**DATA SCIENCE USING PYTHON**

**MODULE1:**

**CORE & ADVANCE PYTHON**

**Data Types in Python**

* **Numbers**
  + Integer
  + Float
  + Complex
* **String**
* **Tuple**
* **Lists**
* **Dictionaries**
* **Sets**

***NUMBER DATA TYPE EXPLAINED***

Examples of Number Data Types

X=10 *#Example of integer*Y=12.7 *#Example of floating point number*

Z=2+3j #Example of complex number

**Arithmetic Operations on Numbers:**

Arithmetic Operators such as -

Addition, Subtraction, Multiplication, Exponent, Division, Floor Division, and Modulus explained with examples.

print(10+5) print(10-5) print(10\*5) print(10\*\*2) print(10/5) print(12//5) print(14%6)

**Comparison Operators**:

Comparison Operators such as ==,</LT,<=/LE,>/GT,>=/GE,NE/!= explained with examples.

x=10  
y=15  
print(x==y)   
print(x<y)   
print(x<=y)   
print(x>y)

print(x>=y)   
print(x!=y)

**Assignment Operations**:

Assignment Operator '='isexplained with examples.

Total=200  
Region='North'

Multiple Assignment Example1

x=y=z=150  
print(x)  
print(y)  
print(z)

Multiple Assignment Example2

x,y=100,200  
print(x)  
print(y)

**Logical Operators:**

Logical Operators such as AND, OR, NOTisexplained with examples.

print (1>4 and 5>4)

print (1>4 or 5>4)

print (not(4>5))

***STRING DATA TYPE EXPLAINED***

Str1="satyamev jayte"

print (Str1)

str1="shramamev jayte" print (str1)

**Operations on Strings**

**1. The Subscript Operator - Used to access the elements of string**

**2. Slicing for Substrings - Used to access the elements of string**

print(str1[2:6])   
print(str1[2:])   
print(str1[:5])   
print(str1[:])

**3. String Methods - COUNT, FIND**

str2="The Avengers"  
print(str1.count('T'))   
print(str1.count('e'))  
str3="peace begins with a smile"  
print(str3.find('with'))

**4. String Case Methods - UPPER, LOWER, CAPITALIZE, TITLE, SWAPCASE**

str4="manoj jangra"  
print(str4.upper())  
print(str4.lower())  
print(str4.capitalize())

print(str4.title

print(str4.swapcase())

**5. String Strip Methods**

str5="manoj jangram"  
print(str5.rstrip("m"))  
str5="manoj jangra#"  
print(str4.rstrip('#'))  
str5=' manoj kumar '  
print(str5.rstrip())

**6. String Split Methods**

str6='27-12-2009'  
print(str6.split('-',1)) #1st part split  
print(str6.split('-',2)) #1st, 2nd part split  
print(str6.split('-',3)) #1st, 2nd part & 3rd part split  
str7="mohit raj"  
print(str7.split())

**7. String Justify Methods - LJUST, RJUST, CENTER, ZFILL, REPLACE, JOIN**

print(str7.ljust(15,'@')) #Left justify  
print(str7.rjust(15,'#')) #Right justify  
print(str7.center(16,'\*')) #Centrally justified  
account\_no='1810037305'  
print(account\_no.zfill(20)) #Filling a string with characters  
str8="manoj kumar"  
print(str8.replace("kumar","jangra"))#Repacing a part of string with other part  
name=['mohit','raj']  
print(" ".join(name))#Joining parts of strings

***TUPLE DATA TYPE EXPLAINED***

tup1=()

Tup2=(2,1,3,”manoj”,10,”hina”)

**Indexing Tuple**

Print(tup2[0])

Print(tup2[2])

**Slicing of tuple**

Print(tup2[2,5])

Print(tup2[2:])

Print(tup2[:5])

Print(tup2[-3,-2])

**Unpacking the items of tuples**

tup3=(1,2,3)

A,b,c=tup3

Print(a,b,c)

**Tuple functions – len(), max(), min()**

Print(len(tup3)): 3

Print(max(tup3)): 3

Print(min(tup3)):1

**Operations on tuples**

**(+) – Addition operator**

Print(tup3+tup4)

**(\*) – Multiplication operator**

Print(tup\*3): 1,2,3,1,2,3,1,2,3

***LIST DATA TYPE EXPLAINED***

**Creating a list**

Lis1=[] #List with null values

Print(lis1)

**Creating a list with values**

Lis2=[‘banana’,’mango’,’apple’,’carrot’]

Print(lis2)

**Unpacking list values**

Lis3=[2,3,5]

X,Y,Z=Lis3

Print(X,Y,Z)

List operations

Print(lis3[0])

Print(lis3[3])

**Slicing of List**

Print(lis3[2,6])

Print(lis3[2:])

Print(lis[:4])

Print(lis3[1:13:3])

**Updating the list**

Avengers=[‘thor’,’hulk’,’captain’]

Avengers=[2]

Avengers[2]=’captain-america’

**Deleting values from a list**

del list1[1]  
*print(list1)*del list1[2:5]  
*print(list1)*del list1  
*print(list1)*

**List Operations**

**Addition of Python lists**

veg\_list=['carrot','ladyfinger','reddish']  
fru\_list=['mango','banana']  
fin\_list=veg\_list+fru\_list

print(fin\_list)

**Multiplication of lists**

av=['vision','sam']  
av\_double=av\*2  
print(av\_double)

**List functions**

**len()** –

print(len(fin\_list))

**max ()** –

print(fin\_list)

**min()** –

print(fin\_list)

**sorted () –**

tup1=(1,4,3)  
tup\_2\_list=list(tup1)  
print(tup\_2\_list)  
print(sorted(tup\_2\_list))

**List methods**

**append () –**

veg\_list.append('spinach')  
veg\_list.append('corriendor')  
print(veg\_list)

**extend() –**

lis1=[10,20,40,30]

lis2=[40,60]

lis1.extend(lis2)

print(lis1)

**count () –**

print(lis1.count(10))

**index () –**

print(lis1.index(20))

**insert()-**

veg\_list.insert(0,'zeera')  
print(veg\_list)

**remove() –**

veg\_list.remove('carrot')  
print(veg\_list)

**reverse() –**

veg\_list.reverse()  
print(veg\_list)

***DICTIONARY DATA TYPE EXPLAINED***

Empty\_dic={}

best\_team={'australia':'bradman','india':'shrikant','shrilanka':'jaisurya'}  
print(best\_team)

**Operations on the dictionary**

*Accessing the values of dictionary*

print(best\_team['shrilanka']

*Deleting an item from the dictionary*

del best\_team['india']  
print(best\_team)

*Updating the values of the dictionary*

best\_team['shrilanka']='sanath'  
print(best\_team)

*Adding an item to the dictionary*

best\_team['england']='batham'  
best\_team['pakistan']='miyandad'  
print(best\_team)

**Dictionary functions**

**len() –**

print(len(best\_team))

**str() –**

str(best\_team)  
print(best\_team)

**max() –**

max(best\_team)  
print(best\_team)

**min() –**

min(best\_team)  
print(best\_team)

**Dictionary Methods –**

new\_bestteam=best\_team.copy()  
print(new\_bestteam)

**get() –**

print(best\_team.get('india'))

**setdefault() –**

print(best\_team.setdefault('india'))

**keys() –**

port1 = {80: 'http', 18: None, 19: 'Unknown', 22: 'SSH', 23: 'Telnet'}  
print(port1.keys())

**values() –**

port1 = {80: 'http', 18: None, 19: 'Unknown', 22: 'SSH', 23: 'Telnet'}  
print(port1.values())

**update() –**

port1 = {80: 'http', 18: None, 19: 'Unknown', 22: 'SSH', 23: 'Telnet'}  
port2={25:'given', 30:'not taken'}  
port1.update(port2)  
print(port1)

**items() –**

port1.items()  
print(port1)

**clear() –**

port1.clear()  
print(port1)

**Dictionary with for loop**

port1 = {21: "FTP", 22 :"SSH", 23: "telnet", 80: "http"}  
for k,v in port1.items():  
 print (k," : ", v)

# Python Sets

A set is an unordered collection of items. Every set element is unique (no duplicates) and must be immutable (cannot be changed).

However, a set itself is mutable. We can add or remove items from it.

Sets can also be used to perform mathematical set operations like union, intersection, symmetric difference, etc.

## Creating Python Sets

A set is created by placing all the items (elements) inside curly braces {}, separated by comma, or by using the built-in set() function.

It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have mutable elements like lists, sets or dictionaries as its elements.

# Different types of sets in Python

# set of integers

my\_set = {1, 2, 3}

print(my\_set)

# set of mixed datatypes

my\_set = {1.0, "Hello", (1, 2, 3)}

print(my\_set)

**Examples-**

# set cannot have duplicates

# Output: {1, 2, 3, 4}

my\_set = {1, 2, 3, 4, 3, 2}

print(my\_set)

# we can make set from a list

# Output: {1, 2, 3}

my\_set = set([1, 2, 3, 2])

print(my\_set)

# set cannot have mutable items

# here [3, 4] is a mutable list

# this will cause an error.

my\_set = {1, 2, [3, 4]}

***CONTROL STATEMENTS EXPLAINED***

**The if statement**

The if statement is used to check a condition:

*if* the condition is true, we run a block of statements (called the *if-block*),

*else* we process another block of statements (called the *else-block*). The *else* clause is optional.

**Python if Statement Syntax**

if test expression:

statement(s)

**Example (save as if.py ):**

*# If the number is positive, we print an appropriate message*num = 3  
if num >0:  
print(num, "is a positive number.")  
print("This is always printed.")  
num = -1  
if num >0:  
print(num, "is a positive number.")  
print("This is also always printed.")

## Python if...else Statement

### Syntax of if...else

if test expression:

statement(s)

else:

body of else

**Example (save as if.py ):**

*# Program checks if the number is positive or negative  
# And displays an appropriate message*num = 3  
*# Try these two variations as well.  
# num = -5  
# num = 0*if num >= 0:  
print("Positive or Zero")  
else:  
print("Negative number")

**Python if...elif...else Statement**

*Syntax of if...elif...else*

if test expression:

Body of if

elif test expression:

Body of elif

else:

Body of else

**Example1 (save as if.py ):**

*# In this program,  
# we check if the number is positive or  
# negative or zero and  
# display an appropriate message*num = 3.4  
*# Try these two variations as well:  
# num = 0  
# num = -4.5*if num >0:  
print("Positive number")  
elif num == 0:  
print("Zero")  
else:  
print("Negative number")

**Example2 (save as if.py ):**

number = 23  
guess = int(input('Enter an integer : '))  
if guess == number:  
*# New block starts here*print('Congratulations, you guessed it.')  
print('(but you do not win any prizes!)')  
*# New block ends here*elif guess <number:  
*# Another block*print('No, it is a little higher than that')  
*# You can do whatever you want in a block ...*else:  
print('No, it is a little lower than that')  
*# you must have guessed > number to reach here*print('Done')  
*# This last statement always executed,  
# # after the if statement is executed. is*

.

**Python Nested if statements**

We can have a if...elif...else statement inside nother if...elif...elsestatement. This is called nesting in computer programming.

Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. This can get confusing, so must be avoided if we can.

**Example (save as if.py ):**

*#In this program, we input a number  
# check if the number is positive or  
# negative or zero and display  
# an appropriate message  
# This time we use nested if*num = float(input("Enter a number: "))  
if num >= 0:  
if num == 0:  
print("Zero")  
else:  
print("Positive number")  
else:  
print("Negative number")

**Python ‘for’loop**

## What is for loop in Python?

The for loop in Python is used to iterate over a sequence ([list](https://www.programiz.com/python-programming/list), [tuple](https://www.programiz.com/python-programming/tuple), [string](https://www.programiz.com/python-programming/string)) or other iterable objects. Iterating over a sequence is called traversal.

**Syntax of for Loop**

for val in sequence:

Body of for

**Example (save as if.py ):**

*# Program to find the sum of all numbers stored in a list  
# List of numbers*numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]  
*# variable to store the sum*sum = 0  
*# iterate over the list*for val in numbers:  
sum = sum+val  
*# Output: The sum is 48*print("The sum is", sum)

## The range() function

We can generate a sequence of numbers using range() function. Range(10) will generate numbers from 0 to 9 (10 numbers)

**Example1 (save as for.py ):**

for i in range(10):  
 print(i)  
else:  
 print('The for loop is over')

**Example2 (save as for.py ):**

for i in range(1, 10):  
 print(i)  
else:  
 print('The for loop is over')

**Example3 (save as for.py ):**

for i in range(1, 10, 2):  
 print(i)  
else:  
 print('The for loop is over')

**Example4 (save as for.py ):**

for i in range(10, 0, -1):  
 print(i)  
else:  
 print('The for loop is over')

**Example5 (save as for.py ):**

primes=[13,5,7,9,11]  
for primes in primes:  
print (primes)

## For loop with else

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts.

digits = [0, 1, 5]  
for i in digits:  
 print(i)  
else:  
 print("No items left.")

