Introduction of python

Ch.1 Introduction of Basic Python

Python is an easy to learn. It is a widely used, high – level and powerful language, with applications in numerous areas including web programming, scripting, scientific computing, artificial intelligence, machine learning, deep learning etc.; it was initially designed by Guido van Rossum in 1991 and developed by Python Software Foundation. It has lots’ of features such as,

* It supports efficient high – level data – structure.
* It works with Object – Oriented Programming.
* It also works with web programming.
* It works with interpreter.
* Developed for emphasis on code readability.

So, Python is very powerful because its’ elegant syntax and dynamic typing together with its’ interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platform.

At present, two different versions of python available in market Python 2.7 and Python 3.X. Here, I am working with python 3.X. To work with Python first step is to install Python on wer system. Here I am giving we installation steps on Windows.

**Step – 1:** Download Python from <https://www.python.org/downloads/>

**Step – 2:** After Downloading, we need to install it into the system.

**Step – 3:** After installation, just set environment variables. (Here I am working with Windows OS) for that go to **Control Panel > System > Advanced System Settings**. It will open System Properties Dialogue box, go to **Advanced tab >** Click on **Environment Variables Button > Set System variables >** Select **Path** property click on Edit Button without removing path just Edit and paste path of Python **Scripts directory**.

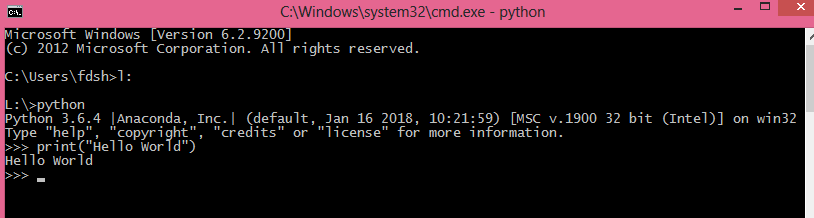
**Step – 4:** Then, just run IDLE, and it will open IDE for Python.

IDLE is a graphical python shell, on which we can run python syntax and debug short programs. And now start coding with python. It has a text editor also.

First of all, test with ***“Hello World”*** so, go to IDLE and just type print(“Hello World”) and it will show we the output as, **Hello World**



**Fig. 1 Print Hello World using IDLE**



**Fig. 2 Print Hello World using cmd**

**Note: we also have an editor like, Jupyter notebook, Spyder, qtconsole etc. And we can get all the IDEs from Anaconda Navigator. We can download Anaconda from** [**https://www.anaconda.com/download/**](https://www.anaconda.com/download/) **we can download latest version of Anaconda from the link. (At present Anaconda 5.2 and Python 3.6 and Python 2.7 both installer are available.)**

**For Linux,** Python comes bundled with Linux. So, no need to install.

**print():** print function is used to print statement in version 3.X. In early version i.e. version 2.X it is used as a statement.

Some examples with Syntax,

print “Hello, World” Python version 2.X

print(“Hello, World”) Python version 3.X print “Addition is:” , 2+2 Python version 2.X print(“Addition is:” , 2+2) Python version 3.X

print x, Python 2.X comma generates new line

print(x, end=” ”) Python 3.X appends a space instead of new line

print Python 2.X generates new line

print() Python 3.X generates new line, call function There are lots of other ways to work with print function.

### Basic Elements of Python

There are lots of **data types** available in python. Let’s see the different basic data types. Such as,

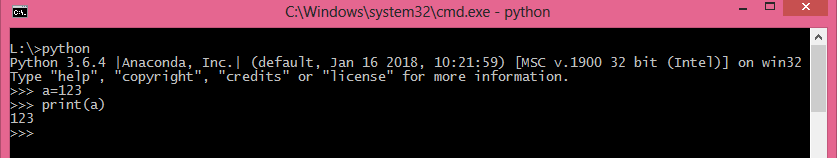
* Boolean
* Integer
* Long
* Float
* String
* List
* Object
* None

**Variables:** means a named memory location, in which we can store value(s).

**Variables** in python work differently that is no need to declare a variable in python and their data types are inferred from the assignment statement. But variable naming rules are same as other programming languages, i.e. name of variable starts with alphabet or underscore. And name of variables are case sensitive.

Examples,

A=123 #Here, **A** is a variable with integer data type a=45.56677 #Here, **a** is a variable with float data type name=”Python” #Here, **name** is a variable with string data type



**Fig. 3 work with variables**

**Expressions:** we can make any expression directly in python or even we can work with script also by using IDLE. To create expressions, we need to use different operators.



**Fig. 4 Expressions**

**Operators:** lots’ of operators are available in python to handle different types of operations.

* **Arithmetic Operator:** to perform any mathematical calculation arithmetic operators are used.

o **+** To perform Addition operation

* + **–** To perform Subtraction operation
  + **\*** To perform Multiplication operation
  + **/** To perform Division operation (returns float data as answer)
  + **//** To perform Division operation (returns floor data as answer)
  + **%** To perform modulus operation (get reminder)

#### Example:

#Arithmetic operators a=5

b=2 print(a+b) print(a-b) print(a\*b) print(a/b) print(a//b) print(a%b)

* **Relational Operator:** is used to check the relation between two operands. In other words it is used to compare two values.

o **==** To check operand is same or **equal to** another operand

* **>** To check which operand is **greater than** another operand
* **>=** To check operands is **greater than or equal to** another
* **<** To check whether the operand is **less than** another
* **<=** To check whether the operand is **less than or equal to** another
* **!=** To check whether operand is **not equal to** another operand As per the relation between two operands it returns either True or False.

#### Example:

#Relational operators a=5

b=2

print(a==b) #equal to print(a>b) #greater than

print(a>=b) #greater than or equal to print(a<b) #less than

print(a<=b) #less than or equal to print(a!=b) #not equal

* **Logical Operators:** to perform logical operations on different operands or expressions logical operator is used.
* **and** Logical and returns true if both operands are true
* **or** Logical or returns true if any one operand is true
* **not** Logical not returns true if operand is false

#### Example,

#Logical operators print(a and b) print(a or b) print(not a)

* **Bitwise Operator:** performs bit by bit operation. o **&** Bitwise AND
* **|** Bitwise OR
* **~** Bitwise NOT
* **^** Bitwise XOR
* **>>** Bitwise right shift
* **<<** Bitwise left shift

#### Explanation:

|  |  |  |
| --- | --- | --- |
| Operator | Example | Description |
| **& Bitwise AND** | X=X&Y | X=5 0000 0101  Y=3 0000 0011  X&Y= 0000 0001 X=X&Y X=1 |
| **| Bitwise OR** | X=X|Y | X=5 0000 0101  Y=3 0000 0011  X|Y= 0000 0111 X=X|Y X=7 |
| **^ Bitwise XOR** | X=X^Y | X=5 0000 0101  Y=3 0000 0011  X^Y= 0000 0110 X=X^Y X=6 |
| **~ Bitwise NOT** | X=~X | X=5 0000 0101  ~X= 1111 1010 X=~X X=-6 |
| **<< Bitwise Left Shift** | X=X<<2 | X=5 0000 0101  X<<2 0001 0100 X=X<<2 X=20 |
| **>> Bitwise**  **Right Shift** | X=X>>2 | X=5 0000 0101  X>>2 0000 0001 X=X>>2 X=1 |

**Example,**

#Bitwise Operator X=5

Y=3

print(X&Y) print(X|Y) print(X^Y) print(~X) print(X<<2) print(X>>2)

* **Assignment Operator:** to assign any value to a variable, assignment operator is used. o **=** Assign a value of right side value or expression to left side operand.
* Short hands: (Arithmetic Assignment Operator)

 **+=** Add right side value or operand to left side value or operand and assign it to left operand

* + **-=** Subtract right side value or operand to left side value or operand and assign it to left operand
  + **\*=** Multiply right side value or operand to left side value or operand and assign it to left operand
  + **/=** Divide left side value or operand to right side value or operand and assign it to left operand returns float value
  + **//=** Divide left side value or operand to right side value or operand and assign it to left operand returns floor value
  + **%=** Divide left operand or value with right side value or operand and returns reminder and store it to left side operand
  + **\*\*=** Calculate the exponent and assign the value in left operand
  + **&=** Perform Bitwise AND operation, and assign value to left operand
  + **|=** Perform Bitwise OR operation, and assign value to left operand
  + **^=** Perform Bitwise XOR operation, and assign value to left operand
  + **>>=** Shift bits right side and assign value to left operand
  + **<<=** Shift bits left side and assign value to right operand
* **Special Operators:** there are other different types of special operators.
* **Identity operator:** there are two identity operators **is** and **is not**. These both are used to check operands are identical or not. A value is of a certain class or type. They are usually used to determine the type of data a certain variable contains,
  + **is** Evaluates to true if the variables on either side of operator point to the same object and false otherwise.
  + **is not** Evaluates to false if the variables on either side of operator point to the same object and true otherwise.

