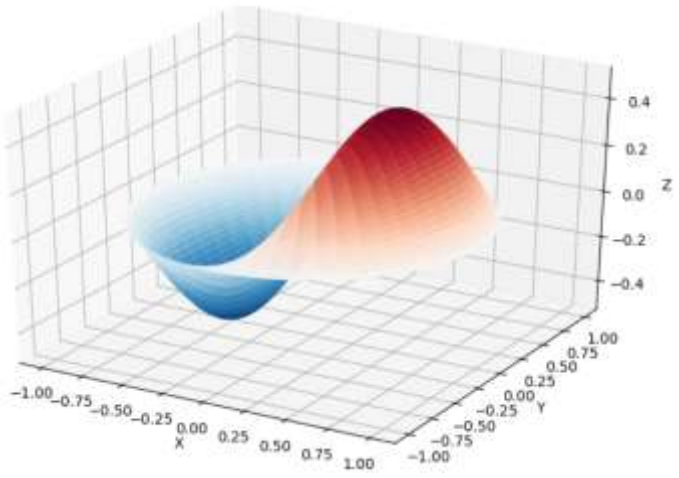
$$x^2 \frac{d^2y}{dx^2} + x\frac{dy}{dx} + (x^2 - \alpha^2)y = 0$$

Among other uses, these functions arise in wave propagation problems, such as the vibrational modes of a thin drum head. Here is an example of a circular drum head anchored at the edge:

```
from scipy import special
import numpy as np
def drumhead_height(n, k, distance, angle, t):
   kth_zero = special.jn_zeros(n, k)[-1]
   return np.cos(t) * np.cos(n*angle) * special.jn(n,
distance*kth_zero)
theta = np.r_[0:2*np.pi:50j]
radius = np.r_[0:1:50j]
x = np.array([r * np.cos(theta) for r in radius])
y = np.array([r * np.sin(theta) for r in radius])
z = np.array([drumhead_height(1, 1, r, theta, 0.5) for r in radius])

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
fig = plt.figure()
ax = Axes3D(fig)
ax.plot_surface(x, y, z, rstride=1, cstride=1, cmap='RdBu_r',
vmin=-0.5, vmax=0.5)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
plt.show()
```

Output



Solving Linear Modelling using Scipy

Linear programming (LP) (also called linear optimization) is a method to achieve the best outcome (such as maximum profit or lowest cost) in a mathematical model whose requirements are represented by linear relationships. Linear programming is a special case of mathematical programming (mathematical optimization).

Linear programming is an extention of Linera algebra we discussed in Numpy. In this section we will solve LP problems with Scipy. You see there are some equations on next page. Here the first equation (to be maximed) is called the Objective function. We either maximize (e.g. Sales, Profit, etc) or minimize (e.g. Cost, Wastage, etc) using this function. We can easily convert a maximization problem into minimization by multiplying by -1 on both the sides. This function is subjected to some constraints. We can solve by variety of ways. In this section, we will see the graph method, Excel solution and Python Scipy package.

- A **linear function to be maximized**

  e.g. $f(x_1, x_2) = c_1 x_1 + c_2 x_2$
- **Problem constraints** of the following form

  e.g.

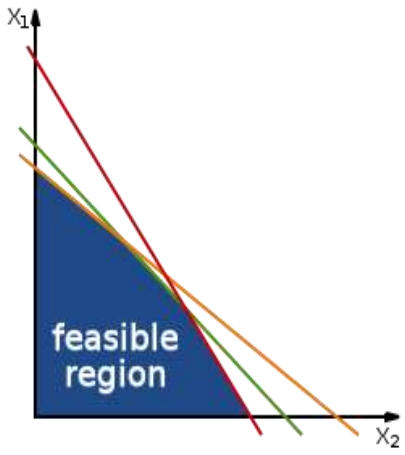
  $$a_{11} x_1 + a_{12} x_2 \leq b_1$$
  $$a_{21} x_1 + a_{22} x_2 \leq b_2$$
  $$a_{31} x_1 + a_{32} x_2 \leq b_3$$
- **Non-negative variables**

  e.g.

  $$x_1 \geq 0$$
  $$x_2 \geq 0$$

The problem is usually expressed in *matrix form*, and then becomes:

$$\max\{\mathbf{c}^T \mathbf{x} \mid A\mathbf{x} \leq \mathbf{b} \wedge \mathbf{x} \geq 0\}$$



Problem Statement:

Let's take a problem to understand the aim and also solve using different methods. Indigo Computers is a manufacturer of Notebooks and Desktops. They make quarterly decisions about their product mix. Indigo Computers would like to know how many of each product to produce in order to **maximize** profit for the quarter. As like any business, Indigo too has some major constraints like:

- Each computer (either notebook or desktop) requires a Processing Chip. Due to tightness in the market, their supplier has allocated only 10,000 such chips to Indigo.
- Each computer requires memory. Memory comes in 16MB chip sets. A notebook computer has 16MB memory installed (so needs 1 chip set) while a desktop computer has 32MB (so requires 2 chip sets). As per the contract negotiation with the suppliers, they have a stock of 15,000 chip sets to use over the next quarter.
- Each computer requires assembly time. Due to tight tolerances, a notebook computer takes more time to assemble: 4 minutes versus 3 minutes for a desktop. There are 25,000 minutes of assembly time available in the next quarter.

Given current market conditions, each notebook computer produced generates Rs 750 profit, and each desktop produces Rs 1000 profit.

There are many questions Indigo Computer might ask. The most obvious are such things as:

- How many of each type computer should Indigo Computer produce in the next quarter? What is the maximum profit Indigo Computer can make?"

Less obvious, but perhaps of more managerial interests are:

- How much should Indigo Computer be willing to pay for an extra memory chip set?
- What is the effect of losing 1,000 minutes of assembly time due to an unexpected machine failure?
- How much profit would we need to make on a 32MB notebook computer to justify its production?

Linear programming gives us a mechanism for answering all of these questions quickly and easily. There are three steps in applying linear programming: modeling, solving, and interpreting.

Modeling a problem using linear programming involves writing it in the language of linear programming. Key to a linear program are the decision variables, objective, and constraints.

Decision Variables: The decision variables represent (unknown) decisions to be made. For this problem, the decision variables are the number of notebooks to produce and the number of desktops to produce. We will represent these unknown values by x_1 and x_2 respectively. To make the numbers more manageable, we will let x_1 be the number of 1000 notebooks produced (so x_1 = 5 means a decision to produce 5000 notebooks) and x_2 be the number of 1000 desktops. Note that a value like the quarterly profit is not (in this model) a decision variable: it is an outcome of decisions x_1 and x_2.