**The while Loop**

The while statement allows you to repeatedly execute a block of statements as long as a condition is true. A while statement is an example of what is called a *looping* statement. A while statement can have an optional else clause.

**Syntax of while Loop in Python**

while test\_expression:

Body of while

**Example1 (save as for.py ):**

*# Program to add natural  
# numbers upto  
# sum = 1+2+3+...+n  
# To take input from the user,  
# n = int(input("Enter n: "))*n = 10  
*# initialize sum and counter*sum = 0  
i = 1  
while i <= n:  
sum = sum + i  
 i = i+1 *# update counter  
# print the sum*print("The sum is", sum)

## While loop with else

**Example1 (save as for.py ):**

*# Example to illustrate  
# the use of else statement  
# with the while loop*counter = 0  
while counter <3:  
print("Inside loop")  
 counter = counter + 1  
else:  
print("Inside else")

**Python break statement**

The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

*# Use of break statement inside loop*for val in "string":  
if val == "i":  
break  
 print(val)  
print("The end")

**Python continue statement**

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

*# Program to show the use of continue statement inside loops*for val in "string":  
if val == "i":  
continue  
 print(val)  
print("The end")

**Python pass statement**

In Python programming, pass is a null statement. The difference between a comment and pass statement in Python is that, while the interpreter ignores a comment entirely, pass is not ignored.

However, nothing happens when pass is executed. It results into no operation (NOP).

**Syntax of pass**

Pass

*# pass is just a placeholder for  
# functionality to be added later.*sequence = {'p', 'a', 's', 's'}  
for val in sequence:  
pass

***FUNCTIONS EXPLAINED***

**How To Define Functions**

A function is a block of instructions that performs an action and, once defined, can be reused. Functions make code more modular, allowing

you to use the same code over and over again.

**Built-in Functions**

**print()**which will print an object to the terminal

**int()** which will convert a string or number data type to an integer data type

**len()**which returns the length of an object

Function names include parentheses and may include parameters.

**Example1:Built-in SQRT function**

from math import sqrt  
# Get value from the user  
num = float(input("Enter number: "))  
# Compute the square root  
root = sqrt(num)  
# Report result  
print("Square root of", num, "=", root)

**Example2:Built-in RANDOM function Randrange-**

Returns a pseudorandom integer value within a specified range.

Seed- Sets the random number seed.

from random import randrange, seed  
seed(23) # Set random number seed  
for i in range(0, 5): # Print 100 random numbers  
print(randrange(1, 1001)) # Range 1...1,000, inclusive  
print()

**Example3: Built-in RANDOM function with CHOICE**

**Choice-** Selects an element at random from a collection of elements.

from random import choice  
for i in range(10):  
print(choice(("one", "two", "three", "four", "five", "six","seven", "eight", "nine", "ten")))

**User Defined Functions**

**Example1: Function without Parameter**

def hello():  
print("This is 1st line\nThis is 2nd line")  
hello()

**Example 2: Function without Parameter**

def names():  
name = 'Manoj Jangra'

if name=='Manoj Jangra':  
print('Your name is:', name)  
else:  
print('Your name is not:', name)  
names()

**Example 3: Function with Keyword Parameter**

def names(name):  
if name=='Manoj Jangra':  
print('Your name is "Manoj Jangra"')  
else:  
print('Your name is not "Manoj Jangra"')  
names('kumar')

**Example 3: Function with Keyword Parameter**

def add\_numbers(x, y, z):  
a = x + y  
b = x + z  
c = y + z  
print(a, b, c)  
add\_numbers(1, 2, 3)

**Example 4: Function with Keyword Parameter**

def profile\_info(username, followers):  
print("Username: " + username)  
print("Followers: " + str(followers))

profile\_info('Manoj','Jangra')

**How To Use \*args**

The single-asterisk form of \*args can be used as a parameter to send a non-keyworded variable-length argument list to functions.

**Example1**

def multiply(x, y):  
print (x \* y)  
multiply(5, 4)

def multiply(x, y):  
print (x \* y)  
multiply(5, 4)  
multiply(5,4,6)

**Error message-**

C:\Python22\python.exe C:/Users/user/PycharmProjects/F1\_Project/Class1.py

Traceback (most recent call last):

File "C:/Users/user/PycharmProjects/F1\_Project/Class1.py", line 5, in ?

multiply(5,4,6)

TypeError: multiply() takes exactly 2 arguments (3 given)

20

**Solution:**

def multiply(\*args):  
z = 1  
for num in args:  
z \*= num  
print(z)  
multiply(4, 5)  
multiply(10, 9)  
multiply(2, 3, 4)  
multiply(3, 5, 10, 6)

# Lambda Functions

# A Lambda Function in Python programming is an anonymous function or a function without name.

# It is a small and restricted function having no more than one line.

# Just like a normal function, a Lambda function can have multiple arguments with one expression.

**Syntax**

The formal syntax to write a lambda function is as given below:

lambda p1, p2: expression

Where, p1 and p2 are the parameters which are passed to the lambda function. You can add as many or few parameters as you need.

### Example1

adder = lambda x, y: x + y

print (adder (1, 2))

### Example 2

#What a lambda returns

string='some kind of a useless lambda'

print(lambda string : print(string))

**Using Lambdas with Python Built-ins**

## lambdas in filter()

The filter function is used to select some particular elements from a sequence of elements. The sequence can be any iterator like lists, sets, tuples, etc.

The elements which will be selected is based on some pre-defined constraint. It takes 2 parameters:

* A function that defines the filtering constraint
* A sequence (any iterator like lists, tuples, etc.)

For example,

sequences = [10,2,8,7,5,4,3,11,0, 1]

filtered\_result = filter (lambda x: x > 4, sequences)

print(list(filtered\_result))

## lambdas in map()

The map function is used to apply a particular operation to every element in a sequence. Like filter(), it also takes 2 parameters:

* A function that defines the op to perform on the elements
* One or more sequences

**Example:**

sequences = [10,2,8,7,5,4,3,11,0, 1]

filtered\_result = map (lambda x: x\*x, sequences)

print(list(filtered\_result))

**lambdas in reduce()**

The reduce function, like map(), is used to apply an operation to every element in a sequence. However, it differs from the map in its working. These are the steps followed by the reduce() function to compute an output:

It also takes two parameters:

* A function that defines the operation to be performed
* A sequence (any iterator like lists, tuples, etc.)

**Example:**

from functools import reduce

sequences = [1,2,3,4,5]

product = reduce (lambda x, y: x\*y, sequences)

print(product)

# Python Errors and Built-in Exceptions

Errors or inaccuracies in a program are often called as bugs. The process of finding and removing errors is called debugging. Errors can be categorized into three major groups:

1. **Syntax errors 2. Runtime errors and 3. Logical errors**

**Syntax errors**

Python will find these kinds of errors when it tries to parse your program, and exit with an error message without running anything. Syntax errors are like spelling or grammar mistakes in a language like English.

**Example:**

print "Hello World!"

**Logical errors**

These are the most difficult type of error to find, because they will give unpredictable results and may crash your program. A lot of different things can happen if you have a logic error.

**Example:**

x = 3

y = 4

average = x + y / 2

print(average)

**Runtime errors**

If a program is free of syntax errors, it will be run by the Python interpreter. However, the program may exit if it encounters a runtime error – a problem that went undetected when the program was parsed, but is only revealed when the code is executed.

**Example:**

x=int(input("enter first number:"))

y=int(input("enter second number:"))

print("Division of X through Y is:",x/y)

**Some examples of Python Runtime errors −**

* division by zero
* performing an operation on incompatible types
* using an identifier which has not been defined
* accessing a list element, dictionary value or object attribute which doesn’t exist
* trying to access a file which doesn’t exist

When writing a program, we, more often than not, will encounter errors.

Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error.

***Errors occur at runtime are called exceptions.***

**What is Exception?**

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

**Exception Handling**

## Handling an exception

If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

### Syntax

Here is simple syntax of *try....except...else* blocks –

try:  
 <do something>  
except Exception:  
 <handle the error>

**Example1**

print("hello world")

try:

print(10/0)

except ZeroDivisionError as e:

print(10/1,e)

print('hey! why are you not coding')

**Example2**

try:   
 open("fantasy.txt")  
except:   
 print('Something went wrong')  
  
print('Should reach here')

***OR/ You can write different logic for each type of exception that happens:***

|  |
| --- |
| try:   # your code here except FileNotFoundError:   # handle exception  except IsADirectoryError:  # handle exception except:  # all other types of exceptions  print('Should reach here') |

**Example3**

This example tries to open a file where you do not have write permission, so it raises an exception –

try:

fh = open("testfile", "r")

fh.write("This is my test file for exception handling!!")

except IOError:

print "Error: can\'t find file or read data"

else:

print "Written content in the file successfully"

**Example4**

The program asks for numeric user input. Instead the user types characters in the input box. The program normally would crash. But with a try-except block it can be handled properly.

try:  
 x = input("Enter number: ")  
 x = x + 1  
 print(x)  
except:  
 print("Invalid input")

**Python File Handling**

**Files**

Files are named locations on disk to store related information. They are used to permanently store data in a non-volatile memory (e.g. hard disk).

In Python, files are treated in two modes as text or binary. The file may be in the text or binary format, and each line of a file is ended with the special character.

Hence, a file operation can be done in the following order.

* Open a file
* Read or write - Performing operation
* Close the file

Python provides an ***open()*** function that accepts two arguments, file name and access mode in which the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.

Syntax:

file object = open(<file-name>, <access-mode>)

**Access Mods Table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **SN** | | **Access mode** | | **Description** | |
| 1 | r | | It opens the file to read-only mode. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed. | |
| 2 | rb | | It opens the file to read-only in binary format. The file pointer exists at the beginning of the file. | |
| 3 | r+ | | It opens the file to read and write both. The file pointer exists at the beginning of the file. | |
| 4 | rb+ | | It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file. | |
| 5 | w | | It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file. | |
| 6 | wb | | It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists. The file pointer exists at the beginning of the file. | |
| 7 | w+ | | It opens the file to write and read both. It is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the previously written file. It creates a new file if no file exists. The file pointer exists at the beginning of the file. | |
| 8 | wb+ | | It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file. | |
| 9 | a | | It opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any. It creates a new file if no file exists with the same name. | |
| 10 | ab | | It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name. | |
| 11 | a+ | | It opens a file to append and read both. The file pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name. | |
| 12 | ab+ | | It opens a file to append and read both in binary format. The file pointer remains at the end of the file. | |

## Example

#opens the file file.txt in read mode

fileptr = open("file.txt","r")

if fileptr:

    print("file is opened successfully")

**The close() Method**

Once all the operations are done on the file, we must close it through our Python script using the **close()** method. Any unwritten information gets destroyed once the **close()** method is called on a file object.

Syntax

fileobject.close()

**Example:**

# opens the file file.txt in read mode

fileptr = open("file.txt","r")

if fileptr:

    print("file is opened successfully")

#closes the opened file

fileptr.close()

fter closing the file, we cannot perform any operation in the file. The file needs to be properly closed. If any exception occurs while performing some operations in the file then the program terminates without closing the file.

***We should use the following method to overcome such type of problem.***

try:

   fileptr = open("file.txt")

   # perform file operations

finally:

   fileptr.close()

**The with statement**

The **with** statement was introduced in python 2.5. The with statement is useful in the case of manipulating the files. It is used in the scenario where a pair of statements is to be executed with a block of code in between.

The syntax to open a file using with the statement is given below.

with open(<file name>, <access mode>) as <file-pointer>:

    #statement suite

The advantage of using with statement is that it provides the guarantee to close the file regardless of how the nested block exits.

It is always suggestible to use the **with** statement in the case of files because, if the break, return, or exception occurs in the nested block of code then it automatically closes the file, we don't need to write the **close()** function. It doesn't let the file to corrupt.

### Example

with open("file.txt",'r') as f:

    content = f.read();

    print(content)

## Writing the File

To write some text to a file, we need to open the file using the open method with one of the following access modes.

**w:** It will overwrite the file if any file exists. The file pointer is at the beginning of the file.

**a:** It will append the existing file. The file pointer is at the end of the file. It creates a new file if no file exists.

### Example1

# open the file.txt in append mode. Create a new file if no such file exists.

fileptr = open("file2.txt", "w")

# appending the content to the file

fileptr.write('''''Python is the modern day language. It makes things so simple.

It is the fastest-growing programing language''')

# closing the opened the file

fileptr.close()

### Example2

#open the file.txt in write mode.

fileptr = open("file2.txt","a")

#overwriting the content of the file

fileptr.write(" Python has an easy syntax and user-friendly interaction.")

#closing the opened file

fileptr.close()

**Reading the File**

To read a file using the Python script, the Python provides the **read()** method. The **read()** method reads a string from the file. It can read the data in the text as well as a binary format.

The syntax of the **read()** method is given below.

Syntax:

fileobj.read(<count>)

Here, the count is the number of bytes to be read from the file starting from the beginning of the file. If the count is not specified, then it may read the content of the file until the end.