#### Example,

#Identity operator X=5

Y=5

print(X is Y) print(type(X) is int) a=5

b=5.0

print(a is not b) print(type(a) is not int) print(type(b) is not int)

* **Membership operator:** there are two membership operators **in** and **not in**. These both operators are used to check the membership of a value. It checks for membership in a sequence, such as strings, lists, tuples.
  + **in** The in operator is used to check if a value exists in an sequence or not. Evaluates to true if it finds a variable in the specified sequence and false otherwise.
  + **not in** Evaluate to true if it does not finds a variable in the specified sequence and false otherwise.

#### Example,

#Membership operators X=[1,2,3]

print(3 in X) print(4 not in X)

**Comments:**

# to create single line comment # sign is used.

#### For example, #this is single line comment.

‘’’ for multi line comment three **single inverted commas** are used, but we need to start it and stop the comment block.

#### For example, ‘’’this is multi line comment’’’

“ ” ” for documentation three double inverted commas are used, but we need to start and end it.

#### For example, """Documentation Comment Author: ABC DEF

**Version:1.0.0"""**

**Decision Making Statements:**

In real life we need to take some decisions on daily basis, same situations arise in programming also so we need to make some decisions and based on these decision we will execute the block.

Decision making statements in programming languages decides the direction of program execution. Decision making statements available in python are:

* if statement
* if\_else statement
* nested if statement
* if\_elif ladder (Nested if else or Multiple if\_else)

**if statement:**

It is very simple decision making statement. It is used to check whether the condition is true or false. If condition is true, statement will execute otherwise not return anything.

#### Syntax:

if(condition):

#statement block

Here, if condition is true statement block will execute.

#### Flowchart,

**Example,**

**Simple If Statement**

#simple if statement X=5

if(X%2!=0):

print("Odd Number")

if this condition is true, print the statement. In python, for any statement **indentation is used**. So, here after if statement print statement is automatically intended.

#### if\_else statement:

Simple if is very simple statement and if condition is true it will execute statement and if condition is false, it will not return any answer. So, there is a new statement in both situation it will return an answer.

#### Syntax:

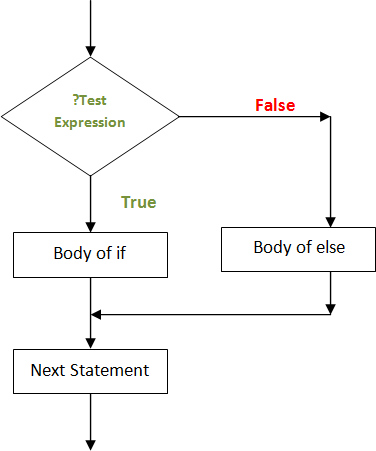
if(condition):

#statement block

else: Flowchart,

#### Example,

#statement block



**if\_else statement**

#if\_else statement X=4

if(X%2!=0):

print("Odd Number") else:

print("Even Number")

If the condition is true, it will print the message Odd Number else it will print message Even Number.

#### Nested if statement:

When we have different condition and based on one condition we need to check one or more conditions and then take any decision. At that time use nested condition. So, Nested if statement is used to test one or more condition. In other words, if statement inside another if statement.

#### Syntax:

if(condition1):

if(condition2):

#statement block

else:

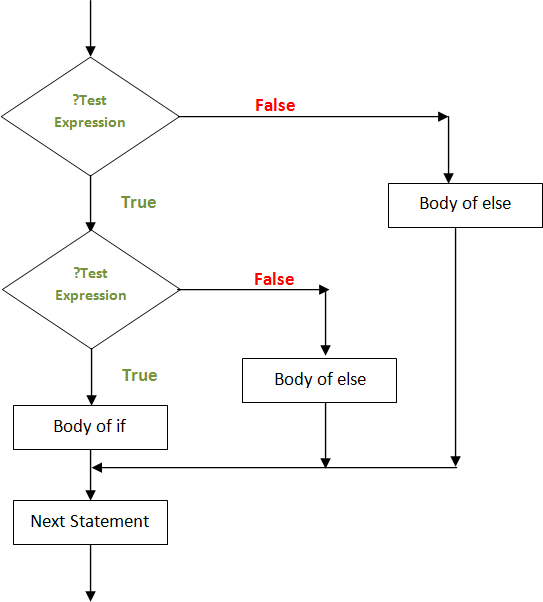
else:

#statement block

if(condition3):

else:

#statement block #statement block

Here, else part is optional. When condition1 is true then it will check for condition2 and it executes statement block of condition2. If only condition2 is false then it executes statement block of else part of condition2. And if condition1 is false and we use else part at that time else part will execute. Flowchart:

**Nested If statement**

#### Note: flowchart of nested if statement is depends on problem, there may be a condition and if it is true, we need to check one more condition and if the main condition is false then even we have one more condition, so based on that conditions we need to execute body of either if or else. And flowchart varies.

**Example,**

#Nested if statement X=50

Y=50 if(X!=Y):

if(X>Y):

print(X," is greater than ",Y) else:

print(Y," is greater than ",X)

else:

print("Both are equal")

**if\_elif statement:**

This statement is also used to check multiple conditions. And it is also known as multiple if else or else if ladder. Because when we have multiple conditions at that time we need to take only one decision, and all other condition is omitted if none condition is true, else part will be executed.

#### Syntax:

if(condition - 1):

#statement block – 1 elif(condition - 2):

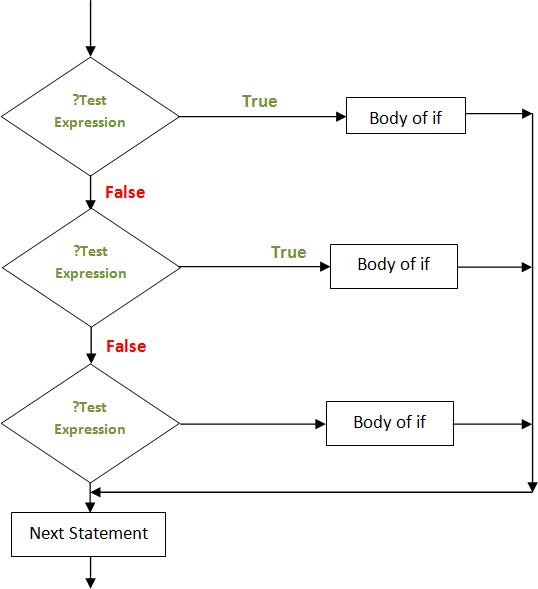
#statement block – 2 elif(condition - N):

#statement block - N

else:

#statement block

#### Flowchart:



**Example,**

**If\_elif statement**

#If elif ladder X=50

Y=32 Z=52

if(X>Y and X>Z):

print(X," is greater than ",Y," and ",Z) elif(Y>X and Y>Z):

print(Y," is greater than ",X," and ",Z) else:

print(Z," is greater than ",X," and ",Y)

#### Note: Unlike other programming languages, Python does not have switch case statement. Switch case statement replace with another statement.