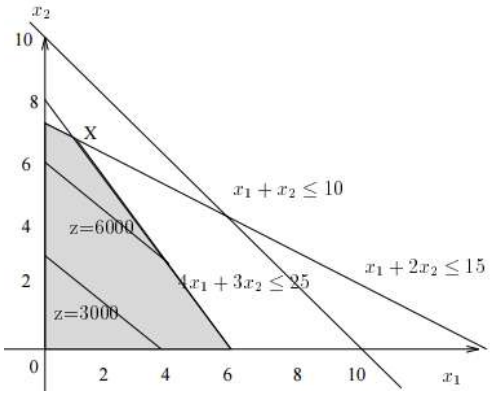
Objective: The objective is to be either minimized or maximized. In this case, our objective is to maximize the function 750x_1 + 1000x_2

Constraints: Here we have four types of constraints: Processing Chips, Memory Sets, Assembly, and Nonnegativity.

- In order to satisfy the limit on the number of chips available, it is necessary that x_1 + x_2 <= 10. If this were not the case (say x_1 = x_2= 6), the decisions would not be implementable (12,000 chips would be required, though we only have 10,000).
- The constraint for memory chip sets is x_1 + 2x_2 <= 15, a linear constraint.
- Our constraint on assembly can be written 4x_1 + 3x_2 <= 25, again a linear constraint.
- Linear constraints x_1 >= 0, x_2 >= 0 to enforce nonnegativity of production.

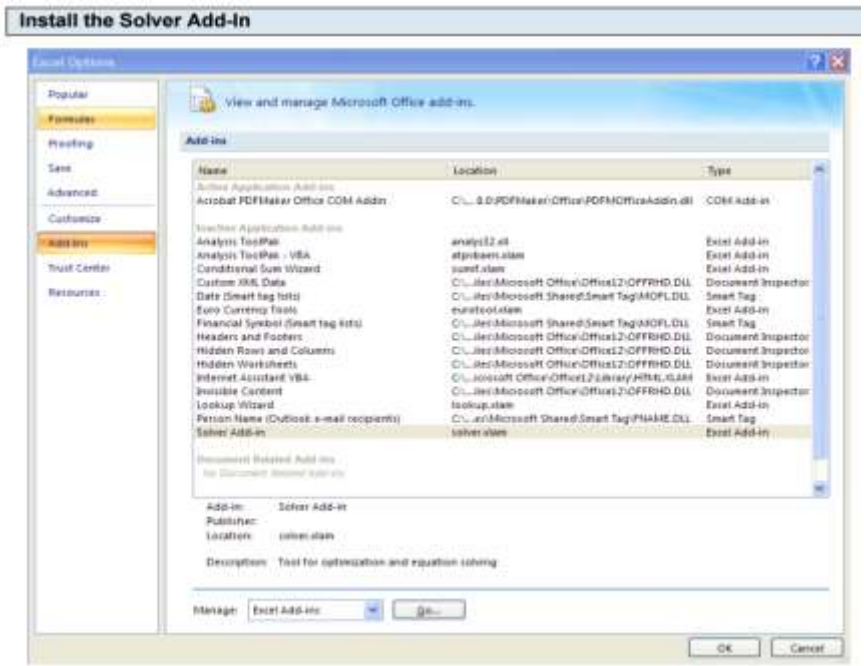
Final Model: This gives us the complete model of this problem:

$$
\begin{aligned}
x_1 + x_2 &\leq 10 \\
x_1 + 2x_2 &\leq 15 \\
4x_1 + 3x_2 &\leq 25 \\
x_1 &\geq 0 \\
x_2 &\geq 0
\end{aligned}
$$

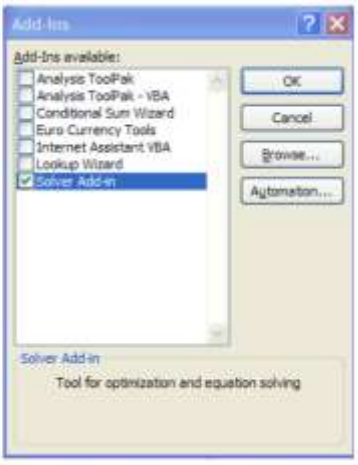


To find the optimal solution, we can include the objective function on this diagram by drawing iso-profit lines: lines along which the profit is the same. Since our goal is to maximize the profit, we can push the isoprofit line out until moving it any further would result in no feasible point (see diagram, z represents profit). Clearly the optimal profit occurs at point X. Note that X is the intersection of the constraints. The solution here is x_1 = 1 and x_2 = 7. The optimal decision is to produce 1,000 notebooks and 7,000 desktops, for a profit of Rs 7,750,000.

Let's solve using Excel first then we will use Numpy to solve.



1. In the Microsoft Office button, go to excel options to click Add-ins
2. In the Add-Ins box, select Solver Add-In and click Go...

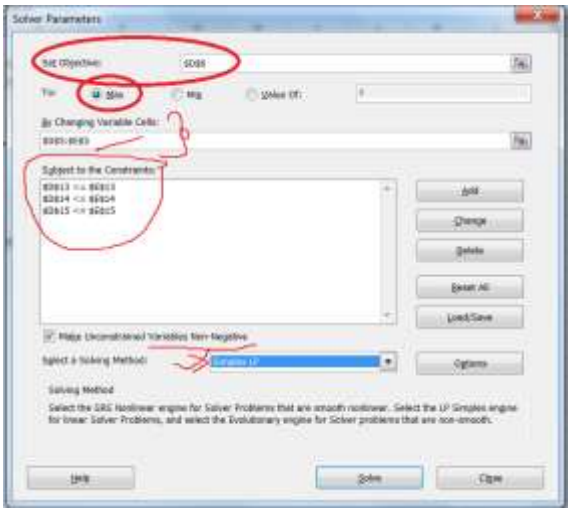


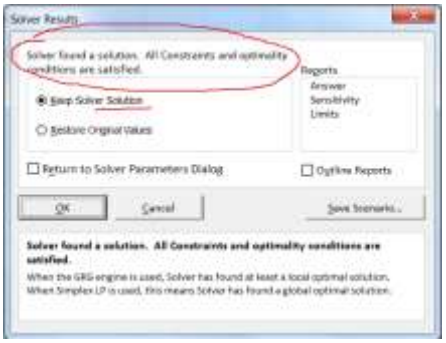
3. In the Add-Ins available box, check the Analysis ToolPak and then OK

	A	B	C	D	E	F	G	H	I
1									
2				Notebook	Desktop				
3			Variables	X1	X2	(Units)			
4			Coefficient	750	1000	(profit per unit)			
5			Solution			(# of units)			
6			Z	=SUMPRODUCT(D4:E4,D5:E5)					
7				SUMPRODUCT(array1, [array2], [array3], [array4], ...)					
8			Constraint 1	1	1	<=	10,000	Processing Chip	
9			Constraint 2	1	2	<=	15,000	Memory	
10			Constraint 3	4	3	<=	25,000	Assembly time	
11									
12				LHS	RHS				
13			Constraint 1	0	10,000	=SUMPRODUCT(D5:E5,D8:E8)			
14			Constraint 2	0	15,000	=SUMPRODUCT(D5:E5,D9:E9)			
15			Constraint 3	0	25,000	=SUMPRODUCT(D5:E5,D10:E10)			
16									