### Example

#open the file.txt in read mode. causes error if no such file exists.

fileptr = open("file2.txt","r")

#stores all the data of the file into the variable content

content = fileptr.read(10)

# prints the type of the data stored in the file

print(type(content))

#prints the content of the file

print(content)

#closes the opened file

fileptr.close()

If we use the following line, then it will print all content of the file.

content = fileptr.read()

print(content)

### Read File through for Loop

We can read the file using for loop. Consider the following example.

#open the file.txt in read mode. causes an error if no such file exists.

fileptr = open("file2.txt","r");

#running a for loop

for i in fileptr:

    print(i) # i contains each line of the file

**Read Lines of the File**

Python facilitates to read the file line by line by using a function **readline()** method. The **readline()** method reads the lines of the file from the beginning, i.e., if we use the readline() method two times, then we can get the first two lines of the file.

Consider the following example which contains a function **readline()** that reads the first line of our file **"file2.txt"** containing three lines. Consider the following example.

### Example 1: Reading lines using readline() function

#open the file.txt in read mode. causes error if no such file exists.

fileptr = open("file2.txt","r");

#stores all the data of the file into the variable content

content = fileptr.readline()

content1 = fileptr.readline()

#prints the content of the file

print(content)

print(content1)

#closes the opened file

fileptr.close()

### Example 2: Reading Lines Using readlines() function

#open the file.txt in read mode. causes error if no such file exists.

fileptr = open("file2.txt","r");

#stores all the data of the file into the variable content

content = fileptr.readlines()

#prints the content of the file

print(content)

#closes the opened file

fileptr.close()

## Creating a New File

The new file can be created by using one of the following access modes with the function open().

**x:** it creates a new file with the specified name. It causes an error a file exists with the same name.

**a:** It creates a new file with the specified name if no such file exists. It appends the content to the file if the file already exists with the specified name.

**w:** It creates a new file with the specified name if no such file exists. It overwrites the existing file.

### Example1

#open the file.txt in read mode. causes error if no such file exists.

fileptr = open("file2.txt","x")

print(fileptr)

if fileptr:

    print("File created successfully")

## Python OS module

### Renaming the file

The Python **os** module enables interaction with the operating system. The os module provides the functions that are involved in file processing operations like renaming, deleting, etc. It provides us the rename() method to rename the specified file to a new name. The syntax to use the **rename()** method is given below.

Syntax:

rename(current-name, new-name)

The first argument is the current file name and the second argument is the modified name. We can change the file name bypassing these two arguments.

**Example 1**

import os

#rename file2.txt to file3.txt

os.rename("file2.txt","file3.txt")

### Removing the file

The os module provides the **remove()** method which is used to remove the specified file. The syntax to use the **remove()** method is given below.

1. remove(file-name)

**Example 1**

import os;

#deleting the file named file3.txt

os.remove("file3.txt")

## Creating the New Directory

The **mkdir()** method is used to create the directories in the current working directory. The syntax to create the new directory is given below.

Syntax:

mkdir(directory name)

**Example 1**

import os

#creating a new directory with the name new

os.mkdir("new")

## The getcwd() method

This method returns the current working directory.

The syntax to use the getcwd() method is given below.

Syntax

os.getcwd()

**Example**

import os

os.getcwd()

## Changing the Current Working Directory

The chdir() method is used to change the current working directory to a specified directory.

The syntax to use the chdir() method is given below.

Syntax

chdir("new-directory")

### Example

import os

# Changing current directory with the new directiory

os.chdir("C:\\Users\\DEVANSH SHARMA\\Documents")

#It will display the current working directory

os.getcwd()

## Deleting Directory

The rmdir() method is used to delete the specified directory.

The syntax to use the rmdir() method is given below.

Syntax

os.rmdir(directory name)

**Example1**

import os

#removing the new directory

os.rmdir("directory\_name")

It will remove the specified directory.

**MODULE2: NumPy**

## What is NumPy?

NumPy stands for Numerical Python and is the core library for numeric and scientific computing.

NumPy is a Python library used for working with 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 Getting Started

### Example

import numpy  
  
arr = numpy.array([1, 2, 3, 4, 5])  
  
print(arr)

## Checking NumPy Version

The version string is stored under \_\_version\_\_ attribute.

### Example

import numpy as np  
  
print(np.\_\_version\_\_)

# NumPy Creating Arrays

# Create a NumPy ndarray Object

NumPy is used to work with arrays. The array object in NumPy is called ndarray.

We can create a NumPy ndarray object by using the array() function.

### Example

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5])  
  
print(arr)  
  
print(type(arr))

To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray:

### Example

Use a tuple to create a NumPy array:

import numpy as np  
  
arr = np.array((1, 2, 3, 4, 5))  
  
print(arr)

# Dimensions in Arrays

A dimension in arrays is one level of array depth (nested arrays).

nested array: are arrays that have arrays as their elements.

# 0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

### Example

Create a 0-D array with value 42

import numpy as np  
  
arr = np.array(42)  
  
print(arr)

# 1-D Arrays

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.

These are the most common and basic arrays.

### Example

Create a 1-D array containing the values 1,2,3,4,5:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5])  
  
print(arr)

# 2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array.

These are often used to represent matrix or 2nd order tensors.

NumPy has a whole sub module dedicated towards matrix operations called numpy.mat

### Example

Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6:

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

# 3-D arrays

An array that has 2-D arrays (matrices) as its elements is called 3-D array.

These are often used to represent a 3rd order tensor.

### Example

Create a 3-D array with two 2-D arrays, both containing two arrays with the values 1,2,3 and 4,5,6:

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

# Check Number of Dimensions?

NumPy Arrays provides the ndim attribute that returns an integer that tells us how many dimensions the array have.

### Example

Check how many dimensions the arrays have:

import numpy as np  
  
a = np.array(42)  
b = np.array([1, 2, 3, 4, 5])  
c = np.array([[1, 2, 3], [4, 5, 6]])  
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])  
print(a.ndim)  
print(b.ndim)  
print(c.ndim)  
print(d.ndim)

# Higher Dimensional Arrays

An array can have any number of dimensions.

When the array is created, you can define the number of dimensions by using the ndmin argument.

### Example

Create an array with 5 dimensions and verify that it has 5 dimensions:

import numpy as np  
  
arr = np.array([1, 2, 3, 4], ndmin=5)  
  
print(arr)  
print('number of dimensions :', arr.ndim)

In this array the innermost dimension (5th dim) has 4 elements, the 4th dim has 1 element that is the vector, the 3rd dim has 1 element that is the matrix with the vector, the 2nd dim has 1 element that is 3D array and 1st dim has 1 element that is a 4D array.

# NumPy Array Indexing

# Access Array Elements

Array indexing is the same as accessing an array element.

You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

### Example

Get the first element from the following array:

import numpy as np  
  
arr = np.array([1, 2, 3, 4])  
  
print(arr[0])

### Example

Get the second element from the following array.

import numpy as np  
  
arr = np.array([1, 2, 3, 4])  
  
print(arr[1])

### Example

Get third and fourth elements from the following array and add them.

import numpy as np  
  
arr = np.array([1, 2, 3, 4])  
  
print(arr[2] + arr[3])

# Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

### Example

Access the 2nd element on 1st dim:

import numpy as np  
  
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])  
  
print('2nd element on 1st dim: ', arr[0, 1])

### Example

Access the 5th element on 2nd dim:

import numpy as np  
  
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])  
  
print('5th element on 2nd dim: ', arr[1, 4])

# Access 3-D Arrays

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

### Example

Access the third element of the second array of the first array:

import numpy as np  
  
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])  
  
print(arr[0, 1, 2])

# Negative Indexing

Use negative indexing to access an array from the end.

### Example

Print the last element from the 2nd dim:

import numpy as np  
  
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])  
  
print('Last element from 2nd dim: ', arr[1, -1])

Top of Form

# NumPy Array Slicing

# Slicing arrays

Slicing in python means taking elements from one given index to another given index.

We pass slice instead of index like this: [*start*:*end*].

We can also define the step, like this: [*start*:*end*:*step*].

If we don't pass start its considered 0

If we don't pass end its considered length of array in that dimension

If we don't pass step its considered 1

### Example

Slice elements from index 1 to index 5 from the following array:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[1:5])

**Note:** The result *includes* the start index, but *excludes* the end index.

### Example

Slice elements from index 4 to the end of the array:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[4:])

### Example

Slice elements from the beginning to index 4 (not included):

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[:4])

# Negative Slicing

Use the minus operator to refer to an index from the end:

### Example

Slice from the index 3 from the end to index 1 from the end:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[-3:-1])

# STEP

Use the step value to determine the step of the slicing:

### Example

Return every other element from index 1 to index 5:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[1:5:2])

### Example

Return every other element from the entire array:

import numpy as np  
  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
  
print(arr[::2])

# Slicing 2-D Arrays

### Example

From the second element, slice elements from index 1 to index 4 (not included):

import numpy as np  
  
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])  
  
print(arr[1, 1:4])

**Note:** Remember that *second element* has index 1.

### Example

From both elements, return index 2:

import numpy as np  
  
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])  
  
print(arr[0:2, 2])

### Example

From both elements, slice index 1 to index 4 (not included), this will return a 2-D array:

import numpy as np  
  
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])  
  
print(arr[0:2, 1:4])

**MODULE3:**

**DATA ANALYSIS USING PANDAS**

## What Is Pandas In Python?

Pandas is an open-source Python package developed by Wes McKinney that is most widely used for data science/data analysis and machine learning tasks.

According to the Wikipedia page on Pandas, “the name is derived from the term “panel data”, an econometrics term for multidimensional structured data sets.”

It is built on the Numpy package and its key data structure is called the DataFrame. DataFrames allow you to store and manipulate tabular data in rows of observations and columns of variables.

**Why Use Pandas?**

* Pandas allows us to analyze big data and make conclusions based on statistical theories.
* Pandas can clean messy data sets, and make them readable and relevant.
* Relevant data is very important in data science.

**Advantages of Pandas**

* Fast and efficient for manipulating and analyzing data.
* Data from different file objects can be loaded.
* Easy handling of missing data (represented as NaN) in floating point as well as non-floating point data
* Size mutability: columns can be inserted and deleted from DataFrame and higher dimensional objects
* Data set merging and joining.
* Flexible reshaping and pivoting of data sets
* Provides time-series functionality.
* Powerful group by functionality for performing split-apply-combine operations on data sets.

**What Can You Do With DataFrames Using Pandas?**

Pandas makes it simple to do many of the time consuming, repetitive tasks associated with working with data, including:

* Data cleansing
* Data fill
* Data normalization
* Merges and joins
* Data visualization
* Statistical analysis
* Data inspection
* Loading and saving data
* And much more

In fact, with Pandas, you can do everything that makes world-leading data scientists vote Pandas as the best data analysis and manipulation tool available.

## Why Pandas is used for Data Science

Pandas is generally used for data science but have you wondered why? This is because pandas is used in conjunction with other libraries that are used for data science. It is built on the top of the **NumPy** library which means that a lot of structures of NumPy are used or replicated in Pandas. The data produced by Pandas is often used as input for plotting functions of **Matplotlib**, statistical analysis in **SciPy**, machine learning algorithm in **Scikit-learn**.

Pandas program can be run from any text editor but it is recommended to use Jupyter Notebook for this as Jupyter given the ability to execute code in a particular cell rather than executing the entire file. Jupyter also provides an easy way to visualize pandas dataframe and plots.

## Getting Started

After the pandas has been installed into the system, you need to import the library. This module is generally imported as –

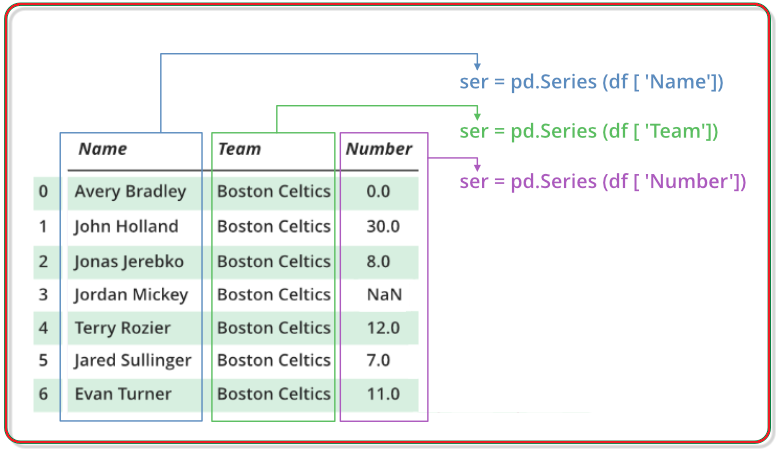
*import pandas as pd*

Pandas generally provide two data structure for manipulating data, They are:

* Series
* DataFrame

Pandas Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called index. Pandas Series is nothing but a column in an excel sheet. Labels need not be unique but must be a hashable type. The object supports both integer and label-based indexing and provides a host of methods for performing operations involving the index.

#### Creating a Series



In the real world, a Pandas Series will be created by loading the datasets from existing storage, storage can be SQL Database, CSV file, and Excel file. Pandas Series can be created from the lists, dictionary, and from a scalar value etc.