**Looking Statements:**

To perform same task is boring thing for humans, but sometimes we have to perform same tasks multiple time. So, it becomes easy in computer science through looping statement. Like other programming languages, python supports different looping statements. Such as,

* While Loop
* For Loop
* Nested Loop

#### While Loop:

Like other programming languages python also supports while loop, it is one of the powerful looping statement, while loop is used when we don’t know about number of iterations in advance.

#### Syntax:

while condition:

#statement block

#### Example:

#While Loop Example count=1

while count<=3: print("Hello Python") count+=1

**For Loop:**

The for statement is used to generate loop iterations over the sequence of different values.

#### Syntax:

for <variable> in <sequence>: #statement block

#### Example:

#for loop example for i in 1,2,3,4,5:

print(i)

**range():** range is a function is used to iterate over a sequence of number, it is a built – in function. It generates a sequence of numbers.

#### Syntax:

* range(int range)
* range(int start,int end)
* range(int start, int end, step)

#### Example:

#for loop with range() for i in range(5):

print(i)

This example generates a sequence of number those are less than 5.

#### Example 2:

#for loop with range() for i in range(5,10):

print(i)

This example generates a sequence with starting element 5 and smaller than 10.

### break, continue and pass statements:

break and continue both are the flow breaking statements.

#### break statement:

This statement is used to break the flow of any statement.

#### Syntax:

break

#### Example:

#break statement

for i in range(1,10): if(i%2==0):

print(i) else:

break

**continue statement:**

This statement is used as flow breaking statement. But unlike break statement, it will continue the statement and if condition is true, it will skip the iteration.

#### Syntax:

continue

#### Example:

#continue statement for i in range(1,10):

if(i%2==0):

print(i) else:

continue

**pass statement:**

pass statement does nothing. When statement required syntactically, program requires no action.

#### Syntax:

pass

#### Example:

#pass statement

for i in range(1,10): if(i%2==0):

print(i) else:

pass

**else clause with loop:**

In looping statements, we can use else clause. It is executed when loop terminates when condition becomes false with while and when loop terminates because of empty list with for, but not break the loop by using break statement.

#### Example:

#for loop with else statement for i in range(1,10):

if(i%2!=0):

print(i)

else:

print("Bye...Bye")

**Nested Loop:**

The concept of nested loop is same as nested if. Nested loop means loop inside another loop.

#### Syntax:

for <variable> in <sequence>:

for <variable> in <sequence>: #statement block

#### Example:

#Nested Loop example for i in range(1,6):

for j in range(1,i+1): print(j,end='')

print()

**Input and Output:**

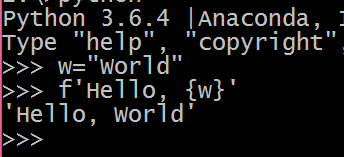
There are lots of different ways to present output with different formats.

#### Different fancy output format:

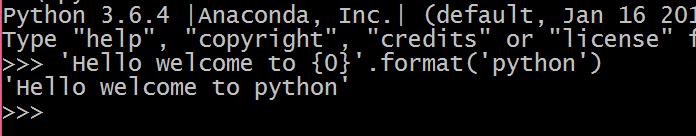
As we have already worked with expression statement and print() to print different output, we can write output through write() also. To write data to file object. The standard output is known as

**sys.stdout.** when we want to display output string sometimes we need some formatting also. So to control over formatting output there are several ways to generate formatting in wer output.

* Formatted string literals, in this method just use **f** or **F** before starting wer string. i.e. before quotation. We can write wer variable between curly braces. {<var>} this can refer we to variables or literals values. (This method is supports in later version of python after 3.6) **Example,**

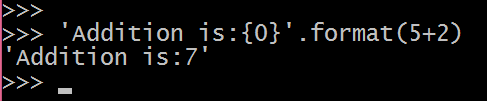


* str.format(), this method is used to give format to string. The string in which we call this method we need to pass arg delimited by {} this arg is working as replacement field. **Example 1,**

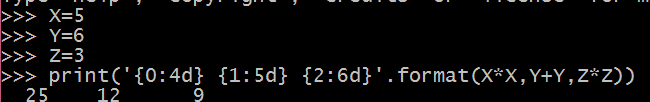


Here, {0} shows we the first argument that we will pass in format().

#### Example 2,



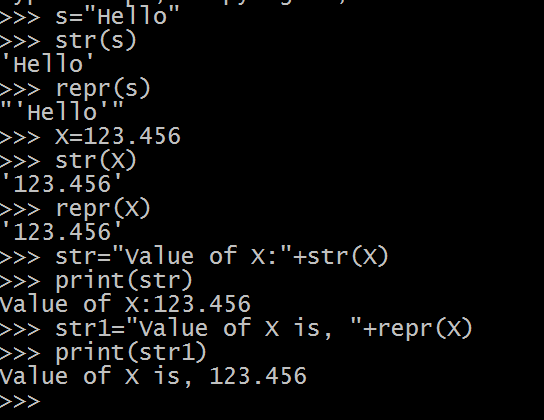
**Example 3,**

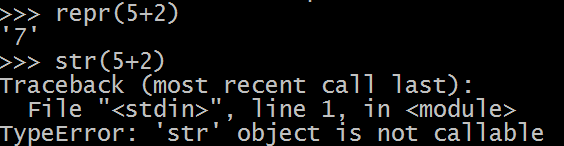


In this example {0:4d} means 0th argument with 4 digit padding. For {1:5d} means 1st argument with 5 digit padding. For {2:6d} means 2nd argument with 6 digit padding.

* The string module contains a Template class which has different methods to convert values to strings. str() or repr() these both functions are used to convert values to string.
* str(): this function is used to return representation of values. (Human Readable format)
* repr(): this function is used to generate representation of values. (interpreter readable format)

#### Some Examples,





We cannot write any expression in str(). But repr() performs well with different expression.

Different Justification methods:

While working with data, when we have digits and we want to give justification after converting data to string, python has three methods for justification.

* ljust(): for left justified.
* Syntax: string.ljust(n)
* rjust(): for right justified.
* Syntax: string.rjust(n)
* center(): for center justified.
* Syntax: string.center(n)

Here, n – number or width to adjust wer data. Say for example if we have 123 this is integral type of data. And it has 3 digits but if I want to set this data in 5 digits width.

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 |  |  |

With left justification:

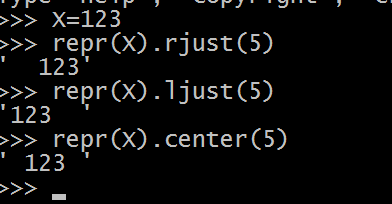
With right justification:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | 1 | 2 | 3 |

With center justification:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 |  |

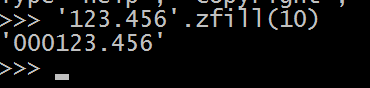
#### With example:



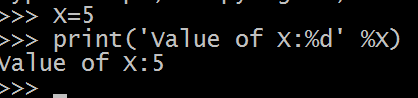
* **zfill()** function is used for padding numeric string on left with zeros.
* Syntax : string.zfill(n)

Here, n – is the number to display number of numbers in the string.

#### Example,



Python also supports % operator like C programming for string formatting.



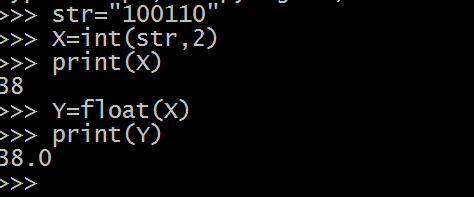
For different conversion of different values there are lots of different functions.