Formulatng the problem:

1. Enter the coefficients of the objective function Z i.e., (750, 1000) in cells D4 and E4. Each unit of Notebook and Desktop will give profit of Rs. 750 and Rs 1,000.
2. Enter the coefficients of the Constraint-1 i.e., (1,1) and RHS value 10,000 in cells D8, E8 and G8 respectively. This is because 1 unit of Processing chip will be consumed by Notebook and Desktop and since the total Processing chip available is 10,000, consumed Processing chips by both should be less than 10,000
3. Enter the coefficients of the Constraint-2 i.e., (1,2) and RHS value 15,000 in cells D9, E9 and G9 respectively. This represents Memory consumption by each unit of Notebook and Desktop.
4. Enter the coefficients of the Constraint-3 i.e., (4,3) and RHS value 25,000 in cells D10, E10 and G10 respectively. This represents Assembly time constraint by each unit of Notebook and Desktop.
5. Solution (C5) represents the number of units that are needed to be produced to meet maximum profit (D5 & E5). We have to find these value so for now leave them blank.
6. Total Profit (Z) is the objective function and it will calculated as: D4*D5 + E4*E5, which can be represented as SUMPRODUCT in excel as shown above. D6 will hold the value and we need to maximize this number using the Solver.





7. Now we represent the values of the constraints to get maximum profit. These values are also calculated as the SUMPRODUCT as shown above under LHS (Left Hand Side) header. The Right Hand Side values (RHS) will be the maximum values that these constraints can take.

Implementing the Solution

1. Start the Solver
2. Set the Objective which is D6 in our example
3. Objective needs to be maximized since we are finding the maximum profit that's possible within given constraints
4. Enter the three constraints. The LHS value needs to be less than or equal to RHS values hence we have entered LHS and RHS cell references
5. We will be solving using the Simplex Method hence select the Simplex method. Simplex method is the most popular method to solve LP Problems. You can refer additional material to learn more about the method. Now click on Solve button.

	Notebook	Desktop			
Variables	X1	X2	(Units)		
Coefficient	750	1000	(profit per unit)		
Solution	1000	7000	(# of units)		
Z	7750000		(Total Profit)		
Constraint 1	1	1	<=	10,000	Processing Chip
Constraint 2	1	2	<=	15,000	Memory
Constraint 3	4	3	<=	25,000	Assembly time
	LHS	RHS			
Constraint 1	8000	10,000			
Constraint 2	15000	15,000			
Constraint 3	25000	25,000			

Now, let's solve using Scipy package: Relook at the equations:

$$
\begin{aligned}
x_1 + x_2 &\leq 10 \\
x_1 + 2x_2 &\leq 15 \\
4x_1 + 3x_2 &\leq 25 \\
x_1 &\geq 0 \\
x_2 &\geq 0
\end{aligned}
$$

Now, we need to form the Matrices:

$$
\begin{pmatrix} [[1\ 1] \\ [1\ 2] \\ [4\ 3] \\ [1\ 0] \\ [0\ 1]] \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} [[10] \\ [15] \\ [25] \\ [\ 0] \\ [\ 0]] \end{pmatrix}
$$

Using the Optimize Module in SciPy

When you need to optimize the input parameters for a function, scipy.optimize contains a number of useful methods for optimizing different kinds of functions:

- minimize_scalar() and minimize() to minimize a function of one variable and many variables, respectively
- curve_fit() to fit a function to a set of data
- root_scalar() and root() to find the zeros of a function of one variable and many variables, respectively
- linprog() to minimize a linear objective function with linear inequality and equality constraints

scipy.optimize also includes the more general minimize(), which we will use for our problem. This function can handle multivariate inputs and outputs and has more complicated optimization algorithms to be able to handle this. In addition, minimize() can handle constraints on the solution to your problem. You can specify three types of constraints:

- LinearConstraint: The solution is constrained by taking the inner product of the solution x values with a user-input array and comparing the result to a lower and upper bound.
- NonlinearConstraint: The solution is constrained by applying a user-supplied function to the solution x values and comparing the return value with a lower and upper bound.
- Bounds: The solution x values are constrained to lie between a lower and upper bound.

Our objective function is to maximize but the function available is minimize so, we need to change the sign of our objective function.

Original function: maximize (750x_1 + 1000x_2)

Now, minimize (750x_1 +1000x_2)*(-1) = -750x_1 – 1000x_2 – multiplied by -1

```
import numpy as np
from scipy.optimize import minimize, LinearConstraint
from scipy.optimize import linprog

#maximize profit - starting with some random values for variables
x_1 = 10 #the number of 1000 notebooks
x_2 = 10 #x_2 be the number of 1000 desktops
profit_per_nb = 750
profit_per_dt = 1000
```

```
profit = (profit_per_nb * x_1 + profit_per_dt * x_2) *1000 #in '000
#to minimize
##  -profit =  -profit_per_nb * x_1 - profit_per_dt * x_2
obj = [-profit_per_nb, - profit_per_dt]  #Coefficient of x and y
variables

lhs_constraints_ineq = [[1,1], # equation 1
                        [1,2],      # equation 2
                        [4,3]]      # equation 3
rhs_value_ineq = [10,     # equation 1
                  15,     # equation 2
                  25]     # equation 3

#lhs_eq & rhs_eq if we have any equality constraints,
#In our example we do not have hence ignoring

#Below mentioning bound condition -
# x_1 and x_2 need to be positive integer numbers hence:
bnd = [(0, float("inf")),  # Bounds of x_1
       (0, float("inf"))]  # Bounds of x_2

opt = linprog(c=obj, A_ub=lhs_constraints_ineq, b_ub=rhs_value_ineq,
    bounds=bnd,      #you can add equality contraints to: A_eq and b_eq
    method="revised simplex")
print(opt)
if opt.success:
    print("Total number of Notebooks that should be manufactured: ",
opt.x[0]*1000)
    print("Total number of Desktops that should be manufactured: ",
opt.x[1]*1000)
    print("Maximum profit that can be realized: ", -1*opt.fun*1000)
#Time to change the sign

else:
    print("There is no solution to the problem!")

#Plot the graphs now
import matplotlib.pyplot as plt

# Construct lines
# x_1 > 0 & x_2 >0
x1 = np.linspace(0, 20, 8000)
# x_2 > 0
y1 = (x1*0)
# Eq1: x1+x2 <=10
y2 = 10-x1
# Eq2: x1+2x2 <=15
y3 = (15-x1)/2.0
# Eq3: 4x1+3x2 <=25
y4 = (25-4*x1)/3.0

# Make plot
#plt.plot(x1, y1, label=r'$y\geq0$')
```