#### Example

Create a simple Pandas Series from a list:

import pandas as pd  
a = [1, 7, 2]  
myvar = pd.Series(a)  
print(myvar)

#### Labels

If nothing else is specified, the values are labeled with their index number. First value has index 0, second value has index 1 etc.

This label can be used to access a specified value.

#### Example

Return the 1st value of the series

print(myvar[0])

#### Create Labels

With the index argument, you can name your own labels.

**Example**

Create your own labels:

import pandas as pd  
a = [1, 7, 2]  
myvar = pd.Series(a, index = ["x", "y", "z"])  
print(myvar)

When you have created labels, you can access an item by referring to the label.

Example

Return the value of "y":

#### Key/Value Objects as Series

You can also use a key/value object, like a dictionary, when creating a Series.

#### Example1

#### Create a simple Pandas Series from a dictionary:

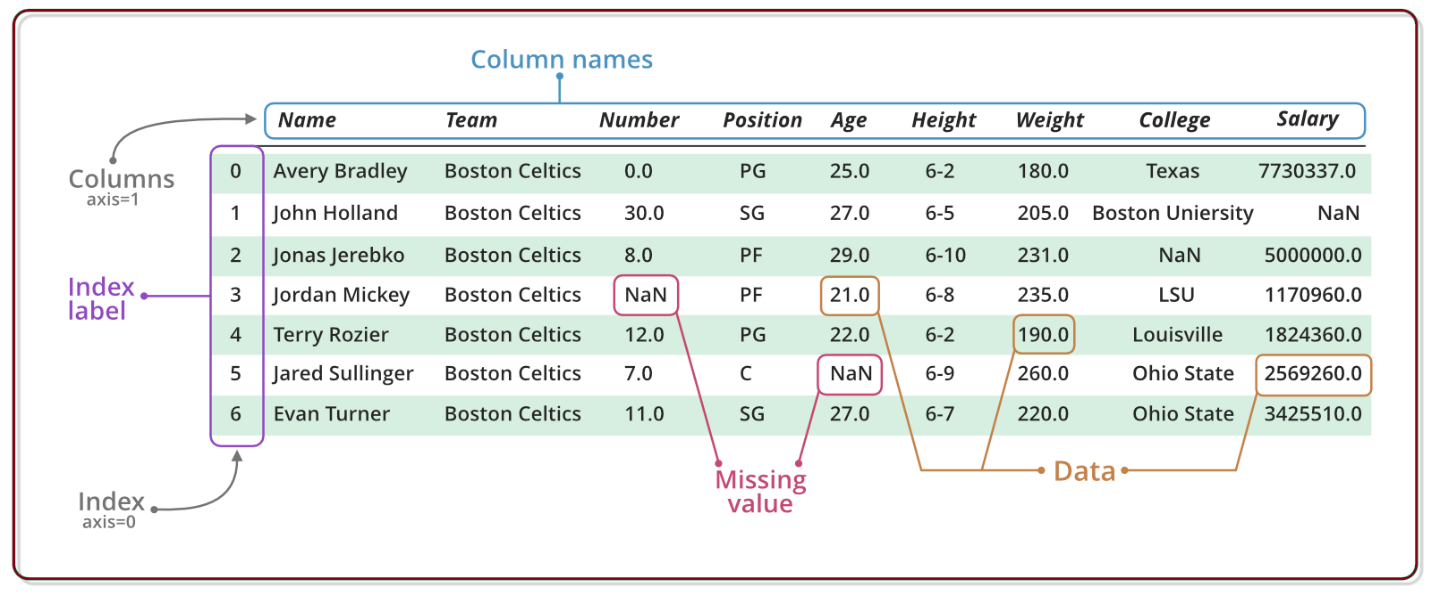
import pandas as pd  
calories = {"day1": 420, "day2": 380, "day3": 390}  
myvar = pd.Series(calories)  
print(myvar)

#### Example2

Create a Series using only data from "day1" and "day2":

import pandas as pd  
calories = {"day1": 420, "day2": 380, "day3": 390}  
myvar = pd.Series(calories, index = ["day1", "day2"])  
print(myvar)

### Creating a DataFrame



#### Pandas DataFrame is two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. Pandas DataFrame consists of three principal components, the data, rows, and columns.

**We can create a DataFrame using following ways:**

* dict
* Lists
* Numpy ndarrrays
* Series

**Create an empty DataFrame**

The below code shows how to create an empty DataFrame in Pandas:

# importing the pandas library

import pandas as pd

df = pd.DataFrame()

print (df)

**Creating a dataframe using List:**

DataFrame can be created using a single list or a list of lists.

|  |
| --- |
| # import pandas as pd  import pandas as pd    # list of strings  lst = ['Python', 'Pandas']    # Calling DataFrame constructor on list  df = pd.DataFrame(lst)  print(df) |

### Create a DataFrame from Dict of ndarrays/ Lists

# importing the pandas library

import pandas as pd

info = {'ID' :[101, 102, 103],'Department' :['B.Sc','B.Tech','M.Tech',]}

df = pd.DataFrame(info)

print (df)

**Create pandas dataframe from lists using dictionary:**

|  |
| --- |
| # importing pandas as pd  import pandas as pd    # dictionary of lists  dict = {'name':["aparna", "pankaj", "sudhir", "Geeku"],          'degree': ["MBA", "BCA", "M.Tech", "MBA"],          'score':[90, 40, 80, 98]}    df = pd.DataFrame(dict)    print(df) |

## Column Selection

We will understand this by selecting a column from the DataFrame.

### Example

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)

print df ['one']

## Column Addition

We will understand this by adding a new column to an existing data frame.

### Example

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)

# 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

## Column Deletion

Columns can be deleted or popped; let us take an example to understand how.

### Example

# Using the previous DataFrame, we will delete a column

# using del function

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

We will now understand row selection, addition and deletion through examples. Let us begin with the concept of selection.

*Rows can be selected by passing row label to a loc function*.

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)

print df.loc['b']

### Selection by integer location

Rows can be selected by passing integer location to an iloc function.

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)

print df.iloc[2]

### Slice Rows

Multiple rows can be selected using ‘ : ’ operator.

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)

print df[2:4]

### Addition of Rows

Add new rows to a DataFrame using the append function. This function will append the rows at the end.

import pandas as pd

df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])

df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])

df = df.append(df2)

print df

### Deletion of Rows

Use index label to delete or drop rows from a DataFrame. If label is duplicated, then multiple rows will be dropped.

If you observe, in the above example, the labels are duplicate. Let us drop a label and will see how many rows will get dropped.

import pandas as pd

df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])

df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])

df = df.append(df2)

# Drop rows with label 0

df = df.drop(0)

print df

## Load Files Into a DataFrame

If your data sets are stored in a file, Pandas can load them into a DataFrame.

### 1). Load a Text file into a DataFrame:

import pandas as pd

#read text file into pandas DataFrame

df = pd.read\_csv("data.txt", sep=" ")

#display DataFrame

print(df)

### ****Read a Text File with No Header****

#read text file into pandas DataFrame

df = pd.read\_csv("data.txt", sep=" ", header=None)

#display DataFrame

print(df)

### ****Read a Text File with No Header & Specify Column Names****

#read text file into pandas DataFrame and specify column names

df = pd.read\_csv("data.txt", sep=" ", header=None, names=["A", "B"])

#display DataFrame

print(df)

### 2). Load a comma separated file (CSV file) into a DataFrame:

#import CSV file as DataFrame

df = pd.read\_csv('data.csv')

#view DataFrame

df

### ****Read Specific Columns from CSV File****

#import only specific columns from CSV file

df = pd.read\_csv('data.csv', usecols=['playerID', 'points'])

#view DataFrame

df

Alternatively you can specify column indices to read into a pandas DataFrame:

#import only specific columns from CSV file

df = pd.read\_csv('data.csv', usecols=[0, 1])

#view DataFrame

df

### ****Specify Header Row when Importing CSV File****

To read this CSV file into a pandas DataFrame, we can specify **header=1**as follows:

#import from CSV file and specify that header starts on second row

df = pd.read\_csv('data.csv', header=1)

#view DataFrame

df

### ****Skip Rows when Importing CSV File****

You can also easily skip rows when importing a CSV file by using the **skiprows**argument. For example, the following code shows how to skip the second row when importing the CSV file:

#import from CSV file and skip second row

df = pd.read\_csv('data.csv', skiprows=[1])

#view DataFrame

df

And the following code shows how to skip the second *and* third row when importing the CSV file:

#import from CSV file and skip second and third rows

df = pd.read\_csv('data.csv', skiprows=[1, 2])

#view DataFrame

df

### ****Read CSV Files with Custom Delimiter****

Occasionally you may have a CSV file with a delimiter that is different from a comma. For example, suppose our CSV file has an underscore as a delimiter:

playerID\_team\_points

1\_Lakers\_26

2\_Mavs\_19

3\_Bucks\_24

4\_Spurs\_22

#import from CSV file and specify delimiter to use

df = pd.read\_csv('data.csv', sep='\_')

#view DataFrame

df

### 3). Load a Excel File into a DataFrame:

import pandas as pd

#import Excel file

df = pd.read\_excel('data.xlsx')

#view DataFrame

df

### ****Read Excel File Using Sheet Name****

To read a specific sheet in as a pandas DataFrame, you can use the **sheet\_name()**argument:

import pandas as pd

#import only second sheet

df = pd.read\_excel('data.xlsx', sheet\_name='second sheet')

#view DataFrame

df

### Load a JSON file into a DataFrame:

import pandas as pd  
  
df = pd.read\_json('data.json')  
  
print(df.to\_string())

#### 

### Loading data from a SQL database

!pip install pysqlite3

import sqlite3

con = sqlite3.connect("database.db")

By passing a SELECT query and our con, we can read from the purchases table:

df = pd.read\_sql\_query("SELECT \* FROM purchases", con)

df

Just like with CSVs, we could pass index\_col='index', but we can also set an index after-the-fact:

df = df.set\_index('index')

df

### Writing a Pandas DataFrame to CSV File:

**Example #1: Save csv to working directory.**

|  |
| --- |
| # importing pandas as pd  import pandas as pd    # list of name, degree, score  nme = ["aparna", "pankaj", "sudhir", "Geeku"]  deg = ["MBA", "BCA", "M.Tech", "MBA"]  scr = [90, 40, 80, 98]    # dictionary of lists  dict = {'name': nme, 'degree': deg, 'score': scr}    df = pd.DataFrame(dict)    # saving the dataframe  df.to\_csv('file1.csv') |

**Example #2: Saving CSV without *headers*and *index*.**

# importing pandas as pd

import pandas as pd

# list of name, degree, score

nme = ["aparna", "pankaj", "sudhir", "Geeku"]

deg = ["MBA", "BCA", "M.Tech", "MBA"]

scr = [90, 40, 80, 98]

# dictionary of lists

dict = {'name': nme, 'degree': deg, 'score': scr}

df = pd.DataFrame(dict)

# saving the dataframe

df.to\_csv('file2.csv', header=False, index=False)

**Example #3: Save csv file to a specified location.**

|  |
| --- |
| # importing pandas as pd  import pandas as pd  # list of name, degree, score  nme = ["aparna", "pankaj", "sudhir", "Geeku"]  deg = ["MBA", "BCA", "M.Tech", "MBA"]  scr = [90, 40, 80, 98]    # dictionary of lists  dict = {'name': nme, 'degree': deg, 'score': scr}    df = pd.DataFrame(dict)    # saving the dataframe  df.to\_csv(r'C:\Users\Admin\Desktop\file3.csv', index=False) |

**Example #4: Write a DataFrame to CSV file using tab separator.**

|  |
| --- |
| import pandas as pd  import numpy as np  users = {'Name': ['Amit', 'Cody', 'Drew'],       'Age': [20,21,25]}  df = pd.DataFrame(users, columns=['Name','Age'])#create DataFrame  print("Original DataFrame:")  print(df)  print('Data from Users.csv:')  df.to\_csv('Users.csv', sep='\t', index=False,header=True)  new\_df = pd.read\_csv('Users.csv')  print(new\_df) |

### Write a DataFrame to Excel File:

import pandas as pd  
import openpyxl  
  
df = pd.DataFrame([[11, 21, 31], [12, 22, 32], [31, 32, 33]],  
 index=['one', 'two', 'three'], columns=['a', 'b', 'c'])

df.to\_excel('pandas\_to\_excel.xlsx', sheet\_name='new\_sheet\_name')

**If you do not need to write index (row name), columns (column name), the argument index, columns is False.**

|  |
| --- |
| df.to\_excel('pandas\_to\_excel\_no\_index\_header.xlsx', index=False, header=False) |

### Write a DataFrame to JSON File:

df = pd.read\_csv("data.csv")

# Save dataframe to JSON format

df.to\_json("data.json")

### Viewing your data

**The first thing to do when opening a new dataset is print out a few rows to keep as a visual reference. We accomplish this with *.head()*:**

movies\_df.head()

**To see the last five rows use *.tail()*. tail() also accepts a number, and in this case we printing the bottom two rows.:**

movies\_df.tail()

### Getting info about your data

***.info()* should be one of the very first commands you run after loading your data:**

movies\_df.info()

**Another fast and useful attribute is *.shape*, which outputs just a tuple of (rows, columns):**

movies\_df.shape

# Descriptive Statistics

**A large number of methods collectively compute descriptive statistics and other related operations on DataFrame. Most of these are aggregations like sum(), mean(), but some of them, like sumsum(), produce an object of the same size. Generally speaking, these methods take an axis argument, just like ndarray.{sum, std, ...}, but the axis can be specified by name or integer**

DataFrame − “index” (axis=0, default), “columns” (axis=1)

**Let us create a DataFrame and use this object throughout this chapter for all the operations.**

### Example

import pandas as pd

import numpy as np

#Create a Dictionary of series

d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',

'Lee','David','Gasper','Betina','Andres']),

'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),

'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])

}

#Create a DataFrame

df = pd.DataFrame(d)

print df

### sum()

**Returns the sum of the values for the requested axis. By default, axis is index (axis=0).**

import pandas as pd

import numpy as np

#Create a Dictionary of series

d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',

'Lee','David','Gasper','Betina','Andres']),

'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),

'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])

}

#Create a DataFrame

df = pd.DataFrame(d)

print df.sum()

Each individual column is added individually (Strings are appended).