* int() – to convert any value in int data type. Syntax – int(n,base)

Here, n – is the value or number and base specifies the base of value or data if n is in string type.

* float() – to convert any value to float data type.

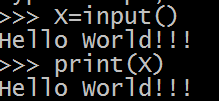
#### Example,



**Input functions:**

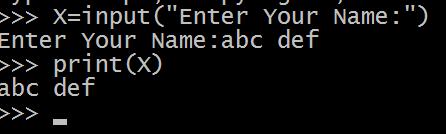
To read values from user python has different input functions. Such as input().

**input()** – this function is used to read string input values. Example,

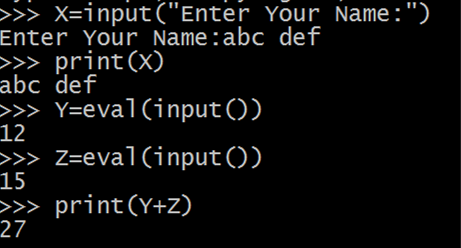


Here, X is a string variable. To get string input from user.

**input()** has an optional parameter, to prompt a message for user. For example,



Here, Enter wer name is a string that gives a proper message to the user.



Here, **eval()** is used to evaluate the value. So, it is consider as a number.

**Working with files:**

**open()** – this function is used to open file object. Syntax, open(filename,mode)

Here, filename – is the name of the file, which we want to open.

Mode – is used to define mode in which mode we want to open wer file. Different mode:

r – Open file will only be read w – Open file for write only

a – Open file for appending

r+ - Open file for reading and writing both b – Open file in binary mode

**write()** – this method is used to write string content in a file object. And it returns a number of characters written in the file.

Syntax, write(string)

**read()** – this method is used to read content from a file, which reads some quantity of data returns it as a string or byte. It returns empty string if file end of file has been reached.

Syntax, read(size)

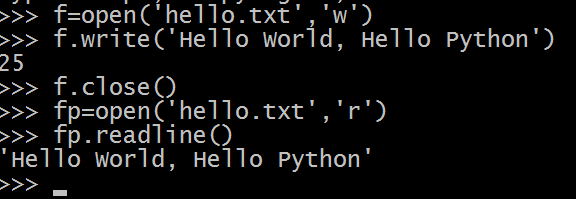
Here, size is an optional argument. If size is negative or omitted, entire data will be read. Otherwise it will read bytes as per the size.

**readline()** – this method is used to read a single line from a file. A new line character \n is left as at the end of string. This function returns empty string if end of file has been reached.

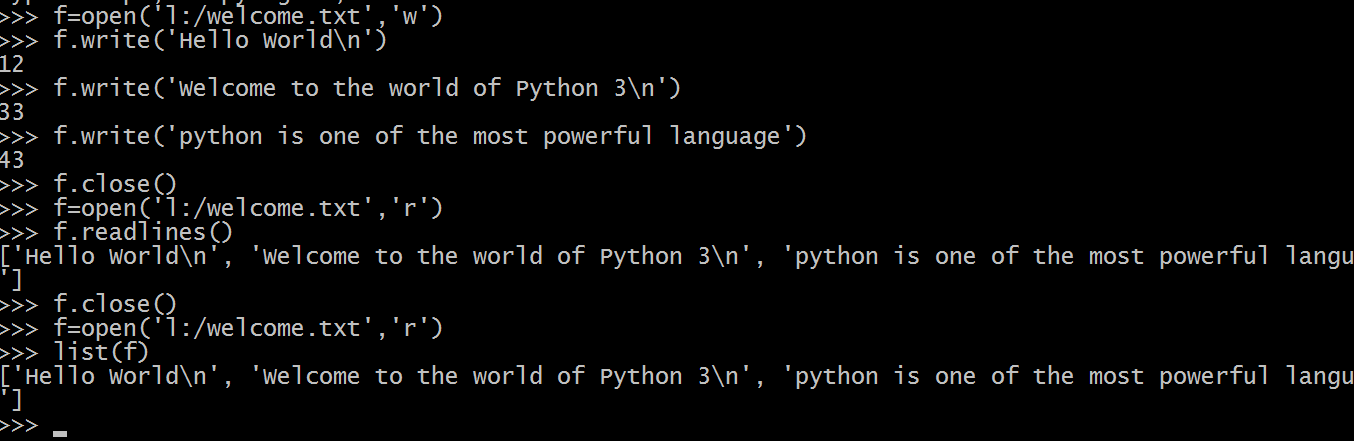
Syntax, readline()

**close()** – this function is used to close a file object. Syntax, close()

#### Example,

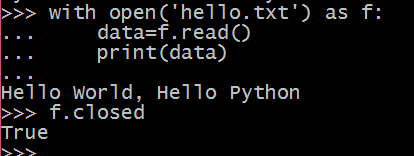


If we want to read entire file we can use **list(fileobject)** or **readlines()**. **Example,**



It is good thing while handling the file object, use with keyword. It is used as try\_except\_finally. with statement is used to encapsulate the execution of a block with methods.

#### Example,



Here in this example with clause open hello.txt file as f, f is a file pointer through which we read a file data by using read() and store it to data and print data. When we checked f i.e. wer file is closed it will return True. i.e. file is closed.

**tell()** – this function is used to get the current position of file object. It returns an integer number in the file represented by the number of bytes.

Syntax: f.tell()

**seek()** – this function is used to set file pointer position in the file. Syntax: f.seek(offset,from\_where)

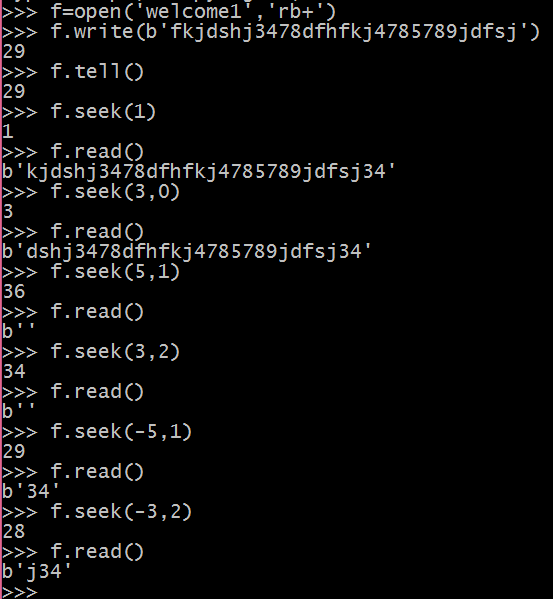
Offset – represents on which position we want to set wer file pointer. From\_where – represents

0 – if from\_where is 0 file pointer starts from beginning of file position. 1 – if from\_where is 1 file pointer starts from current position.

2 – if from\_where is 2 file pointer starts from end of file position.

And set wer file pointer. From\_where is optional. If from\_where argument is omitted by default 0 is set, so file pointer starts from 0 and set wer file pointer position as we have mentioned in offset.

#### Example,

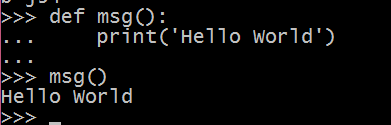


In text file from\_where option doesn’t work. It work with binary file as well.

## Functions:

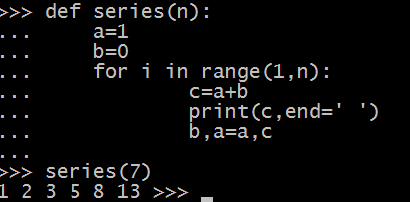
Python also supports functions. i.e. user defined function. The keyword ***def*** is used to define a function.