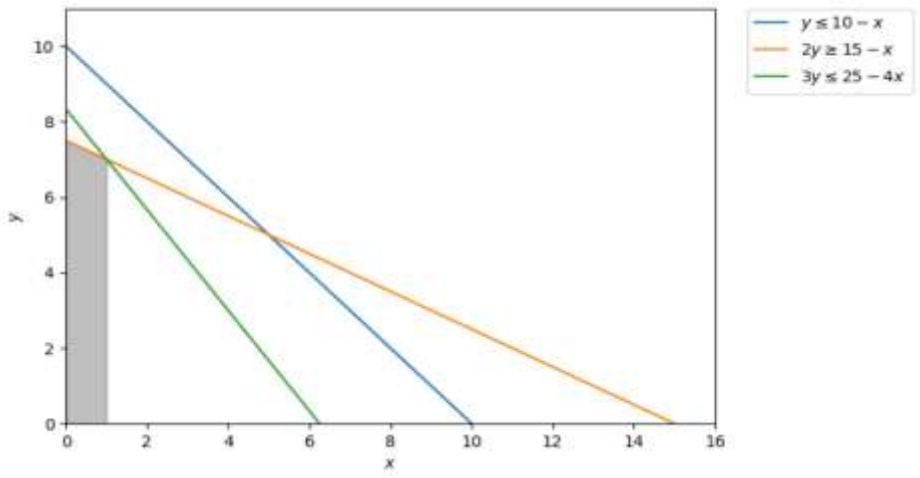
```
plt.plot(x1, y2, label=r'$y\leq10-x$')
plt.plot(x1, y3, label=r'$2y\geq 15 - x$')
plt.plot(x1, y4, label=r'$3y\leq 25-4x$')
plt.xlim((0, 16))
plt.ylim((0, 11))
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')

# Fill feasible region
y5 = np.minimum(y2, y4)
y6 = np.minimum(y2, y3)
plt.fill_between(x1, y1, y6, where=y5>y6, color='grey', alpha=0.5)
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.show()
```

Output

```
     con: array([], dtype=float64)
     fun: -7750.0
 message: 'Optimization terminated successfully.'
     nit: 2
   slack: array([2., 0., 0.])
  status: 0
 success: True
       x: array([1., 7.])
Total number of Notebooks that should be manufactured:  1000.0
Total number of Desktops that should be manufactured:  7000.0
Maximum profit that can be realized:  7750000.0
```

The result tells you that the maximal profit is 7,750,000 and corresponds to x_1 = 1000 (# of Notebooks) and x_2 = 7000 (# of Desktops).



PuLP Package: PuLP has a more convenient linear programming API than SciPy. You don’t have to mathematically modify your problem or use vectors and matrices. Everything is cleaner and less prone to errors. Its left for the readers to practice this Python package.

DATA WRANGLING USING PANDAS

Pandas package is for data extraction and preparation. Pandas is a very popular library that provides high-level data structures which are simple to use as well as intuitive. It has many inbuilt methods for grouping, combining data and filtering as well as performing time series analysis. Pandas can easily fetch data from different sources like SQL databases, CSV, Excel, JSON files and manipulate the data to perform operations on it.

NOTE: ALL THE DATASET USED IN THIS BOOK CAN BE DOWNLOADED FROM GITHUB LOCATION:
HTTPS://GITHUB.COM/SWAPNILSAURAV/THEATOZOFDATASCIENCEBOOK

Pandas deals with the following three data structures −

- Series
- DataFrame

Data Structure	Dimensions	Descriptions
Series	1	1D labeled homogeneous array, size immutable
DataFrame	2	General 2D labeled, size-mutable tabular structure

These data structures are built on top of Numpy array, which means they are fast. The best way to think of these data structures is that the higher dimensional data structure is a container of its lower dimensional data structure. For example, DataFrame is a container of Series. Panel was a container of DataFrame for 3D labeled data but now it has been removed. The recommended way to represent 3-D data are with a MultiIndex on a DataFrame.

SERIES

Series is a one-dimensional array like structure with homogeneous data. For example, the ollowing series is a collection of integers:

44	**33**	**66**	**88**	**77**	**99**	**55**

PANDAS.SERIES

pandas.Series: A pandas Series can be created using the following constructor –

```
pandas.Series( data, index, dtype, copy)
```

The parameters of the constructor are as follows –

1	**data: data takes various forms like ndarray, list, constants**
2	**index: Index values must be unique and hashable, same length as data. Default np.arrange(n) if no index is passed**
3	**dtype: dtype is for data type. If None, data type will be Object**
4	**copy: Copy data. Default False**

Example: Creating empty series

```
import pandas as pd
s = pd.Series(dtype= "float64")
print(s)
```

Example: Create a Series from ndarray

```
#Example 2: Create a Series from ndarray
import pandas as pd
import numpy as np
data = np.array(['a','b','c','d'])
s = pd.Series(data)
```

```
print(s)
```

We did not pass any index, so by default, it assigned the indexes ranging from 0 to len(data)-1

Example

```
import pandas as pd
import numpy as np
data = np.array(['a','b','c','d'])
s = pd.Series(data,index=[30,33,36,39])
print(s)
```

We passed the index values here. Now we can see the customized indexed values in the output.

Example: More examples of creating Series:

```
import pandas as pd
import numpy as np
#Create a Series from dict
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data)
print(s)
#Create a Series from Scalar
s = pd.Series(5, index=[0, 1, 2, 3])
print(s)
```

Note: If data is a scalar value, an index must be provided. The value will be repeated to match the length of index.

Example: Accessing Data from Series with Position

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
# Retrieve the first element, which is stored at zeroth position.
print(s[0])
#retrieve the first four element
print(s[:4])
#retrieve the last four element
print(s[-4:])
```

Refer the chapter on Strings to understand more on indexing. Strings are part of Basic Programming concept and has been covered in the book “Learn and Practice Python Programming by Swapnil Saurav”.

Example: Retrieve Data Using Label (Index)

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
```

```
#retrieve a single element
print(s['a'])
#retrieve multiple elements
print(s[['a','c','d']])
```

DATAFRAME

DataFrame is a two-dimensional array with heterogeneous data. For example,

ROLLNO	NAME	BRANCH	PHONE	EMAIL
45	Ajay	CSE	9945689651	ajay.cse45@example.org
46	Sachin	ECE	7896523459	sachin.ece46@ example.org
48	Kapil	CE	9009876456	kapil.ce48@ example.org
49	Saurabh	CHE	9087136745	saurabh.che49@ example.org

The data is represented in rows and columns. Each column represents an attribute and each row represents a person (specific record).