### mean()

Returns the average value

import pandas as pd

import numpy as np

#Create a Dictionary of series

d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',

'Lee','David','Gasper','Betina','Andres']),

'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),

'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])

}

#Create a DataFrame

df = pd.DataFrame(d)

print df.mean()

### std()

Returns the Bressel standard deviation of the numerical columns.

import pandas as pd

import numpy as np

#Create a Dictionary of series

d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',

'Lee','David','Gasper','Betina','Andres']),

'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),

'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])

}

#Create a DataFrame

df = pd.DataFrame(d)

print df.std()

### Functions & Description

Let us now understand the functions under Descriptive Statistics in Python Pandas. The following table list down the important functions –

|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **Function** | **Description** |
| 1 | count() | Number of non-null observations |
| 2 | sum() | Sum of values |
| 3 | mean() | Mean of Values |
| 4 | median() | Median of Values |
| 5 | mode() | Mode of values |
| 6 | std() | Standard Deviation of the Values |
| 7 | min() | Minimum Value |
| 8 | max() | Maximum Value |
| 9 | abs() | Absolute Value |
| 10 | prod() | Product of Values |
| 11 | cumsum() | Cumulative Sum |
| 12 | cumprod() | Cumulative Product |

### Summarizing Data

The **describe()** function computes a summary of statistics pertaining to the DataFrame columns.

import pandas as pd

import numpy as np

#Create a Dictionary of series

d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',

'Lee','David','Gasper','Betina','Andres']),

'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),

'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])

}

#Create a DataFrame

df = pd.DataFrame(d)

print df.describe()

Its **output** is as follows −

Age Rating

count 12.000000 12.000000

mean 31.833333 3.743333

std 9.232682 0.661628

min 23.000000 2.560000

25% 25.000000 3.230000

50% 29.500000 3.790000

75% 35.500000 4.132500

max 51.000000 4.800000

This function gives the **mean, std** and **IQR** values. And, function excludes the character columns and given summary about numeric columns. **'include'** is the argument which is used to pass necessary information regarding what columns need to be considered for summarizing. Takes the list of values; by default, 'number'.

* **object** − Summarizes String columns
* **number** − Summarizes Numeric columns
* **all** − Summarizes all columns together (Should not pass it as a list value)

Now, use the following statement in the program and check the output –

import pandas as pd

import numpy as np

#Create a Dictionary of series

d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',

'Lee','David','Gasper','Betina','Andres']),

'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),

'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])

}

#Create a DataFrame

df = pd.DataFrame(d)

print df.describe(include=['object'])

Its **output** is as follows −

Name

count 12

unique 12

top Ricky

freq 1

Now, use the following statement and check the output –

import pandas as pd

import numpy as np

#Create a Dictionary of series

d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',

'Lee','David','Gasper','Betina','Andres']),

'Age':pd.Series([25,26,25,23,30,29,23,34,40,30,51,46]),

'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])

}

#Create a DataFrame

df = pd.DataFrame(d)

print df. describe(include='all')

### Handling duplicates

This dataset does not have duplicate rows, but it is always important to verify you aren't aggregating duplicate rows.

To demonstrate, let's simply just double up our movies DataFrame by appending it to itself:

temp\_df = movies\_df.append(movies\_df)

temp\_df.shape

Using append() will return a copy without affecting the original DataFrame. We are capturing this copy in temp so we aren't working with the real data.

Notice call .shape quickly proves our DataFrame rows have doubled.

Now we can try dropping duplicates:

temp\_df = temp\_df.drop\_duplicates()

temp\_df.shape

Another important argument for drop\_duplicates() is keep, which has three possible options:

first: (default) Drop duplicates except for the first occurrence.

last: Drop duplicates except for the last occurrence.

False: Drop all duplicates.

temp\_df = movies\_df.append(movies\_df) # make a new copy

temp\_df.drop\_duplicates(inplace=True, keep=False)

temp\_df.shape

### How to work with Missing values

Missing data is always a problem in real life scenarios. Areas like machine learning and data mining face severe issues in the accuracy of their model predictions because of poor quality of data caused by missing values. In these areas, missing value treatment is a major point of focus to make their models more accurate and valid.

## When and Why Is Data Missed?

Let us consider an online survey for a product. Many a times, people do not share all the information related to them. Few people share their experience, but not how long they are using the product; few people share how long they are using the product, their experience but not their contact information. Thus, in some or the other way a part of data is always missing, and this is very common in real time.

Let us now see how we can handle missing values (say NA or NaN) using Pandas.

# import the pandas library

import pandas as pd

import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',

'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df

Using reindexing, we have created a DataFrame with missing values. In the output, **NaN** means **Not a Number.**

### Check for Missing Values

To make detecting missing values easier (and across different array dtypes), Pandas provides the **isnull()** and **notnull()** functions, which are also methods on Series and DataFrame objects −

### Example 1

import pandas as pd

import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',

'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df['one'].isnull()

### Example 2

import pandas as pd

import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',

'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df['one'].notnull()

### Calculations with Missing Data

* When summing data, NA will be treated as Zero
* If the data are all NA, then the result will be NA

### Example 1

import pandas as pd

import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',

'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df['one'].sum()

### Example 2

### 

import pandas as pd

import numpy as np

df = pd.DataFrame(index=[0,1,2,3,4,5],columns=['one','two'])

print df['one'].sum()

## Cleaning / Filling Missing Data

Pandas provides various methods for cleaning the missing values. The fillna function can “fill in” NA values with non-null data in a couple of ways, which we have illustrated in the following sections.

## Replace NaN with a Scalar Value

The following program shows how you can replace "NaN" with "0".

import pandas as pd

import numpy as np

df = pd.DataFrame(np.random.randn(3, 3), index=['a', 'c', 'e'],columns=['one',

'two', 'three'])

df = df.reindex(['a', 'b', 'c'])

print df

print ("NaN replaced with '0':")

print df.fillna(0)

Here, we are filling with value zero; instead we can also fill with any other value.

## Fill NA Forward and Backward

Using the concepts of filling discussed in the ReIndexing Chapter we will fill the missing values.

|  |  |
| --- | --- |
| **Sr.No** | **Method & Action** |
| 1 | **pad/fill**  Fill methods Forward |
| 2 | **bfill/backfill**  Fill methods Backward |

### Example 1

import pandas as pd

import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',

'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df.fillna(method='pad')

### Example 2

import pandas as pd

import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',

'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df.fillna(method='backfill')

## Drop Missing Values

If you want to simply exclude the missing values, then use the **dropna** function along with the **axis** argument. By default, axis=0, i.e., along row, which means that if any value within a row is NA then the whole row is excluded.

### Example 1

import pandas as pd

import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',

'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df.dropna()

### Example 2

import pandas as pd

import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',

'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df.dropna(axis=1)

## Replace Missing (or) Generic Values

Many times, we have to replace a generic value with some specific value. We can achieve this by applying the replace method.

Replacing NA with a scalar value is equivalent behavior of the **fillna()** function.

### Example1

import pandas as pd

import numpy as np

df = pd.DataFrame({'one':[10,20,30,40,50,2000], 'two':[1000,0,30,40,50,60]})

print df.replace({1000:10,2000:60})

### Example 2

import pandas as pd

import numpy as np

df = pd.DataFrame({'one':[10,20,30,40,50,2000], 'two':[1000,0,30,40,50,60]})

print df.replace({1000:10,2000:60})

# GroupBy

Any **groupby** operation involves one of the following operations on the original object. They are −

* **Splitting** the Object
* **Applying** a function
* **Combining** the results

In many situations, we split the data into sets and we apply some functionality on each subset. In the apply functionality, we can perform the following operations −

* **Aggregation** − computing a summary statistic
* **Transformation** − perform some group-specific operation
* **Filtration** − discarding the data with some condition

Let us now create a DataFrame object and perform all the operations on it –

#import the pandas library

import pandas as pd

ipl\_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils', 'Kings',

'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals', 'Riders'],

'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],

'Year': [2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],

'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}

df = pd.DataFrame(ipl\_data)

print df

## Split Data into Groups

Pandas object can be split into any of their objects. There are multiple ways to split an object like −

* obj.groupby('key')
* obj.groupby(['key1','key2'])
* obj.groupby(key,axis=1)

Let us now see how the grouping objects can be applied to the DataFrame object

### Example

# import the pandas library

import pandas as pd

ipl\_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils', 'Kings',

'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals', 'Riders'],

'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],

'Year': [2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],

'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}

df = pd.DataFrame(ipl\_data)

print df.groupby('Team')

## View Groups

# import the pandas library

import pandas as pd

ipl\_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils', 'Kings',

'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals', 'Riders'],

'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],

'Year': [2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],

'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}

df = pd.DataFrame(ipl\_data)

print df.groupby('Team').groups

### Example

**Group by** with multiple columns –

# import the pandas library

import pandas as pd

ipl\_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils', 'Kings',

'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals', 'Riders'],

'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],

'Year': [2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],

'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}

df = pd.DataFrame(ipl\_data)

print df.groupby(['Team','Year']).groups

## Iterating through Groups

With the **groupby** object in hand, we can iterate through the object similar to itertools.obj.

# import the pandas library

import pandas as pd

ipl\_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils', 'Kings',

'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals', 'Riders'],

'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],

'Year': [2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],

'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}

df = pd.DataFrame(ipl\_data)

grouped = df.groupby('Year')

for name,group in grouped:

print name

print group

By default, the **groupby** object has the same label name as the group name.

## Select a Group

Using the **get\_group()** method, we can select a single group.

# import the pandas library

import pandas as pd

ipl\_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils', 'Kings',

'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals', 'Riders'],

'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],

'Year': [2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],

'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}

df = pd.DataFrame(ipl\_data)

grouped = df.groupby('Year')

print grouped.get\_group(2014)

## Aggregations

An aggregated function returns a single aggregated value for each group. Once the **group by** object is created, several aggregation operations can be performed on the grouped data.

An obvious one is aggregation via the aggregate or equivalent **agg** method –

# import the pandas library

import pandas as pd

import numpy as np

ipl\_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils', 'Kings',

'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals', 'Riders'],

'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],

'Year': [2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],

'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}

df = pd.DataFrame(ipl\_data)

grouped = df.groupby('Year')

print grouped['Points'].agg(np.mean)

Another way to see the size of each group is by applying the size() function –

import pandas as pd

import numpy as np

ipl\_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils', 'Kings',

'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals', 'Riders'],

'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],

'Year': [2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],

'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}

df = pd.DataFrame(ipl\_data)

Attribute Access in Python Pandas

grouped = df.groupby('Team')

print grouped.agg(np.size)

### Applying Multiple Aggregation Functions at Once

With grouped Series, you can also pass a **list** or **dict of functions** to do aggregation with, and generate DataFrame as output –

# import the pandas library

import pandas as pd

import numpy as np

ipl\_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils', 'Kings',

'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals', 'Riders'],

'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],

'Year': [2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],

'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}

df = pd.DataFrame(ipl\_data)

grouped = df.groupby('Team')

print grouped['Points'].agg([np.sum, np.mean, np.std])

## Transformations

Transformation on a group or a column returns an object that is indexed the same size of that is being grouped. Thus, the transform should return a result that is the same size as that of a group chunk.

# import the pandas library

import pandas as pd

import numpy as np

ipl\_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils', 'Kings',

'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals', 'Riders'],

'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],

'Year': [2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],

'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}

df = pd.DataFrame(ipl\_data)

grouped = df.groupby('Team')

score = lambda x: (x - x.mean()) / x.std()\*10

print grouped.transform(score)

# FILTER PANDAS DATAFRAME

**Examples of Data Filtering**

It is one of the most initial step of data preparation for predictive modeling or any reporting project. It is also called 'Subsetting Data'.

See some of the examples of data filtering below.