#### Example,



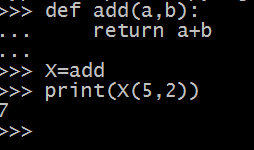
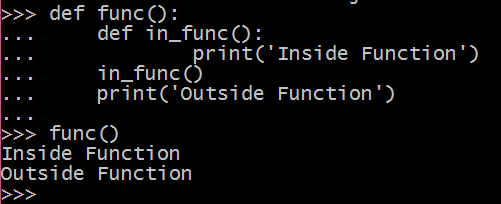
***def*** keyword must be followed by name of the function and the parenthesis list of formal parameter, it depends on type of function, parameters are not compulsory.

#### Example,



Here in python we can assign function to a variable also. It is known as *decorators*. Even we can define function within function, it is known as nested functions.

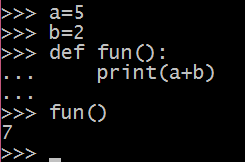
Example,

## Scope of Variables:

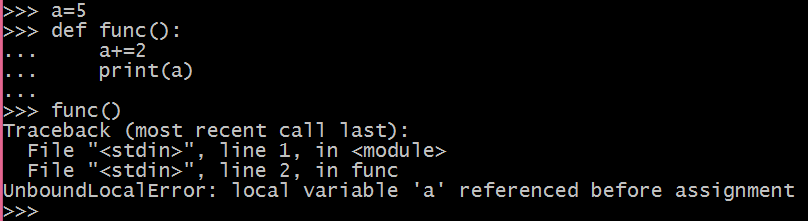
Generally, most variables are local. But as we know local variables work locally. So we cannot use it outside function or other functions. The scope of local variable is local where it is declared. If we want to make it global to use anywhere in the program, we need to declare it globally by using **Global** keyword.

For example,

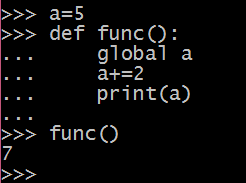


Here a = 5 and b = 2 both are global variables because it is declared outside of a function. Here a & b both are global variable so we can make addition operation possible.

Now,



In the above example, we cannot call global variable inside function, while trying to add 2 in variable a it shows an error local variable ‘a’ referenced before assignment. So to work with a in the above example, we have to use **global** keyword.



Python also supports nested function, it means function within function.

#wap to demonstrate nested function def funout():

X=5

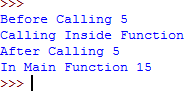
def funin():

global X X=15

print("Before Calling",X) print("Calling Inside Function") funin()

print("After Calling",X) funout()

print("In Main Function",X)

**Output:**

## Anonymous / Lambda function

Python also supports anonymous function. It means to define a function without name is known as anonymous function. It is also known as lambda function.

Syntax:

lambda argument : expression

Here, lambda is a keyword to create anonymous function. We can pass arguments after lambda keyword and after colon (:) what we want to execute through this function. i.e. function body in expression.

Example,

X=lambda a: a\*a

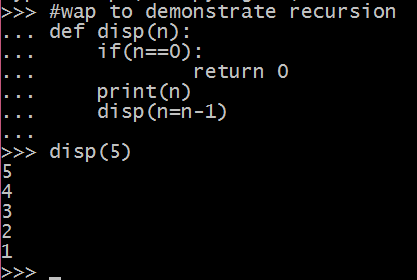
>>> X(5)

25 #Output

>>>

## Recursion:

Recursion is a technique provided by majority of programming languages, python also supports this technique. In simple words, recursion means function call itself is known as recursion.



## Modules:

Like other programming languages Python supports functionality in the form of module. **Definition:** python has a way to put definition in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module.

We can import different modules in a main module or in other modules. Python also supports user defined module. In which we can put different functionalities.

A module is a file containing python definitions and statements. The file name is a name of module and it has an extension .*py*, in python name of module we will get through global variable

name .

**User Defined Module :** To create wer own module, create the below function and save this file in

#### myModule.py

#wap to demonstrate def add(a,b):

return a+b

def sub(a,b):

return a-b

def prod(a,b): return a\*b

def div(a,b):

return a//b

Now, import this module. By using import keyword and name of wer file followed by import keyword and then we can call function of that file.

>>> import myModule

>>> print(myModule.add(5,2)) 7

>>>

We can import module with alias.

>>> import myModule as myMod

>>> myMod.div (5,2)

2

>>>

Even we can import module and use some functions only from that module,

from myModule import add, prod print(add(5,4))

print(prod(5,2))

>>> 9

10

>>>

We can get the name of module as string, by using global variable name

>>> import myModule

>>> myModule. name 'myModule'

We can also import module by using, and call it’s function and even we can set alias for that function.

>>> from myModule import add as ad

>>> ad(5,4) 9

>>>

When we import module at that time code is not run, but if we want to run that code, we can use module as script also. For that instance, we need to put a small code in wer module file.

if name ==’ main ’:

#code

In place of #code we need to put wer code if name ==’ main ’ means when we are in main module at that time we want to call functions from wer module and wer script i.e. we write in wer module. We want to run

So here my module looks like,

#wap to demonstrate module def add(a,b):

return a+b

def sub(a,b):

return a-b

def prod(a,b): return a\*b

def div(a,b):

return a//b

if name ==' main ': a=int(input("Enter first no:")) b=int(input('Enter second no:')) print(prod(a,b))

I want to call prod function in my main module so I take input from user when we are there in main and whatever the values entered by the user, we will get product of those values.

How to execute this module as script?

Just use python <moduleName>.py <arguments>

<arguments> is not mandatory if it is needed.

Here, in our example we need to call *python myModule.py* and we will get the output.

If we want to work with command line argument and execute the script, we need to *import sys*

module. Like

#wap to demonstrate command line argument with module. def add(a,b):

return a+b

def sub(a,b):

return a-b

def prod(a,b): return a\*b

def div(a,b):

return a//b

if name ==' main ': import sys

print(prod(int(sys.argv[1]),int(sys.argv[2])))

To execute the above code we need to call this script by the command *python myModule.py 9 2*

Here, 9 and 2 is wer sys.argv[1] and sys.argv[2] respectively.

## Standard Modules in python

Python has lots of built in standard modules. These all modules are built into python interpreter. These provide access to built in operations, either for efficiency or provide access to an operating system through system calls. These types of modules are configuration option which depends on the underlying platform. For example, *winreq* module supports with windows platform.

One of the most important module is *sys*, which comes built in with python interpreter.

The variables *sys.ps1* and *sys.ps2* defines the strings, which are used as primary and secondary prompt.

>>> import sys

>>> sys.ps1 '>>> '

>>> sys.ps2 '... '

>>>

Variable *sys.path* returns list of strings that determines interpreter’s search path for modules. It is initialized to a default path taken from the environment variable PYTHONPATH, if this variable is not set, it takes default path from built in path.

>>> import sys

>>> sys.path

['', 'C:\\Windows\\SYSTEM32\\python34.zip', 'L:\\Python34\\DLLs', 'L:\\Python34\\lib', 'L:\\Python34', 'L:\\Python34\\lib\\site-packages']

>>>

We can modify it by using following command:

>>> sys.path.append('l:\\python34\\scripts')

>>> sys.path

['', 'C:\\Windows\\SYSTEM32\\python34.zip', 'L:\\Python34\\DLLs', 'L:\\Python34\\lib', 'L:\\Python34', 'L:\\Python34\\lib\\site-packages', 'l:\\python34\\scripts']

>>>

## dir() function

dir() is used to find out names a module defines. It returns string.

>>> import myModule

>>> dir(myModule)

[' builtins ', ' cached ', ' doc ', ' file ', ' loader ',

' name ', ' package ', ' spec ', 'add', 'div', 'prod', 'sub']

>>>

It returns list of functions and variables those are defined by module. In the above example, dir() displays all variables and function names of *myModule* module, when we have multiple module imported at that time we can use this syntax.

>>> dir()

[' builtins ', ' doc ', ' loader ', ' name ', ' package ', ' spec ', 'myModule']

>>>

Even we can use dir() without argument if there is only one imported module.