PANDAS.DATAFRAME

pandas.DataFrame: A pandas DataFrame can be created using the following constructor –

```
pandas.DataFrame(data, index, columns, dtype, copy)
```

The parameters of the constructor are as follows:

data	**data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame.**
index	**For the row labels, the Index to be used for the resulting frame is Optional Default np.arrange(n) if no index is passed.**
columns	**For column labels, the optional default syntax is - np.arrange(n). This is only true if no index is passed.**
dtype	**Data type of each column.**
copy	**This command (or whatever it is) is used for copying of data, if the default is False.**

Example: Working with DataFrames

```
#import the pandas library and aliasing as pd
import pandas as pd
#Create an Empty DataFrame
df = pd.DataFrame()
print(df)
```

```
#Create a DataFrame from Lists
data = [1,2,3,4,5]
df = pd.DataFrame(data)
print(df)
#Create a DataFrame from 2D Lists
data = [['Ajay',40],['Baran',52],['Chandan',33]]
df = pd.DataFrame(data,columns=['Name','Avg'])
print(df)
# with float value
data = [['Ajay',40],['Baran',52],['Chandan',33]]
df = pd.DataFrame(data,columns=['Name','Avg'],dtype=float)
print(df)
#Create a DataFrame from Dict of ndarrays / Lists
data = {'Name':['Ram', 'Jad', 'Mahi', 'Sachin'],'Age':[28,34,29,42]}
df = pd.DataFrame(data)
print(df)

#Create a DataFrame from List of Dicts
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data)
print(df)
#Note: NaN (Not a Number) is appended in missing areas.
#by passing a list of dictionaries and the row indices
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data, index=['first', 'second'])
print(df)

#with a list of dictionaries, row indices, and column indices.
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]

#With two column indices, values same as dictionary keys
df1 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b'])

#With two column indices with one index with other name
df2 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b1'])
print(df1)
print(df2)
#Note: df2 DataFrame is created with a column index other than the dictionary key;
thus, appended the NaN's in place. Whereas, df1 is created with column indices same
as dictionary keys, so NaN's appended.

#Create a DataFrame from Dict of Series
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print(df)
```

CREATING DATAFRAMES FROM VARIOUS DATA

Let's cover some additional topics like creating a DataFrame object from Numpy arrays, from list of tuples, from dictionary, etc.

```
import pandas as pd
```

```
import numpy as np

#Form a Dataframe
data = {
    'apples': [25000, 15000, 18000, 21000],
    'oranges': [80000, 34000, 12000, 12000]
}
#pass it to the pandas DataFrame constructor:
production = pd.DataFrame(data)
#Each (key, value) item in data corresponds to a column in the resulting
DataFrame.
print(production)

#The Index of this DataFrame was given to us on creation as the numbers
0-3,
# but we could also create our own when we initialize the DataFrame.
purchases = pd.DataFrame(data, index=['Jan', 'Feb', 'Mar', 'Apr'])
print(production)
#So now we could locate a production details by using month:
print(purchases.loc['Mar'])

#Reading from a csv file
df2 = pd.read_csv('employee_file.csv')
#CSVs don't have indexes like our DataFrames, so
#we will designate the index_col =0
df = pd.read_csv('employee_file.csv', index_col=0)
print(df2)

#Reading data from JSON
df3 = pd.read_json('person.json')
print(df3)

#Connecting to Database (sqllite)
import sqlite3
con = sqlite3.connect("database.db")
df4 = pd.read_sql_query("SELECT * FROM Employee", con)
print(df4)

#Create a DataFrame from a dictionary, containing two columns
df = pd.DataFrame({'RollNo': [101, 102, 103], 'Names': ['Rahul',
'Rohit', 'Ramesh']})
print(df)
#Pandas orders columns alphabetically as dict are not ordered.
# To specify the order, use the columns parameter.
df = pd.DataFrame({'RollNo': [101, 102, 103], 'Names': ['Rahul',
'Rohit', 'Ramesh']},
 columns=['Names', 'RollNo'])
print(df)
```

Slicing and Dicing in a Dataframe

In the programs put in this section, we will see how to read a subset of data from a dataframe and also how to modify the data.

```
import pandas as pd
import numpy as np

#Output Names columns come before RollNo
#Create a DataFrame of random numbers:
# Set the seed for a reproducible sample
np.random.seed(0)
df = pd.DataFrame(np.random.randn(4, 3), columns=list('XYZ'))
print(df)
#Create a DataFrame with integers:
df =
pd.DataFrame(np.arange(42).reshape(7,6),columns=list('ABCDEF'))
print(df)

# iloc[row slicing, column slicing]
#iloc has 2 parts before , represents the rows and
#after comma represents the columns
#Slicing is similar to String
print("1: \n",df.iloc[0:3, 1:4])

# Select all columns for rows of index values 0 and 2
print("2: \n",df.loc[[0, 2], :])

# Select first row of columns B,C,D
print("3: \n",df.loc[0, ['B', 'C', 'D']])

# Select first, third and fifth rows of all columns
print("4: \n",df.loc[[0, 2, 4], :])

#Include nan across second row all columns
 # in column 0, set elements with indices 1,2 ... to NaN
print("NaN:")
df.iloc[[1,2],0] = np.nan
# in column 1, set elements with indices 0,4, to np.NaT (Not a Time)
df.iloc[[0,4],1] = pd.NaT
# in column 2, set elements with index from 0 to 3 to 'nan'
df.iloc[0:3,2] = 'nan'
# in column 5, set all elements to None
df.iloc[:,5] = None
# in row 5, set all elements to NaN
df.iloc[5,:] = np.nan
print(df)
```

Output is shown on next page: shown in 3 columns for better readability

```
          X         Y         Z
0  1.764052  0.400157  0.978738
1  2.240893  1.867558 -0.977278
2  0.950088 -0.151357 -0.103219
3  0.410599  0.144044  1.454274
    A   B   C   D   E   F
0   0   1   2   3   4   5
1   6   7   8   9  10  11
2  12  13  14  15  16  17
3  18  19  20  21  22  23
4  24  25  26  27  28  29
5  30  31  32  33  34  35
6  36  37  38  39  40  41
```

```
1:
   B  C  D
0  1  2  3
1  7  8  9
2 13 14 15
2:
    A   B   C   D   E   F
0   0   1   2   3   4   5
2  12  13  14  15  16  17
3:
 B    1
C    2
D    3
Name: 0, dtype: int32
4:
    A   B   C   D   E   F
0   0   1   2   3   4   5
2  12  13  14  15  16  17
4  24  25  26  27  28  29
```

```
NaN:
     A    B    C     D     E     F
0  0.0  NaT  nan   3.0   4.0  None
1  NaN    7  nan   9.0  10.0  None
2  NaN   13  nan  15.0  16.0  None
3 18.0   19   20  21.0  22.0  None
4 24.0  NaT   26  27.0  28.0  None
5  NaN  NaN  NaN   NaN   NaN   NaN
6 36.0   37   38  39.0  40.0  None
```