* Select all the active customers whose accounts were opened after 1st January 2019
* Extract details of all the customers who made more than 3 transactions in the last 6 months
* Fetch information of employees who spent more than 3 years in the organization and received highest rating in the past 2 years
* Analyze complaints data and identify customers who filed more than 5 complaints in the last 1 year
* Extract details of metro cities where per capita income is greater than 40K dollars

**Filter pandas dataframe by column value**

**Method1: DataFrame Way**

newdf = df[(df.origin == "JFK") & (df.carrier == "B6")]

**Method2: Query Function**

newdf = df.query('origin == "JFK" & carrier == "B6"')

**Method3: loc function**

loc is an abbreviation of **location**term. All these 3 methods return same output. It's just a different ways of doing filtering rows.

newdf = df.loc[(df.origin == "JFK") & (df.carrier == "B6")]

**Filter Pandas Dataframe by Row and Column Position**

Suppose you want to select specific rows by their position (let's say from second through fifth row). We can use df.iloc[ ] function for the same.

Indexing in python starts from zero. df.iloc[0:5,] refers to first to fifth row (excluding end point 6th row here). df.iloc[0:5,] is equivalent to df.iloc[:5,]

df.iloc[:5,] #First 5 rows

df.iloc[1:5,] #Second to Fifth row

df.iloc[5,0] #Sixth row and 1st column

df.iloc[1:5,0] #Second to Fifth row, first column

df.iloc[1:5,:5] #Second to Fifth row, first 5 columns

df.iloc[2:7,1:3] #Third to Seventh row, 2nd and 3rd column

**Merging/Joining**

Pandas has full-featured, high performance in-memory join operations idiomatically very similar to relational databases like SQL.

Pandas provides a single function, **merge**, as the entry point for all standard database join operations between DataFrame objects –

pd.merge(left, right, how='inner', on=None, left\_on=None, right\_on=None,

left\_index=False, right\_index=False, sort=True)

Here, we have used the following parameters –

* **left** − A DataFrame object.
* **right** − Another DataFrame object.
* **on** − Columns (names) to join on. Must be found in both the left and right DataFrame objects.
* **left\_on** − Columns from the left DataFrame to use as keys. Can either be column names or arrays with length equal to the length of the DataFrame.
* **right\_on** − Columns from the right DataFrame to use as keys. Can either be column names or arrays with length equal to the length of the DataFrame.
* **left\_index** − If **True,** use the index (row labels) from the left DataFrame as its join key(s). In case of a DataFrame with a MultiIndex (hierarchical), the number of levels must match the number of join keys from the right DataFrame.
* **right\_index** − Same usage as **left\_index** for the right DataFrame.
* **how** − One of 'left', 'right', 'outer', 'inner'. Defaults to inner. Each method has been described below.
* **sort** − Sort the result DataFrame by the join keys in lexicographical order. Defaults to True, setting to False will improve the performance substantially in many cases.

Let us now create two different DataFrames and perform the merging operations on it.

# import the pandas library

import pandas as pd

left = pd.DataFrame({

'id':[1,2,3,4,5],

'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],

'subject\_id':['sub1','sub2','sub4','sub6','sub5']})

right = pd.DataFrame(

{'id':[1,2,3,4,5],

'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],

'subject\_id':['sub2','sub4','sub3','sub6','sub5']})

print left

print right

### Merge Two DataFrames on a Key

import pandas as pd

left = pd.DataFrame({

'id':[1,2,3,4,5],

'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],

'subject\_id':['sub1','sub2','sub4','sub6','sub5']})

right = pd.DataFrame({

'id':[1,2,3,4,5],

'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],

'subject\_id':['sub2','sub4','sub3','sub6','sub5']})

print pd.merge(left,right,on='id')

### Merge Two DataFrames on Multiple Keys

import pandas as pd

left = pd.DataFrame({

'id':[1,2,3,4,5],

'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],

'subject\_id':['sub1','sub2','sub4','sub6','sub5']})

right = pd.DataFrame({

'id':[1,2,3,4,5],

'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],

'subject\_id':['sub2','sub4','sub3','sub6','sub5']})

print pd.merge(left,right,on=['id','subject\_id'])

### Merge Using 'how' Argument

The **how** argument to merge specifies how to determine which keys are to be included in the resulting table. If a key combination does not appear in either the left or the right tables, the values in the joined table will be NA.

Here is a summary of the **how** options and their SQL equivalent names –

|  |  |  |
| --- | --- | --- |
| **Merge Method** | **SQL Equivalent** | **Description** |
| left | LEFT OUTER JOIN | Use keys from left object |
| right | RIGHT OUTER JOIN | Use keys from right object |
| outer | FULL OUTER JOIN | Use union of keys |
| inner | INNER JOIN | Use intersection of keys |

### Left Join

import pandas as pd

left = pd.DataFrame({

'id':[1,2,3,4,5],

'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],

'subject\_id':['sub1','sub2','sub4','sub6','sub5']})

right = pd.DataFrame({

'id':[1,2,3,4,5],

'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],

'subject\_id':['sub2','sub4','sub3','sub6','sub5']})

print pd.merge(left, right, on='subject\_id', how='left')

### Right Join

import pandas as pd

left = pd.DataFrame({

'id':[1,2,3,4,5],

'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],

'subject\_id':['sub1','sub2','sub4','sub6','sub5']})

right = pd.DataFrame({

'id':[1,2,3,4,5],

'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],

'subject\_id':['sub2','sub4','sub3','sub6','sub5']})

print pd.merge(left, right, on='subject\_id', how='right')

### Outer Join

import pandas as pd

left = pd.DataFrame({

'id':[1,2,3,4,5],

'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],

'subject\_id':['sub1','sub2','sub4','sub6','sub5']})

right = pd.DataFrame({

'id':[1,2,3,4,5],

'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],

'subject\_id':['sub2','sub4','sub3','sub6','sub5']})

print pd.merge(left, right, how='outer', on='subject\_id')

### Inner Join

Joining will be performed on index. Join operation honors the object on which it is called. So, **a.join(b)** is not equal to **b.join(a)**.

import pandas as pd

left = pd.DataFrame({

'id':[1,2,3,4,5],

'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],

'subject\_id':['sub1','sub2','sub4','sub6','sub5']})

right = pd.DataFrame({

'id':[1,2,3,4,5],

'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],

'subject\_id':['sub2','sub4','sub3','sub6','sub5']})

print pd.merge(left, right, on='subject\_id', how='inner')

# Concatenation

Pandas provides various facilities for easily combining together **Series, DataFrame**, and **Panel** objects.

pd.concat(objs,axis=0,join='outer',join\_axes=None,

ignore\_index=False)

* **objs** − This is a sequence or mapping of Series, DataFrame, or Panel objects.
* **axis** − {0, 1, ...}, default 0. This is the axis to concatenate along.
* **join** − {‘inner’, ‘outer’}, default ‘outer’. How to handle indexes on other axis(es). Outer for union and inner for intersection.
* **ignore\_index** − boolean, default False. If True, do not use the index values on the concatenation axis. The resulting axis will be labeled 0, ..., n - 1.
* **join\_axes** − This is the list of Index objects. Specific indexes to use for the other (n-1) axes instead of performing inner/outer set logic.

### Concatenating Objects

The **concat** function does all of the heavy lifting of performing concatenation operations along an axis. Let us create different objects and do concatenation.

import pandas as pd

one = pd.DataFrame({

'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],

'subject\_id':['sub1','sub2','sub4','sub6','sub5'],

'Marks\_scored':[98,90,87,69,78]},

index=[1,2,3,4,5])

two = pd.DataFrame({

'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],

'subject\_id':['sub2','sub4','sub3','sub6','sub5'],

'Marks\_scored':[89,80,79,97,88]},

index=[1,2,3,4,5])

print pd.concat([one,two])

Suppose we wanted to associate specific keys with each of the pieces of the chopped up DataFrame. We can do this by using the **keys** argument –

import pandas as pd

one = pd.DataFrame({

'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],

'subject\_id':['sub1','sub2','sub4','sub6','sub5'],

'Marks\_scored':[98,90,87,69,78]},

index=[1,2,3,4,5])

two = pd.DataFrame({

'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],

'subject\_id':['sub2','sub4','sub3','sub6','sub5'],

'Marks\_scored':[89,80,79,97,88]},

index=[1,2,3,4,5])

print pd.concat([one,two],keys=['x','y'])

The index of the resultant is duplicated; each index is repeated.

If the resultant object has to follow its own indexing, set **ignore\_index** to **True**.

import pandas as pd

one = pd.DataFrame({

'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],

'subject\_id':['sub1','sub2','sub4','sub6','sub5'],

'Marks\_scored':[98,90,87,69,78]},

index=[1,2,3,4,5])

two = pd.DataFrame({

'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],

'subject\_id':['sub2','sub4','sub3','sub6','sub5'],

'Marks\_scored':[89,80,79,97,88]},

index=[1,2,3,4,5])

print pd.concat([one,two],keys=['x','y'],ignore\_index=True)

Observe, the index changes completely and the Keys are also overridden.

If two objects need to be added along **axis=1**, then the new columns will be appended.

import pandas as pd

one = pd.DataFrame({

'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],

'subject\_id':['sub1','sub2','sub4','sub6','sub5'],

'Marks\_scored':[98,90,87,69,78]},

index=[1,2,3,4,5])

two = pd.DataFrame({

'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],

'subject\_id':['sub2','sub4','sub3','sub6','sub5'],

'Marks\_scored':[89,80,79,97,88]},

index=[1,2,3,4,5])

print pd.concat([one,two],axis=1)

### Concatenating Using append

A useful shortcut to concat are the append instance methods on Series and DataFrame. These methods actually predated concat. They concatenate along **axis=0**, namely the index –

import pandas as pd

one = pd.DataFrame({

'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],

'subject\_id':['sub1','sub2','sub4','sub6','sub5'],

'Marks\_scored':[98,90,87,69,78]},

index=[1,2,3,4,5])

two = pd.DataFrame({

'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],

'subject\_id':['sub2','sub4','sub3','sub6','sub5'],

'Marks\_scored':[89,80,79,97,88]},

index=[1,2,3,4,5])

print one.append(two)

The **append** function can take multiple objects as well –

import pandas as pd

one = pd.DataFrame({

'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],

'subject\_id':['sub1','sub2','sub4','sub6','sub5'],

'Marks\_scored':[98,90,87,69,78]},

index=[1,2,3,4,5])

two = pd.DataFrame({

'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],

'subject\_id':['sub2','sub4','sub3','sub6','sub5'],

'Marks\_scored':[89,80,79,97,88]},

index=[1,2,3,4,5])

print one.append([two,one,two])

# Sorting

There are two kinds of sorting available in Pandas. They are −

* By label
* By Actual Value

Let us consider an example with an output.

import pandas as pd

import numpy as np

unsorted\_df=pd.DataFrame(np.random.randn(10,2),index=[1,4,6,2,3,5,9,8,0,7],colu

mns=['col2','col1'])

print unsorted\_df

In **unsorted\_df**, the **labels** and the **values** are unsorted. Let us see how these can be sorted.

### By Label

Using the **sort\_index()** method, by passing the axis arguments and the order of sorting, DataFrame can be sorted. By default, sorting is done on row labels in ascending order.

import pandas as pd

import numpy as np

unsorted\_df = pd.DataFrame(np.random.randn(10,2),index=[1,4,6,2,3,5,9,8,0,7],colu

mns = ['col2','col1'])

sorted\_df=unsorted\_df.sort\_index()

print sorted\_df

### Order of Sorting

By passing the Boolean value to ascending parameter, the order of the sorting can be controlled. Let us consider the following example to understand the same.

import pandas as pd

import numpy as np

unsorted\_df = pd.DataFrame(np.random.randn(10,2),index=[1,4,6,2,3,5,9,8,0,7],colu

mns = ['col2','col1'])

sorted\_df = unsorted\_df.sort\_index(ascending=False)

print sorted\_df

### Sort the Columns

By passing the axis argument with a value 0 or 1, the sorting can be done on the column labels. By default, axis=0, sort by row. Let us consider the following example to understand the same.

import pandas as pd

import numpy as np

unsorted\_df = pd.DataFrame(np.random.randn(10,2),index=[1,4,6,2,3,5,9,8,0,7],colu

mns = ['col2','col1'])

sorted\_df=unsorted\_df.sort\_index(axis=1)

print sorted\_df

### By Value

Like index sorting, **sort\_values()** is the method for sorting by values. It accepts a 'by' argument which will use the column name of the DataFrame with which the values are to be sorted.

import pandas as pd

import numpy as np

unsorted\_df = pd.DataFrame({'col1':[2,1,1,1],'col2':[1,3,2,4]})

sorted\_df = unsorted\_df.sort\_values(by='col1')

print sorted\_df

Observe, col1 values are sorted and the respective col2 value and row index will alter along with col1. Thus, they look unsorted.