## Packages

Packages are used to give structure to pythons’ different modules. It works as namespace. For example *myPackage.myModule*, here *myPackage* is a directory and *myModule* is a module. *myModule* is in *myPackage* directory. In simple words, package is a directory in which we can find different modules. Example package structure.

#Another sub package

#Top – level package

# initialize the package. #Sub package

myPackage/

init .py subPackage/

init .py myModule1.py myModule2.py

....

secSubPackage/

init .py Mod1.py Mod2.py

When we import package python finds through directories on sys.path.

The init .py file is required to make python treat the directory as package, to prevent directory with same name.

## Importing from package

Now we have that question How to import or use these packages and it’s modules. So, for that we can use *import* keyword. If we want to import all modules of package, we can use all variable if it has been define. To define all variable just put the below code in init .py file.

all = [‘myModule1’,’myModule2’]

Meaning of above code is, myPackage/subPackage/ init .py file has that code, and we can write like this also if all variable is undefined.

*from myPackage.subPackage import \**

We can import individual package also,

import myPackage.subPackage.myModule1

An alternative way of importing module is:

from myPackage.subPackage import myModule1

Even we also have one more variation to import module function or variable also:

from myPackage.subPackage.myModule1 import myVariable

We can also give the reference to set the relative import to indicate current or parent package.

from . import myModule1 from .. import subModule1

from ..<subPackageName> import <moduleName>

# Tuples & Lists

Tuple is a collection of different object separated by commas, it is similar to a list in terms of indexing, but the basic difference between tuple and list is tuple is immutable, whereas list is mutable.

Creating Tuple:

>>> X=()

>>> print(X) ()

>>>

Here, X is an empty tuple.

>>> X=('a','b','c')

>>> print(X) ('a', 'b', 'c')

>>> Y='x','y','z'

>>> print(Y) ('x', 'y', 'z')

>>>

In the above code, two ways to define tuple. If there is only one element in tuple, you just need to put comma after element otherwise it will consider as a variable. For example,

>>> A='XYZ'

>>> print(A) XYZ

>>> A='XYZ',

>>> print(A) ('XYZ',)

>>>

### Concatenation of tuples:

We can concatenate two tuples also:

>>> print(X+Y)

('a', 'b', 'c', 'x', 'y', 'z')

>>>

### Nested tuples:

We can nest two tuples:

>>> tuple1=X,(1,2,3)

>>> print(tuple1)

(('a', 'b', 'c'), (1, 2, 3))

>>>

>>> tup1=([1,2,3],[4,5,6])

>>> tup1

([1, 2, 3], [4, 5, 6])

>>>

Tuples are immutable, but it can contain mutable objects.

#### Mutable and immutable objects meaning:

Mutable objects can be changed after created, where immutable objects cannot be changed after created.

>>> X[0]=123

Traceback (most recent call last):

File "<pyshell#15>", line 1, in <module> X[0]=123

TypeError: 'tuple' object does not support item assignment

>>>

Tuples are immutable so, we cannot assign value to it.

Find the length of tuples by using len().

>>> X=()

>>> len(X) 0

>>> X=(1,2,3)

>>> X (1, 2, 3)

>>> len(X) 3

>>>

For empty tuples, X length of tuple is 0.

Assign elements of tuples to variables by using:

>>> a,b,c=X

>>> a 1

>>> b 2

>>> c 3

>>>

Repetition of tuples i.e. if you want to store same values to tuple just multiply tuple by number,

>>> A=('apple','banana','mango')\*3 #it will repeat tuple three times.

>>> print(A)

('apple', 'banana', 'mango', 'apple', 'banana', 'mango', 'apple', 'banana', 'mango')

>>>

Slicing of tuples:

decrease by two elements (5, 3, 1)

>>>

#remove last element #reverse tuple

#reverse by two i.e. starts with last element and

#from 0th element to next three elements

>>> X[:3]

(1, 2, 3)

>>> X[:-1]

(1, 2, 3, 4)

>>> X[::-1]

(5, 4, 3, 2, 1)

>>> X[::-2]

('banana', 'mango', 'apple', 'banana', 'mango', 'apple', 'banana', 'mango')

>>> A[2:4] #from second element to 4th element ('mango', 'apple')

>>> X=(1,2,3,4,5) #define new tuple

#start from first element

>>> A[1:]

Deletion of tuples:

>>> del A

>>> A

Traceback (most recent call last):

File "<pyshell#27>", line 1, in <module> A

NameError: name 'A' is not defined

>>>

Here in the above example *del* is a keyword to delete object.

Lists:

As I mentioned in the above tuple section, both tuples and lists are same just mutability differs. i.e. List is mutable and tuple is immutable. List is another data structure like tuple. List is also a collection to store different elements.

Define lists:

>>> X=[1,2,3,4,5]

>>> X

[1, 2, 3, 4, 5]

>>>

List is mutable so we can assign element,

>>> X[0]=0

>>> X

[0, 2, 3, 4, 5]

>>>

Even we can append element to list at the end of list by using append(). Syntax: list.append(val)

>>> X.append(6)

>>> X

[0, 2, 3, 4, 5, 6]

>>>

Another method to append element into the list is extend, this method is used as append all the elements from the iterable.

Syntax: list.extend(iterable)

>>> X.extend([123,987])

>>> X

[0, 2, 3, 4, 5, 6, 123, 987]

>>>

Append and extend both methods are used to add elements at the end of list. So, to add element in a list at any index there is insert method in python.

Syntax: list.insert(index,element)

>>> X.insert(1,1)

>>> X

[0, 1, 2, 3, 4, 5, 6, 123, 987]

>>>

Even we can use + operator to add elements to the list. + operator is used to concatenate current list with new list.

>>> X=X+[12,32]

>>> X

[0, 1, 2, 3, 4, 5, 6, 123, 987, 12, 32]

>>>

To remove an element from the list whose value is *x*. If the element is not there in the list, it generates an error.

Syntax: list.remove(x)

>>> X=[1,2,3,4,5,6,7]

>>> X

[1, 2, 3, 4, 5, 6, 7]

>>> X.remove(5)

>>> X

[1, 2, 3, 4, 6, 7]

>>>

To remove an element from the list at the specified index. Pop() is used. Syntax: list.pop(index)

>>> X=[1,2,3,4,5,6,7,8,9]

>>> X

[1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> X.pop(5) 6

>>> X

[1, 2, 3, 4, 5, 7, 8, 9]

>>>

To remove all elements from list use clear(). Syntax: list.clear()

>>> X.clear()

>>> X []

>>>

To get the index of first occurred element from the list, index() is used. Syntax: list.index(x[,start[,end]])

Here start and end both are optional.

>>> X.index(5)

4

>>> X.index(5,5,10) 9

>>> X.index(5,7) 9

>>>

This method can cause ValueError if there is no such value available in the list.

To find the number of occurrences of an element from the list. Count() is used. Syntax: list.count(x)

>>> X.count(5) 2

>>>

To sort the items of the list. Sort() is used. Syntax: list.sort(key=none,reverse=false)

>>> A=['c','e','a','d','b','g','f']

>>> A

['c', 'e', 'a', 'd', 'b', 'g', 'f']

>>> A.sort()

>>> A

['a', 'b', 'c', 'd', 'e', 'f', 'g']

>>> A.sort(reverse=True)

>>> A

['g', 'f', 'e', 'd', 'c', 'b', 'a']

>>> A.sort(key=len,reverse=True)

>>> A

['g', 'f', 'e', 'd', 'c', 'b', 'a']

>>>

Here key is any callable function, and reverse is set to either true or false to sort the element either

in reverse order. These both arguments are optional.