Creating Dataframe with Dates

```
import pandas as pd
import numpy as np
np.random.seed(0)
# create an array of 5 dates starting at '2020-08-15', one per minute
rng = pd.date_range('2020-08-15', periods=5, freq='T')
df = pd.DataFrame({ 'Date': rng, 'Val': np.random.randn(len(rng)) })
print (df)
# create an array of 5 dates starting at '2020-08-15', one per day
rng = pd.date_range('2020-08-15', periods=5, freq='D')
df = pd.DataFrame({ 'Date': rng, 'Val' : np.random.randn(len(rng))})
print (df)
# create an array of 5 dates starting at '2020-08-15', one every 3 years
rng = pd.date_range('2020-08-15', periods=5, freq='3A')
df = pd.DataFrame({ 'Date': rng, 'Val' : np.random.randn(len(rng))})
print (df)
np.random.seed(0)
rng = pd.date_range('2020-08-15', periods=5, freq='T')
df = pd.DataFrame({ 'Val' : np.random.randn(len(rng)) }, index=rng)
```

Output (presented in 3 columns)

```
                 Date       Val
0 2020-08-15 00:00:00  1.764052
1 2020-08-15 00:01:00  0.400157
2 2020-08-15 00:02:00  0.978738
3 2020-08-15 00:03:00  2.240893
4 2020-08-15 00:04:00  1.867558
```

```
                 Date       Val
0 2020-08-15 -0.977278
1 2020-08-16  0.950088
2 2020-08-17 -0.151357
3 2020-08-18 -0.103219
4 2020-08-19  0.410599
```

```
                 Date       Val
0 2020-12-31  0.144044
1 2023-12-31  1.454274
2 2026-12-31  0.761038
3 2029-12-31  0.121675
4 2032-12-31  0.443863
```

COLUMN SELECTION, ADDITION, AND DELETION IN A DATAFRAME

Example: Let us now understand column selection and column addition through examples.

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
      'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
#Column value selection
print( df ['one'])

# Adding a new column to an existing DataFrame object with column label by passing
new series

print ("Adding a new column by passing as Series:")
df['three']=pd.Series([10,20,30],index=['a','b','c'])
print(df)

print ("Adding a new column using the existing columns in DataFrame:")
df['four']=df['one']+df['three']

print(df)
```

Explanation: 4th column is added by adding of the values from column “one” and “three”.

Example: Let us now understand column deletion from a data frame through examples.

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']),
     'three' : pd.Series([10,20,30], index=['a','b','c'])}

df = pd.DataFrame(d)
print ("Our dataframe is:")
print(df)

# using del function
print ("Deleting the first column using DEL function:")
del df['one']
print(df)

# using pop function
print ("Deleting another column using POP function:")
df.pop('two')
print(df)
```

ROW SELECTION, ADDITION, AND DELETION IN A DATAFRAME

Example: Let us now understand row selection and row addition through examples.

```
import pandas as pd
```

```
#Appending a row by a single column value:
df3 = pd.DataFrame(columns=['Name', 'Location', 'Phone'])
df3.loc[0, 'Name'] = "Javagal"
print(df3)

#Appending a row, given list of values:
df3.loc[1] = ["Anil", "Bangalore", 12345]
print(df3)

#Appending a row given a dictionary:
df3.loc[2] = {'Name': "Romesh", 'Location': "Mumbai", 'Phone': 12349}
print(df3)
#Overwrite
df3.loc[2] = {'Name': "Ramesh", 'Location': "Mumbai", 'Phone': 12349}
print(df3)

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print(df)
#Rows can be selected by passing row label to a loc function
print(df.loc['b'])

#Slice Rows : Multiple rows can be selected using ' : ' operator.
print(df[2:4])

print("Add new rows to a DataFrame using the append function")
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])
df1 = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])

df1 = df1.append(df2, sort=True)
print(df1)
```

In this above example, two rows were dropped because those two contain the same label 0.

Boolean indexing of dataframes

What if the index is Boolean? Let's look at the some examples to play with Boolean index using .loc and .iloc

```
import pandas as pd
#Index represents if its a rainbow color
df = pd.DataFrame({"color": ['red', 'blue', 'orange', 'green',
                             'yellow','brown','black','white',
                             'indigo','violet','Pink','Purple'],
                   "name": ["Rose", "Sky","Orange","Tree",
                            "Ball","Bread","Bird","Milk",
                            "Car","Book","Dress","Cake"]},
 index=[True,True,True,True,
        True,False, False, False,
```

```
        True, True, False, False])

print("Dataframe: \n",df)
print("Colors of Rainbow: \n",df.loc[True])
print("Non Rainbow Colors:\n",df.loc[False])
print("First Element:\n",df.iloc[0])

#Accessing values
sample_size = df['name'] == 'Rose'
print("Details of Data with Name as Rose: \n",df[sample_size])
```

Output: For better visibility, output is broken into 3 columns

```
Dataframe:
        color    name
True      red    Rose
True     blue     Sky
True   orange  Orange
True    green    Tree
True   yellow    Ball
False   brown   Bread
False   black    Bird
False   white    Milk
True   indigo     Car
True   violet    Book
False    Pink   Dress
False  Purple    Cake
Colors of Rainbow:
        color    name
True      red    Rose
True     blue     Sky
True   orange  Orange
True    green    Tree
True   yellow    Ball
True   indigo     Car
True   violet    Book
Non Rainbow Colors:
        color    name
False   brown   Bread
False   black    Bird
False   white    Milk
False    Pink   Dress
False  Purple    Cake
First Element:
 color     red
name      Rose
Name: True, dtype: object
Details of Data with Name as Rose:
      color  name
True    red  Rose
```

Query data using Pandas

We can use the syntax below when querying data by criteria from a DataFrame. Experiment with selecting various subsets of the “Playerss” data.

- Equals: ==
- Not equals: !=
- Greater than, less than: > or <
- Greater than or equal to >=
- Less than or equal to <=

```
import pandas as pd

players = {"name": pd.Series(["Sachin", "Anil", "Zaheer", "Sourav"]),
           "primary_type":
pd.Series(["Batsman","Bowler","Bowler","Batsman"]),
           "runs": pd.Series([25000,8000,7000,19000]),
           "wickets": pd.Series([40,700,600,150])}
players_df = pd.DataFrame(players)
players_df = players_df.set_index(players_df["name"])
print ("Our dataframe is:")
print(players_df)
#1. Checking the value
print("\n1. Get the data of the players in the dataframe:")
print(players_df[players_df['name'].isin(["Sachin", "Kaif",
"Zaheer"])])
#1. Explanation: since Kaif is not in the main data,
# his records are not displayed
```

```
#2. Average wickets taken by the players
print("\n2. Average wickets taken: ",end=" ")
print(players_df['wickets'].mean())

#3. Average of all numeric columns based on type of the players
print("\n3. Average value for Player type:\n")
print(players_df.groupby('primary_type').mean())

#4. Get the sums of the runs scored by all the players
print("\n4. Total runs scored:",end=" ")
print(players_df["runs"].sum())
```