**'by'** argument takes a list of column values.

import pandas as pd

import numpy as np

unsorted\_df = pd.DataFrame({'col1':[2,1,1,1],'col2':[1,3,2,4]})

sorted\_df = unsorted\_df.sort\_values(by=['col1','col2'])

print sorted\_df

## Sorting Algorithm

**sort\_values()** provides a provision to choose the algorithm from mergesort, heapsort and quicksort. Mergesort is the only stable algorithm.

import pandas as pd

import numpy as np

unsorted\_df = pd.DataFrame({'col1':[2,1,1,1],'col2':[1,3,2,4]})

sorted\_df = unsorted\_df.sort\_values(by='col1' ,kind='mergesort')

print sorted\_df

## Working with Text Data

Pandas provides a set of string functions which make it easy to operate on string data. Most importantly, these functions ignore (or exclude) missing/NaN values.

Let us now see how each operation performs.

|  |  |
| --- | --- |
| **Sr.No** | **Function & Description** |
| 1 | **lower()**  Converts strings in the Series/Index to lower case. |
| 2 | **upper()**  Converts strings in the Series/Index to upper case. |
| 3 | **len()**  Computes String length(). |
| 4 | **strip()**  Helps strip whitespace(including newline) from each string in the Series/index from both the sides. |
| 5 | **split(' ')**  Splits each string with the given pattern. |
| 6 | **cat(sep=' ')**  Concatenates the series/index elements with given separator. |
| 7 | **get\_dummies()**  Returns the DataFrame with One-Hot Encoded values. |
| 8 | **contains(pattern)**  Returns a Boolean value True for each element if the substring contains in the element, else False. |
| 9 | **replace(a,b)**  Replaces the value **a** with the value **b**. |
| 10 | **repeat(value)**  Repeats each element with specified number of times. |
| 11 | **count(pattern)**  Returns count of appearance of pattern in each element. |
| 12 | **startswith(pattern)**  Returns true if the element in the Series/Index starts with the pattern. |
| 13 | **endswith(pattern)**  Returns true if the element in the Series/Index ends with the pattern. |
| 14 | **find(pattern)**  Returns the first position of the first occurrence of the pattern. |
| 15 | **findall(pattern)**  Returns a list of all occurrence of the pattern. |
| 16 | **swapcase**  Swaps the case lower/upper. |
| 17 | **islower()**  Checks whether all characters in each string in the Series/Index in lower case or not. Returns Boolean |
| 18 | **isupper()**  Checks whether all characters in each string in the Series/Index in upper case or not. Returns Boolean. |
| 19 | **isnumeric()**  Checks whether all characters in each string in the Series/Index are numeric. Returns Boolean. |

Let us now create a Series and see how all the above functions work.

import pandas as pd

import numpy as np

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t', np.nan, '1234','SteveSmith'])

print s

### lower()

import pandas as pd

import numpy as np

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t', np.nan, '1234','SteveSmith'])

print s.str.lower()

### upper()

import pandas as pd

import numpy as np

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t', np.nan, '1234','SteveSmith'])

print s.str.upper()

### len()

import pandas as pd

import numpy as np

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t', np.nan, '1234','SteveSmith'])

print s.str.len()

### strip()

import pandas as pd

import numpy as np

s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])

print s

print ("After Stripping:")

print s.str.strip()

### split(pattern)

import pandas as pd

import numpy as np

s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])

print s

print ("Split Pattern:")

print s.str.split(' ')

### cat(sep=pattern)

import pandas as pd

import numpy as np

s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])

print s.str.cat(sep='\_')

### contains ()

import pandas as pd

s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])

print s.str.contains(' ')

### replace(a,b)

import pandas as pd

s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])

print s

print ("After replacing @ with $:")

print s.str.replace('@','$')

### repeat(value)

import pandas as pd

s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])

print s.str.repeat(2)

### count(pattern)

import pandas as pd

s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])

print ("The number of 'm's in each string:")

print s.str.count('m')

### startswith(pattern)

import pandas as pd

s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])

print ("Strings that start with 'T':")

print s.str. startswith ('T')

### endswith(pattern)

import pandas as pd

s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])

print ("Strings that end with 't':")

print s.str.endswith('t')

### find(pattern)

import pandas as pd

s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])

print s.str.find('e')

### findall(pattern)

import pandas as pd

s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])

print s.str.findall('e')

Null list([ ]) indicates that there is no such pattern available in the element.

### swapcase()

import pandas as pd

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t'])

print s.str.swapcase()

### islower()

import pandas as pd

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t'])

print s.str.islower()

### isupper()

import pandas as pd

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t'])

print s.str.isupper()

### isnumeric()

import pandas as pd

s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t'])

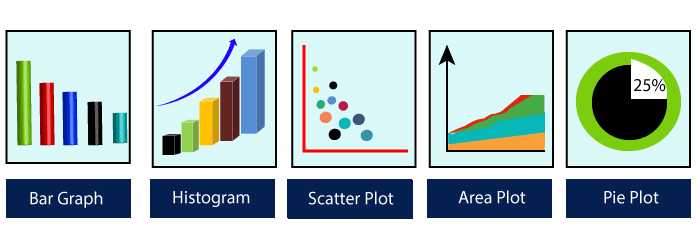
print s.str.isnumeric()

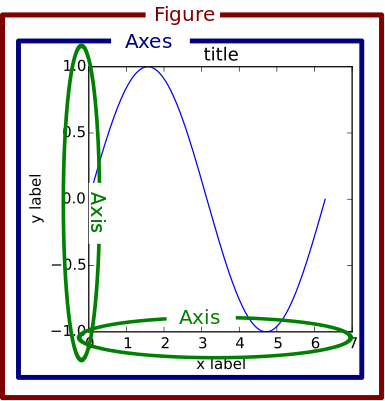
**MODULE4:**

**DATA VIS. USING MATPLOTLIB**

**Matplotlib** is a Python library which is defined as a multi-platform data visualization library built on Numpy array. It can be used in python scripts, shell, web application, and other graphical user interface toolkit.

The **John D. Hunter** originally conceived the matplotlib in **2002**. It has an active development community and is distributed under a **BSD-style license**. Its first version was released in 2003, and the latest **version 3.1.1** is released on **1 July 2019**.





* Visualize: We analyze the raw data, which means it makes complex data more accessible, understandable, and more usable. Tabular data representation is used where the user will look up a specific measurement, while the chart of several types is used to show patterns or relationships in the data for one or more variables.
* Analysis: Data analysis is defined as cleaning, inspecting, transforming, and modeling data to derive useful information. Whenever we make a decision for the business or in daily life, is by past experience. What will happen to choose a particular decision, it is nothing but analyzing our past. That may be affected in the future, so the proper analysis is necessary for better decisions for any business or organization.
* Document Insight: Document insight is the process where the useful data or information is organized in the document in the standard format.
* Transform Data Set: Standard data is used to make the decision more effectively.

### Why need data visualization?

Data visualization can perform below tasks:

* It identifies areas that need improvement and attention.
* It clarifies the factors.
* It helps to understand which product to place where.
* Predict sales volumes.

### Benefit of Data Visualization

Here are some benefits of the data visualization, which helps to make an effective decision for the organizations or business:

**1. Building ways of absorbing information**

Data visualization allows users to receive vast amounts of information regarding operational and business conditions. It helps decision-makers to see the relationship between multi-dimensional data sets. It offers new ways to analyses data through the use of maps, fever charts, and other rich graphical representations.

Visual data discovery is more likely to find the information that the organization needs and then end up with being more productive than other competitive companies.

**2. Visualize relationship and patterns in Businesses**

The crucial advantage of data visualization is that it is essential to find the correlation between operating conditions and business performance in today's highly competitive business environment.

The ability to make these types of correlations enables the executives to identify the root cause of the problem and act quickly to resolve it.

Suppose a food company is looking their monthly customer data, and the data is presented with bar charts, which shows that the company's score has dropped by five points in the previous months in that particular region; the data suggest that there's a problem with customer satisfaction in this area.

**3. Take action on the emerging trends faster**

Data visualization allows the decision-maker to grasp shifts in customer behavior and market conditions across multiple data sets more efficiently.

Having an idea about the customer's sentiments and other data discloses an emerging opportunity for the company to act on new business opportunities ahead of their competitor.

**4. Geological based Visualization**

Geo-spatial visualization is occurred due to many websites providing web-services, attracting visitor's interest. These types of websites are required to take benefit of location-specific information, which is already present in the customer details.

### Installing Matplotlib

pip install matplotlib

# Functions used for different plots

The following methods are used to draw diffrent types of graphs in matplot library

### 

### Working with Pyplot

The matplotlib.pyplot is the collection command style functions that make matplotlib feel like working with MATLAB. The pyplot functions are used to make some changes to figure such as create a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot including labels, etc.

It is good to use when we want to plot something quickly without instantiating any figure or Axes.

While working with matplotlib.pyplot, some states are stored across function calls so that it keeps track of the things like current figure and plotting area, and these plotting functions are directed to the current axes.

The pyplot module provide the plot() function which is frequently use to plot a graph. Let's have a look on the simple example:

We can pass the arbitrary number of arguments to the plot(). For example, to plot x versus y, we can do this following way:

from matplotlib import pyplot as plt

plt.plot([1,2,3,4,5],[1,4,9,16,25])

plt.ylabel("y axis")

plt.xlabel('x axis')

plt.show()

**Formatting the style of the plot**

There is an optional third argument, which is a format string that indicates the color and line type of the plot. The default format string is '**b-**'which is the solid blue as you can observe in the above plotted graph. Let's consider the following example where we plot the graph with the red circle.

from matplotlib import pyplot as plt

plt.plot([1, 2, 3, 4,5], [1, 4, 9, 16,25], 'ro')

plt.axis([0, 6, 0, 20])

plt.show()

**Example format String**

|  |  |
| --- | --- |
| **'b'** | Using for the blue marker with default shape. |
| **'ro'** | Red circle |
| **'-g'** | Green solid line |
| **'--'** | A dashed line with the default color |
| **'^k:'** | Black triangle up markers connected by a dotted line |

The matplotlib supports the following color abbreviation:

|  |  |  |  |
| --- | --- | --- | --- |
| **Character** | | **Color** | |
| **'b' Blue** |  | |
| **'g'** | Green | |
| **'r'** | Red | |
| **'c'** | Cyan | |
| **'m'** | Magenta | |
| **'y'** | Yellow | |
| **'k'** | Black | |
| **'w'** | White | |

**Plotting with categorical variables**

Matplotlib allows us to pass categorical variables directly to many plotting functions: consider the following example

from matplotlib import pyplot

names = ['Abhishek', 'Himanshu', 'Devansh']

marks= [87,50,98]

plt.figure(figsize=(9,3))

plt.subplot(131)

plt.bar(names, marks)

plt.subplot(132)

plt.scatter(names, marks)

plt.subplot(133)

plt.plot(names, marks)

plt.suptitle('Categorical Plotting')

plt.show()

**What is subplot()**

The subplots() function takes three arguments that describes the layout of the figure.

The layout is organized in rows and columns, which are represented by the first and second argument.

The third argument represents the index of the current plot.

plt.subplot(1, 2, 1)  
#the figure has 1 row, 2 columns, and this plot is the first plot.

plt.subplot(1, 2, 2)  
#the figure has 1 row, 2 columns, and this plot is the second plot.

So, if we want a figure with 2 rows an 1 column (meaning that the two plots will be displayed on top of each other instead of side-by-side), we can write the syntax like this:

### Example#1

Draw 2 plots on top of each other:

import matplotlib.pyplot as plt  
import numpy as np  
  
#plot 1:  
x = np.array([0, 1, 2, 3])  
y = np.array([3, 8, 1, 10])  
  
plt.subplot(2, 1, 1)  
plt.plot(x,y)  
  
#plot 2:  
x = np.array([0, 1, 2, 3])  
y = np.array([10, 20, 30, 40])  
  
plt.subplot(2, 1, 2)  
plt.plot(x,y)  
  
plt.show()

### Example#2: Draw 6 plots:

import matplotlib.pyplot as plt  
import numpy as np  
  
x = np.array([0, 1, 2, 3])  
y = np.array([3, 8, 1, 10])  
  
plt.subplot(2, 3, 1)  
plt.plot(x,y)  
  
x = np.array([0, 1, 2, 3])  
y = np.array([10, 20, 30, 40])  
  
plt.subplot(2, 3, 2)  
plt.plot(x,y)  
  
x = np.array([0, 1, 2, 3])  
y = np.array([3, 8, 1, 10])  
  
plt.subplot(2, 3, 3)  
plt.plot(x,y)  
  
x = np.array([0, 1, 2, 3])  
y = np.array([10, 20, 30, 40])  
  
plt.subplot(2, 3, 4)  
plt.plot(x,y)  
  
x = np.array([0, 1, 2, 3])  
y = np.array([3, 8, 1, 10])  
  
plt.subplot(2, 3, 5)  
plt.plot(x,y)  
  
x = np.array([0, 1, 2, 3])  
y = np.array([10, 20, 30, 40])  
  
plt.subplot(2, 3, 6)  
plt.plot(x,y)  
  
plt.show()

## Adding Title to Each Plot

You can add a title to each plot with the title() function:

### Example#3: 2 plots, with titles:

import matplotlib.pyplot as plt  
import numpy as np  
  
#plot 1:  
x = np.array([0, 1, 2, 3])  
y = np.array([3, 8, 1, 10])  
  
plt.subplot(1, 2, 1)  
plt.plot(x,y)  
plt.title("SALES")  
  