To get the elements in reverse order, reverse() is used. Syntax: list.reverse()

>>> lst=[1,2,3,4,5]

>>> lst.reverse()

>>> lst

[5, 4, 3, 2, 1]

>>>

To get the copy of list, copy() is used. Syntax: list.copy()

>>> lst.copy()

[5, 4, 3, 2, 1]

>>>

**Dictionary:**

One more data type in the form of data structure we have is dictionary. It is one of the most powerful data structure in python. It works as hash map or in other language sometimes it refers as associative array. So, in dictionary we can define keys as well as values associated to that key. For example,

>>> stud={123:'abc',234:'pqr',345:'xyz'}

>>> stud

{345: 'xyz', 234: 'pqr', 123: 'abc'}

>>>

Here in the above example, 123, 234, 345 are keys. Keys can be any immutable types. Dictionary objects are indexed by keys. Generally, keys are made up with number or string type. Tuples are also immutable so to define a key for dictionary we can use tuples also based on one condition i.e. tuple should be made with number or string or tuple. If tuple contains any mutable object it cannot be a key for dictionary object. As lists are mutable we cannot use lists as key of dictionary object.

Dictionary is a set of keys : value pair. In which key should be unique. To define dictionary {} curly braces are used. Empty curly brace means empty dictionary. Otherwise, pairs of keys : values separated by comma, defines dictionary and give initial value to dictionary.

Generally, dictionary is used to sort values with associated keys. It is also possible to delete pair of key and value from dictionary by using *del*. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing *list(d.keys())* returns list of all key values. But if we want to fetch sorted keys use

*sorted(d.keys())*. To check whether the key is there in the dictionary use *in* keyword.

>>> stud={123:'abc',234:'pqr'}

>>> stud

{234: 'pqr', 123: 'abc'}

>>> stud[345]='xyz'

>>> stud

{345: 'xyz', 234: 'pqr', 123: 'abc'}

>>>stud[123] 'abc'

>>> del stud[234]

>>> stud

{345: 'xyz', 123: 'abc'}

>>> stud.keys() dict\_keys([345, 123])

>>> list(stud.keys()) [345, 123]

>>> sorted(stud.keys()) [123, 345]

>>> 345 in stud True

>>> 234 in stud False

>>> 123 not in stud False

>>>

The *dict()* constructor is used to create dictionary object.

>>> dict([('a','Apple'),('b','Banana'),('c','Chikoo')])

{'c': 'Chikoo', 'a': 'Apple', 'b': 'Banana'}

>>>

By using arbitrary keys we can create dictionary object through dict comprehension.

>>> sqr={x:x\*\*2 for x in (2,4,6)}

>>> sqr

{2: 4, 4: 16, 6: 36}

>>>

While working with dict() if keys are simple string, we can assign its’ value easily.

>>> dict(abc=1,qpr=2,xyz=3)

{'abc': 1, 'xyz': 3, 'qpr': 2}

When we use loop for dictionary object, we need to work with its’ keys and values. By using item() method we can get the keys and values.

>>> for k,v in stud.items(): print(k,v)

345 xyz

123 abc

>>>

## Function as object:

### Default Argument:

We can also set the default argument, in UDF. To work with default argument, we need to set

*“arg=value”*

>>> def add(a,b=2):

return a+b

>>> add(5) 7

>>> add(5,4) 9

>>>

### Variable Length Argument:

We have already discussed about function i.e. User Defined Function in previous topic, but python provides more functionality with UDF. So, lets’ take a look on different functionalities:

Variable Length Argument:

To define variable length in our argument, use asterisk (\*) sign with argument. For example, if we want to pass simple argument with variable length argument we need to write:

*def prnt(first,second,\*var):*

Here, in the above function definition *prnt* is the name of function, *first* and *second* both are simple argument and *\*var* is variable length argument.

>>> def Msg(\*a):

print(a)

>>> Msg("Hello"," World",", Hello"," Python!!!")

('Hello', ' World', ', Hello', ' Python!!!')

>>>

## Function as Object:

A function object's data is divided into two primary parts. The parts that would be the same for all functions created by the same function definition are stored in the function's *code object*, while the parts that can change even between functions created from the same function definition are stored in the function object.

The most interesting part of a function is probably its bytecode. This is the core data structure that says what to actually do to execute a function. It's stored as a bytestring in the function's code object, and you can examine it directly:

>>> Msg. code

<code object Msg at 0x03002930, file "<pyshell#15>", line 1>

>>> Msg. code .co\_code b't\x00\x00|\x00\x00\x83\x01\x00\x01d\x00\x00S'

>>>

It is not human readable.

## Strings

Like other languages, python also supports strings to perform different operations on string. there is a built – in class *str* in python to work with string.

To define a string we can use either single inverted commas or double inverted commas.

>>> str1="Hello World"

>>> str1 'Hello World'

>>> str2='Welcome to Python'

>>> str2

'Welcome to Python'

>>>

Python strings are immutable i.e. after creating string object will not change. So, we create a new string after operations. To work with characters of strings we can use [] (pair of square brackets) and index should be starts with zero like Java and C++. And + operator is used to concatenate two strings.

>>> str1[2] 'l'

>>> str2[0] 'W'

>>> str1+str2

'Hello WorldWelcome to Python'

>>> str1[:5]+str2[10:]

'Hello Python'

>>>

In Java when we use + operator to concatenate string with any numeric value, that numeric value converted into string, but it is not possible in python. In Python it will not automatically convert. To Convert numeric value to string value *str()* is used in python.

>>> a=5

>>> s1='Value of a:' + str(a)

>>> s1

'Value of a:5'

>>>

There are some characters through which if we start string it works differently with some characters with print(). For example, you can check the difference.

>>> s2=r'Hello\tWorld'

>>> s2

'Hello\\tWorld'

>>> s3='Hello\tWorld'

>>> s3

'Hello\tWorld'

>>> print(s2) Hello\tWorld

>>> print(s3) Hello World

>>>

Here in the above example, we use *r* character in s2 and it treats \t like all other characters. No special treatment for \ backslashes. If we use *u* before string, it allows us to write Unicode characters.

### Different String Methods

String.lower(), string.upper() – convert case of string.

>>> nm='Hello world'

>>> nm.upper() 'HELLO WORLD'

>>> nm.lower() 'hello world'

>>>

String.strip() – is used to remove white spaces from start and end.

>>> nm1=' Hello Python '

>>> len(nm1) 14

>>> len(nm1.strip()) 12

>>>

String.isalpha() – is used to check characters in a string is alphabetic or not.

>>> 'Hello'.isalpha() True

>>>

String.isdigit() - is used to check characters in a string is digit or not.

>>> '123'.isdigit() True

>>>

String.isspace() - is used to check characters in a string is space or not.

>>> ' 123'.isspace () False

>>> ' '.isspace() True

>>>

String.isalnum() – is used to check characters in a string is alphabet or numeric or not.

>>> '123abc'.isalnum() True

>>>

String.startswith(‘string’) – is used to check string starts with a character or string that we have passed in argument.

String.endswith(‘string’) – is used to check whether the string ends with the string that you pass in argument or not.

>>> 'hello'.startswith('h') True

>>> 'hello'.startswith('hel') True

>>> 'hello'.endswith('o') True

>>> 'hello'.endswith('llo') True

>>>

String.find(‘arg’) – is used to search string arg that you have passed in an argument with in string. in returns an index from where it begins. Returns -1 if not found.

String.replace(‘old’,’new’) – is used to replace old string and put new string in place of old one.

>>> 'hello world'.find('el') 1

>>> 'hello world'.replace('o','\*') 'hell\* w\*rld'

>>>

String.split(‘delimiter’) – is used to split the string in list of substring separated by delimiter.

>>> 'hello world hello python'.split(' ') ['hello', 'world', 'hello', 'python']

>>>

String.join(list) – is used to join the list of substring in a string.

>>> lst=['hello','welcome','to','python']

>>> ','.join(lst) 'hello,welcome,to,python'

>>>

String.capitalize() – is used to capitalize each word. i.e. first character capitalize and rest are lower case.

>>> 'hello world'.capitalize () 'Hello world'

>>>

String.casefold()- is used to casefolding to string. it generally works like lowercase.