Output: set in 3 columns

```
Our dataframe is:
         name primary_type   runs  wickets
name
Sachin  Sachin      Batsman  25000       40
Anil      Anil       Bowler   8000      700
Zaheer  Zaheer       Bowler   7000      600
Sourav  Sourav      Batsman  19000      150

1. Get the data of the players in the dataframe:
         name primary_type   runs  wickets
name
Sachin  Sachin      Batsman  25000       40
Zaheer  Zaheer       Bowler   7000      600

2. Average wickets taken:  372.5

3. Average value for Player type:
              runs  wickets
primary_type
Batsman      22000       95
Bowler        7500      650

4. Total runs scored: 55000
```

DATA MANIPULATION USING DROP()

We can use Drop function of Pandas to drop/delete Rows And Columns In pandas Dataframe. Let's taken an example of following dataframe:

```
import pandas as pd
data1={
    "Name":
["Joy","Pancham","Sabuj","Vivek","Shinjith","Vinay","Priyvrat"],
    "Marks": [95,89,65,78,99,78,87],
    "Branch": ["CSE","ECE","TEL","MECH","CIVIL","EEE","ROBOTICS"]
}
df1 = pd.DataFrame(data1)
#Let it use Default Index of 0,1,...
print("Dataframe 1 with default Index: \n",df1)
df2 = pd.DataFrame(data1, index = [101,102,103,104,105,106,107])
#Let it use Default Index of 0,1,...
print("Dataframe 2 with Roll No. as Index: \n",df2)
```

Output:

```
Dataframe 1 with default Index:
        Name  Marks    Branch
0        Joy     95       CSE
1    Pancham     89       ECE
2      Sabuj     65       TEL
3      Vivek     78      MECH
4   Shinjith     99     CIVIL
5      Vinay     78       EEE
6   Priyvrat     87  ROBOTICS
Dataframe 2 with Roll No. as Index:
          Name  Marks    Branch
101       Joy     95       CSE
102   Pancham     89       ECE
103     Sabuj     65       TEL
104     Vivek     78      MECH
105  Shinjith     99     CIVIL
106     Vinay     78       EEE
107  Priyvrat     87  ROBOTICS
```

Drop an observation (row)

Axis is variable in Pandas which decides if we want to work with Rows or Columns. Axis = 0 indicate it's a row and axis is 1 for columns. Default value of axis is 0.

Drop rows with index 1 and 2 in firstdataframe and 105 and 107 from the second dataframe: so the output should look like:

```
After Drop Dataframe 1 with default Index:
        Name  Marks    Branch
0        Joy     95       CSE
3      Vivek     78      MECH
4   Shinjith     99     CIVIL
5      Vinay     78       EEE
6   Priyvrat     87  ROBOTICS
After Drop Dataframe 2 with Roll No. as Index:
         Name  Marks Branch
101       Joy     95    CSE
102   Pancham     89    ECE
103     Sabuj     65    TEL
104     Vivek     78   MECH
106     Vinay     78    EEE
```

Python Code:

```
#Deleting a row:
df1 = df1.drop([1,2],axis=0)
print("After Drop Dataframe 1 with default Index: \n",df1)
#We can ignore axis value as default value of axis
#is zero anyway
df2 = df2.drop([105,107])
print("After Drop Dataframe 2 with Roll No. as Index: \n",df2)
```

Drop a variable (column)

Note: axis=1 denotes that we are referring to a column, not a row

From dataframe delete column Branch. Output should look something like:

```
After Drop Dataframe 1 with default Index:
        Name  Marks
0        Joy     95
3      Vivek     78
4   Shinjith     99
5      Vinay     78
6   Priyvrat     87
```

Program:

```
#Deleting a variable (column)
#Delete column Branch from df1
df1 = df1.drop("Branch", axis=1)
```

```
print("After Drop Dataframe 1 with default Index: \n",df1)
```

More Programs:

```
#Delete row from DF2 where name is Priyvrat
df3 = df2[df2.Name != 'Priyvrat']
#Name= N is cap, Python is case sensitive
print("DF2 after dropping name Priyvrat: \n", df3)

#Drop a row by row number (in this case, row 4)
#Note that Pandas uses zero based numbering,
# so 0 is the first row, 1 is the second row, etc
df4 = df1.drop(df1.index[3])
print("DF1 after dropping index 3: \n", df4)

#dropping relative to the end of the DF.
#Drop second last row in df1
df5 = df1.drop(df1.index[-2])
print("DF1 after dropping second last index: \n", df5)
```

Output:

```
DF2 after dropping name Priyvrat:
         Name  Marks Branch
101       Joy     95    CSE
102  Pancham      89    ECE
103     Sabuj     65    TEL
104     Vivek     78   MECH
106     Vinay     78    EEE
DF1 after dropping index 3:
        Name  Marks
0        Joy     95
3      Vivek     78
4  Shinjith      99
6  Priyvrat      87
DF1 after dropping second last index:
        Name  Marks
0        Joy     95
3      Vivek     78
4  Shinjith      99
6  Priyvrat      87
```

MERGE THE DATASET

The Pandas merge() command takes the left and right dataframes, matches rows based on the “on” columns, and performs different types of merges – left, right, etc.

PANDAS.MERGE()

Dataframe merge can be done using –

> *pandas.merge(left_data, right_data, left_on, right_on, how)*

The parameters of the constructor are as follows:

left_data	**“left” dataframe**
right_data	**“right” dataframe**
left_on	**left_on specifies merge column names(s) in left dataframe**
right_on	**right_on specifies merge column names(s) in right dataframe**
how	**how can be one of: left, right, inner, outer**

Example 11: Merge dataframes

In this example, we have taken 3 dataframes which can be downloaded from the github.

- user_usage.csv – A first dataset containing users monthly mobile usage statistics
- user_device.csv – A second dataset containing details of an individual “use” of the system, with dates and device information.
- android_devices.csv – A third dataset with device and manufacturer data, which lists all Android devices and their model code, obtained from Google.

Step 1: Read the csv files

```
import pandas as pd
#Read the 3 csv files
d1 = pd.read_csv(r'https://raw.githubusercontent.com/swapnilsaurav/Dataset/master/user_
#showing monthly mobile usage statistics for a subset of users.
print(d1.head(5))
d2 = pd.read_csv(r'https://raw.githubusercontent.com/swapnilsaurav/Dataset/master/user_d
#giving the device and OS version for individual
print(d2.head(5))
d3 =
pd.read_csv(r'https://raw.githubusercontent.com/swapnilsaurav/Dataset/master/android_dev
```

```
#containing all Android devices with manufacturer and model details
print(d3.head(5))
```

There are linking attributes between the sample datasets that are important to note – “use_id” is shared between the user_usage and user_device, and the “device” column of user_device and “Model” column of the devices dataset contain common codes. We want to form a single dataframe with columns for user usage figures (calls per month, sms per month etc) and also columns with device information (model, manufacturer, etc). We will need to “merge” (or “join”) our sample datasets together into one single dataset for analysis.