#plot 2:  
x = np.array([0, 1, 2, 3])  
y = np.array([10, 20, 30, 40])  
  
plt.subplot(1, 2, 2)  
plt.plot(x,y)  
plt.title("INCOME")  
plt.show()

## Super Title

You can add a title to the entire figure with the suptitle() function:

### Example#4: Add a title for the entire figure:

import matplotlib.pyplot as plt  
import numpy as np  
  
#plot 1:  
x = np.array([0, 1, 2, 3])  
y = np.array([3, 8, 1, 10])  
  
plt.subplot(1, 2, 1)  
plt.plot(x,y)  
plt.title("SALES")  
  
#plot 2:  
x = np.array([0, 1, 2, 3])  
y = np.array([10, 20, 30, 40])  
  
plt.subplot(1, 2, 2)  
plt.plot(x,y)  
plt.title("INCOME")  
  
plt.suptitle("MY SHOP")  
plt.show()

**Creating different types of graphs**

### 1. Line graph

The line graph is one of charts which shows information as a series of the line. The graph is plotted by the plot() function. The line graph is simple to plot; let's consider the following example:

from matplotlib import pyplot as plt

x = [4,8,9]

y = [10,12,15]

plt.plot(x,y)

plt.title("Line graph")

plt.ylabel('Y axis')

plt.xlabel('X axis')

plt.show()

We can customize the graph by importing the style module. The style module will be built into a matplotlib installation. It contains the various functions to make the plot more attractive. In the below program, we are using the style module:

from matplotlib import pyplot as plt

from matplotlib import style

style.use('ggplot')

x = [16, 8, 10]

y = [8, 16, 6]

x2 = [8, 15, 11]

y2 = [6, 15, 7]

plt.plot(x, y, 'r', label='line one', linewidth=5)

plt.plot(x2, y2, 'm', label='line two', linewidth=5)

plt.title('Epic Info')

fig = plt.figure()

plt.ylabel('Y axis')

plt.xlabel('X axis')

plt.legend()

plt.grid(True, color='k')

plt.show()

In Matplotlib, the figure (an instance of class plt.Figure) can be supposed of as a single container that consists of all the objects denoting axes, graphics, text, and labels.

### Example-3

import numpy as np

import matplotlib.pyplot as plt

fig = plt.figure()

ax = plt.axes()

x = np.linspace(0, 10, 1000)

ax.plot(x, np.sin(x))

The matplotlib provides the **fill\_between()** function which is used to fill area around the lines based on the user defined logic.

### Example-4

import numpy as np

import matplotlib.pyplot as plt

fig = plt.figure()

ax = plt.axes()

x = np.linspace(0, 10, 1000)

ax.plot(x, np.sin(x))

import matplotlib.pyplot as plt

import numpy as np

x = np.arange(0.0, 2, 0.01)

y1 = np.sin(2 \* np.pi \* x)

y2 = 1.2 \* np.sin(4 \* np.pi \* x)

fig, ax = plt.subplots(1, sharex=True)

ax.plot(x, y1, x, y2, color='black')

ax.fill\_between(x, y1, y2, where=y2 >= y1, facecolor='blue', interpolate=True)

ax.fill\_between(x, y1, y2, where=y2 <= y1, facecolor='red', interpolate=True)

ax.set\_title('fill between where')

**2. Bar graphs**

Bar graphs are one of the most common types of graphs and are used to show data associated with the categorical variables. Matplotlib provides a **bar()** to make bar graphs which accepts arguments such as: categorical variables, their value and color.

from matplotlib import pyplot as plt

players = ['Virat','Rohit','Shikhar','Hardik']

runs = [51,87,45,67]

plt.bar(players,runs,color = 'green')

plt.title('Score Card')

plt.xlabel('Players')

plt.ylabel('Runs')

plt.show()

Another function **barh()** is used to make horizontal bar graphs.

from matplotlib import pyplot as plt

players = ['Virat','Rohit','Shikhar','Hardik']

runs = [51,87,45,67]

plt.barh(players,runs, color = 'green')

plt.title('Score Card')

plt.xlabel('Players')

plt.ylabel('Runs')

plt.show()

Let's have a look on the other example using the **style()** function:

from matplotlib import pyplot as plt

from matplotlib import style

style.use('ggplot')

x = [5,8,10]

y = [12,16,6]

x2 = [6,9,11]

y2 = [7,15,7]

plt.bar(x, y, color = 'y', align='center')

plt.bar(x2, y2, color='c', align='center')

plt.title('Information')

plt.ylabel('Y axis')

plt.xlabel('X axis')

Similarly to vertical stack, the bar graph together by using the bottom argument and define the bar graph, which we want to stack below and its value.

from matplotlib import pyplot as plt

import numpy as np

countries = ['USA', 'India', 'China', 'Russia', 'Germany']

bronzes = np.array([38, 17, 26, 19, 15])

silvers = np.array([37, 23, 18, 18, 10])

golds = np.array([46, 27, 26, 19, 17])

ind = [x for x, \_ in enumerate(countries)]

plt.bar(ind, golds, width=0.5, label='golds', color='gold', bottom=silvers+bronzes)

plt.bar(ind, silvers, width=0.5, label='silvers', color='silver', bottom=bronzes)

plt.bar(ind, bronzes, width=0.5, label='bronzes', color='#CD853F')

plt.xticks(ind, countries)

plt.ylabel("Medals")

plt.xlabel("Countries")

plt.legend(loc="upper right")

plt.title("2019 Olympics Top Scorers")

### Enumerate() method adds a counter to an iterable and returns it in a form of enumerating object. This enumerated object can then be used directly for loops or converted into a list of tuples using the list() method.

### 3. Pie Chart

A pie chart is a circular graph that is broken down in the segment or slices of pie. It is generally used to represent the percentage or proportional data where each slice of pie represents a particular category. Let's have a look at the below example:

from matplotlib import pyplot as plt

# Pie chart, where the slices will be ordered and plotted counter-clockwise:

Players = 'Rohit', 'Virat', 'Shikhar', 'Yuvraj'

Runs = [45, 30, 15, 10]

explode = (0.1, 0, 0, 0)  # it "explode" the 1st slice

fig1, ax1 = plt.subplots()

ax1.pie(Runs, explode=explode, labels=Players, autopct='%1.1f%%',

        shadow=True, startangle=90)

ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.

plt.show()

### 4. Histogram

First, we need to understand the difference between the bar graph and histogram. A histogram is used for the distribution, whereas a bar chart is used to compare different entities. A histogram is a type of bar plot that shows the frequency of a number of values compared to a set of values ranges.

**For example** we take the data of the different age group of the people and plot a histogram with respect to the bin. Now, bin represents the range of values that are divided into series of intervals. Bins are generally created of the same size.

from matplotlib import pyplot as plt

from matplotlib import pyplot as plt

population\_age = [21,53,60,49,25,27,30,42,40,1,2,102,95,8,15,105,70,65,55,70,75,60,52,44,43,42,45]

bins = [0,10,20,30,40,50,60,70,80,90,100]

plt.hist(population\_age, bins, histtype='bar', rwidth=0.8)

plt.xlabel('age groups')

plt.ylabel('Number of people')

plt.title('Histogram')

plt.show()

Let's consider the another example of plotting histogram:

from matplotlib import pyplot as plt

# Importing Numpy Library

import numpy as np

plt.style.use('fivethirtyeight')

mu = 50

sigma = 7

x = np.random.normal(mu, sigma, size=200)

fig, ax = plt.subplots()

ax.hist(x, 20)

ax.set\_title('Historgram')

ax.set\_xlabel('bin range')

ax.set\_ylabel('frequency')

fig.tight\_layout()

plt.show()

### 5. Scatter plot

The scatter plots are mostly used for comparing variables when we need to define how much one variable is affected by another variable. The data is displayed as a collection of points. Each point has the value of one variable, which defines the position on the horizontal axes, and the value of other variable represents the position on the vertical axis.

Let's consider the following simple example:

**Example-1:**

from matplotlib import pyplot as plt

from matplotlib import style

style.use('ggplot')

x = [5,7,10]

y = [18,10,6]

x2 = [6,9,11]

y2 = [7,14,17]

plt.scatter(x, y)

plt.scatter(x2, y2, color='g')

plt.title('Epic Info')

plt.ylabel('Y axis')

plt.xlabel('X axis')

plt.show()

**Example-2**

import matplotlib.pyplot as plt

x = [2, 2.5, 3, 3.5, 4.5, 4.7, 5.0]

y = [7.5, 8, 8.5, 9, 9.5, 10, 10.5]

x1 = [9, 8.5, 9, 9.5, 10, 10.5, 12]

y1 = [3, 3.5, 4.7, 4, 4.5, 5, 5.2]

plt.scatter(x, y, label='high income low saving', color='g')

plt.scatter(x1, y1, label='low income high savings', color='r')

plt.xlabel('saving\*100')

plt.ylabel('income\*1000')

plt.title('Scatter Plot')

plt.legend()

plt.show()

### 6. 3D graph plot

Matplotlib was initially developed with only two-dimension plot. Its 1.0 release was built with some of three-dimensional plotting utilities on top of two-dimension display, and the result is a convenient set of tools for 3D data visualization.

Three-dimension plots can be created by importing the **mplot3d** toolkit, include with the main Matplotlib installation:

from mpl\_toolkits import mplot3d

When this module is imported in the program, three-dimension axes can be created by passing the keyword **projection='3d'** to any of the normal axes creation routines:

Let's see the simple 3D plot

**Example-1:**

from mpltoolkits import mplot3d

import numpy as np

import matplotlib.pyplot as plt

fig = plt.figure()

ax = plt.axes(projection='3d')\_

**Example-2:**

from mpl\_toolkits import mplot3d

import numpy as np

import matplotlib.pyplot as plt

height = np.array([100,110,87,85,65,80,96,75,42,59,54,63,95,71,86])

weight = np.array([105,123,84,85,78,95,69,42,87,91,63,83,75,41,80])

plt.scatter(height,weight)

fig = plt.figure()

ax = plt.axes(projection='3d')

# This is used to plot 3D scatter

ax.scatter3D(height,weight)

plt.title("3D Scatter Plot")

plt.xlabel("Height")

plt.ylabel("Weight")

plt.title("3D Scatter Plot")

plt.xlabel("Height")

plt.ylabel("Weight")

plt.show()

**Example-3**

import matplotlib as mpl

from mpl\_toolkits.mplot3d import Axes3D

import numpy as np

import matplotlib.pyplot as plt

mpl.rcParams['legend.fontsize'] = 10

fig = plt.figure()

ax = fig.gca(projection='3d')

theta1 = np.linspace(-4 \* np.pi, 4 \* np.pi, 100)

z = np.linspace(-2, 2, 100)

r = z\*\*2 + 1

x = r \* np.sin(theta1)

y = r \* np.cos(theta1)

ax.plot3D(x, y, z, label='parametric curve', color = 'red')

ax.legend()

plt.show()

## Important functions of Matplotlib

|  |  |
| --- | --- |
| **Functions** | **Description** |
| plot(x-axis value, y-axis-values) | It is used to plot a simple line graph with x-axis  value against the y-axis values. show().  It is used to display the graph. |
| title("string") | It is used to set the title of the plotted graph as  specified by the string. |
| xlabel("string") | It is used to set the label for the x-axis as specified  by the string. |
| ylabel("string") | It is used to set the label for y-axis as specified  by the string. |
| figure() | It is used to control a figure level attributes. |
| subplots(nrows,ncol,index) | It is used to add a subplot to recent figure. |
| subtitle("string") | It adds a common title to the plotted graph specified  by the string. |
| subplots(nrows,ncols,figsize) | It provides the simple way to create subplot, in a  single call and returns a tuple of a figure and  number of axes. |
| set\_title("string") | It is an axes level method which is used to set the  title of the subplots. |
| bar(categorical variables, values, color) | It is used to create a vertical bar graph. |
| barh(categorical variables, values, color) | It is used to create horizontal bar graphs. |
| legend(loc) | It is used to make a legend of the graph. |
| xtricks(index, categorical variables) | It is used to set or get the current tick locations  labels of the x-axis. |
| pie(value, categorical variables) | It is used to create a pie chart. |
| hist(value, number of bins) | It is used to create a histogram. |
| xlim(start value, end value) | It is used to set the limit of values of the x-axis. |
| ylim(start value, end value) | It is used to set the limit of values of the y-axis. |
| scatter(x-axis values, y-axis values) | It is used to plots a scatter plot with x-axis value  against the y-axis values. |
| axes() | It is used to add axes to the recent figure. |
| set\_xlabel("string") | It is an axes level method which is used to set the  x-label of the plot specified as a string. |
| set\_ylabel("string") | It is used to set the y-label of the plot specified as  a string. |
| scatter3D(x-axis values, y-axis values) | It is used to plot a three-dimension scatter plot with  x- axis value against the y-axis. |
| plot3D(x-axis values, y-axis values) | It is used to plots a three-dimension line graph with  x- axis values against y-axis values. |