Let’s see how we can correctly add the “device” and “platform” columns to the user_usage dataframe using the Pandas Merge command.

```
#add the "device" and "platform" columns to the user_usage dataframe
result = pd.merge(d1,
                  d2[['use_id', 'platform', 'device']],
                  on='use_id')
result.head()
```

Merging requires a “left” dataset, a “right” dataset, and a common column to merge “on”. Pandas merge() defaults to an “inner” merge operation. Let’s now analyze the return dataframes size. First, let’s look at the sizes or shapes of our inputs and outputs to the merge command:

```
#Analyze the dimensions of the dataframes
print("user_usage Dimensions: ",d1.shape)
print("user_device Dimensions: ", d2[['use_id', 'platform', 'device']].shape)
print("result Dimensions: ",result.shape)
```

Output:

```
user_usage Dimensions:  (240, 4)
user_device Dimensions:  (272, 3)
result Dimensions:  (159, 6)
```

Why is the result a different size to both the original dataframes?

An inner merge, (or inner join) keeps only the common values in both the left and right dataframes for the result. In our example above, only the rows that contain use_id values that are common between user_usage and user_device remain in the result dataset. We can validate this by looking at how many values are common:

```
#how many values are common
print(d1['use_id'].isin(d2['use_id']).value_counts())
```

Output:

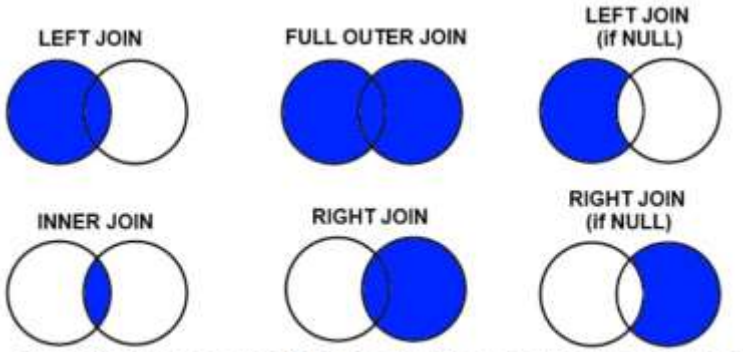
```
True     159
False     81
Name: use_id, dtype: int64
```

This explains why only 159 rows are in the result dataframe.

Other Merge type

Apart from Inner Merge/Inner Join, other types of Merge are:

- Left Merge / Left outer join – Keep every row in the left dataframe. For missing values of the "on" variable in the right dataframe, add empty / NaN values in the result.
- Right Merge / Right outer join –Keep every row in the right dataframe. For missing values of the "on" variable in the left column, add empty / NaN values in the result.
- Outer Merge / Full outer join – A full outer join returns all the rows from the left dataframe, all the rows from the right dataframe, and matches up rows where possible, with NaNs elsewhere.



Merge/Join types as used in Pandas, R, SQL, and other data-orientated languages and libraries. Source: Stack Overflow.

You can change the merge to a left-merge with the "how" parameter to your merge command.

```
#You can change the merge to a left-merge with the "how" parameter to your merge command.
result = pd.merge(d1,
                  d2[['use_id', 'platform', 'device']],
                  on='use_id',
                  how='left',
                 indicator =True)
result.head()
print("user_usage Dimensions: ",d1.shape)
print("result Dimensions: ",result.shape)
print("There are {} missing values in the
result.".format(result['device'].isnull().sum()))
```

Output:

```
user_usage Dimensions:  (240, 4)
result Dimensions:  (240, 7)
There are 81 missing values in the result.
```

Indicator: To assist with the identification of where rows originate from, Pandas provides an "indicator" parameter that can be used with the merge function which creates an additional column called "_merge" in the output that labels the original source for each row.

Practice Problems

1. Extend the examples given in this section with right merge / right join and outer merge / full outer join

Using left_on and right_on to merge with different column names

Now let's add the third dataframe. We will redo the first merge to get back to inner merge and then merge devices dataframe.

```
# Add the platform and device to the user usage
result = pd.merge(d1,
                  d2[['use_id', 'platform', 'device']],
                  on='use_id',
                  how='left')

# Merge based on the "device" column in result, match the "Model" column in devices
(d3).
d3.rename(columns={"Retail Branding": "manufacturer"}, inplace=True)
result = pd.merge(result,
                   d3[['manufacturer', 'Model']],
                   left_on='device',
                   right_on='Model',
                   how='left')
print(result.head())
```

The columns used in a merge operator do not need to be named the same in both the left and right dataframe. In the second merge above, note that the device ID is called "device" in the left dataframe, and called "Model" in the right dataframe. Different column names are specified for merges in Pandas using the "left_on" and "right_on" parameters, instead of using only the "on" parameter.

Calculating statistics based on device

We can use the data aggregation functionality of Pandas to quickly work out the mean usage for users based on device manufacturer.

```
#Calculating statistics based on device
print(result.groupby("manufacturer").agg({
        "outgoing_mins_per_month": "mean",
        "outgoing_sms_per_month": "mean",
        "monthly_mb": "mean",
        "use_id": "count"
    }))

#Get Memoery usage
result.info(memory_usage='deep')
```

Describe

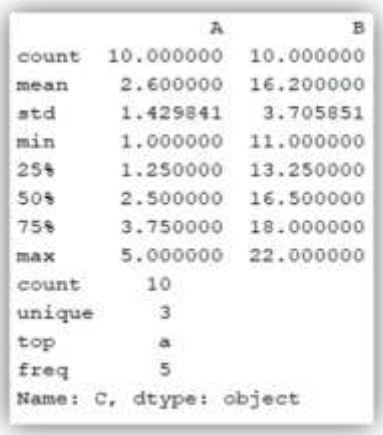
Descriptive statistics (mean, standard deviation, number of observations, minimum, maximum, and quartiles) of numerical columns can be calculated using the .describe() method, which returns a pandas dataframe of descriptive statistics.

```
import pandas as pd

data = pd.DataFrame({'A': [1, 2, 1, 4, 3, 5, 2, 3, 4, 1],
                     'B': [12, 14, 11, 16, 18, 18, 22, 13, 21, 17],
                     'C': ['a', 'a', 'b', 'a', 'b', 'c', 'b', 'a', 'b',
'a']})
print(data.describe())


#Note that since C is not a numerical column, it is excluded from the
output.
print(data['C'].describe())
```

Output:



	A	B
count	10.000000	10.000000
mean	2.600000	16.200000
std	1.429841	3.705851
min	1.000000	11.000000
25%	1.250000	13.250000
50%	2.500000	16.500000
75%	3.750000	18.000000
max	5.000000	22.000000
count	10	
unique	3	
top	a	
freq	5	

Name: C, dtype: object