>>> 'A'.casefold() 'a'

>>>

String.center(length[,fillchar]) – is used to return centered string as the length, you have passed. And if we pass fillchar, it fills the char.

>>> 'hello'.center(10) ' hello '

>>> 'hello world'.center(25,'\*') '\*\*\*\*\*\*\*hello world\*\*\*\*\*\*\*'

>>>

String.decode(encoding = 'UTF-8',errors = 'strict') - Decodes the string using the codec registered for encoding.

String.encode(encoding = 'UTF-8',errors = 'strict') - Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.

>>> s1.encode(encoding='UTF-8')

b'Hello World'

>>> s2=s1.encode(encoding='UTF-8')

>>> s2.decode('UTF-8') 'Hello World'

>>>

String.expandtabs(tabsize = 8) - Expands tabs in string to multiple spaces; defaults to 8 spaces per

tab if tabsize not provided. It returns copy of string with all tab characters replace with whitespace character.

World'

Hello

'

>>>

>>> str1='\tHello\tWorld'

>>> str1 '\tHello\tWorld'

>>> str1.expandtabs() ' Hello World'

>>> str1.expandtabs(tabsize=16)

String.index(str, start = 0, end = len(string)) - Same as find(), it searches from start to end. But raises

an exception if str not found.

>>> s1.index('Wor',0,len(s1)) 6

>>> s1.index('world',0,len(s1)) #world not found (case sensitive) Traceback (most recent call last):

File "<pyshell#17>", line 1, in <module> s1.index('world',0,len(s1))

ValueError: substring not found

String.swapcase() - Inverts case for all letters in string. It creates toggle effect on string.

>>> s1.swapcase() 'hELLO wORLD'

>>>

String.title() - Returns title cased string, that is, all words begin with uppercase and the rest are

lowercase.

>>> s1.title() 'Hello World'

>>>

String.zfill (width) - Returns original string left padded with zeros to a total of width characters;

intended for numbers, zfill() retains any sign given (less one zero).

>>> number='9999'

>>> number.zfill(7) '0009999'

>>>

### Escape sequence

Escape sequences are those character symbols which are used with backslashes. And it works with some specific meaning. Some symbols I give here in table below.

|  |  |
| --- | --- |
| **Escape Sequence Symbol** | **Meaning** |
| \a | Bell or alert |
| \b | Backspace |
| \cx or \C – X | Control – X |
| \e | Escape |
| \f | Formfeed |
| \M-\C-X | Meta Control X |
| \n | New line |
| \nnn | Octal notation, here n means the range 0, 7 |
| \r | Carriage return |
| \s | Space |
| \t | Tab |
| \v | Vertical tab |
| \X | X – character |
| \xnn | HexaDecimal notations. |

### String Format Specifiers

While working with formatted string, we need to show different values with string in different format. So, here we have some format specifiers. In other words, we can say string formatting operator. It works same as printf() in C programming language.

|  |  |
| --- | --- |
| **Characters** | **Meaning** |
| %c | Character |
| %s | String |
| %i & %d | Signed decimal integer |
| %u | Unsigned decimal integer |
| %o | Octal Number |
| %x | Hexadecimal Number (in Small Letters) |
| %X | Hexadecimal Number (in Capital Letters) |
| %e | Exponential notation(lowercase ‘e’) |
| %E | Exponential notation(uppercase ‘E’) |
| %f | Floating point number |
| %g | Shorter of %f & %e |
| %G | Shorter of %f & %E |

## Switch case:

Unlike other programming languages, python doesn’t have switch case statement. So, we can use dictionary mapping in replacement of switch case statement.

>>> def test\_switch(arg): switcher={

1: 'One',

2: 'Two',

3: 'Three',

}

return switcher.get(arg,'Nothing')

>>> test\_switch(1) 'One'

>>>

**References:**

Dive into python 3 <http://docs.python.org/>

<https://developers.google.com/edu/python/>

<https://www.geeksforgeeks.org/>

<https://www.tutorialspoint.com/python3/>

# Ch.2 Object Oriented Programming

As we have seen basic introduction of Python 3.X, now we will move to Object Oriented Programming using python.

This type of programming approach comes, to manage data, provide security, to manage with real world problems. The main aim behind object oriented programming is real world problems. From the concept of real world entities, this programming approach comes.

Python is a multi paradigm programming language, and it supports Object Oriented Programming also.

### What is Classes?

Classes are made up with data and functions known as class members. For example, class of fruits, class of animals, etc. Class is an Abstract Data Type it is used to create new object.

Python class is slight different than other programming languages. It has minimum of new syntaxes and semantic. It is a mixture of the class mechanisms found in C++ and Modula – 3. Python classes provide all the standard features of Object Oriented Programming.

### What is Objects?

Objects represent class. For example, Mango is an object of fruit class. Object represents real world entities that can represent a class. Object is a combination of data and functions those are member of class. So, we can say ***object = data + function***. Or even we can say different attribute an object has and it behaves differently. For example mango is an object of fruit class but it has different data (attributes) such as, colour, height, width, smell etc. And it has different function (behaviour) such as

, it is sweet and sour in taste.

***Creating class and object example,***

*>>> class Fruit: pass*

*>>> x=Fruit()*

*>>> y=Fruit()*

*>>> print(x)*

*< main .Fruit object at 0x025174F0>*

*>>> print(y)*

*< main .Fruit object at 0x02517450>*

*>>>*

*>>> x1=x*

*>>> print(x1)*

*< main .Fruit object at 0x025174F0>*

*>>> print(x==y) False*

*>>> print(x1==x) True*

*>>> print(type(x))*

*<class ' main .Fruit'>*

*>>>*

***#wap to demonstate class & object.***

class Fruit:

var = 'Mango'

def init (self,name): self.name=name

obj=Fruit('Banana') print('Fruit is:',obj.name)

print('Class Fruit is:',obj. class .var)

***O/P:***

>>>

Fruit is: Banana Class Fruit is: Mango

>>>

Here, var is class variable obj is object. And through obj we get name property of Fruit class. And

class uses to access class attributes. init is a special method in python it is known as constructor of class or initialization method of class. That python called each time when you create a new instance of your class. Self is used to create a reference of that particular class.

**Method**

Method is used to define any action that is performed by objects. When it is needed to call method by an object, it will perform its’ task. As we learned class is made up with different attributes and functions, methods are those functions to perform the tasks and work on different attributes.

Example,

*obj=Fruit('Banana') obj.show()*

*print('Class Fruit is:',obj. class .var)*

*#Method show*

*#wap to demonstate method. class Fruit:*

*var = 'Mango'*

*def init (self,name): self.name=name*

*def show(self):*

*print('Fruit is:',self.name)*

In the above example we made a simple show method. The proper way to define a method in Python OOP:

* Method is there inside the class. Here, we define a method show inside a class Fruit,
* If method is inside a class we can use it’s attributes easily, it binds class attributes and perform different OOP tasks. Such as, abstraction, encapsulation, inheritance etc.
* The first parameter should be the reference of the class instance, i.e. self
* We can make any type of method i.e. with parameters, without parameters (other than self), with return value, without return value.
* To call the methods of class we need an object.

This is special method in Python whereas; in other programming languages such as, Java and C++ we call this type of method as constructor to build an object while creating it.

**init method**

In python, it is automatically called after object has been successfully created, no need to call it like other method, again like other programming languages like C++ or Java init method is either parameterized or without parameterized (other than, self parameter)

There is no constructor in python, to build an object we need to crate init method first for our class.

Till now we have discussed about class, objects and methods this three basic things are there in any object oriented programming languages but python is very vast and popular in the matter of OO programming. Let’s explore it more,

### Static methods

This type of method we can define to call without reference of class. i.e. either by class name or object. For that we need to use *@staticmethod* decorator (annotation) before we define method such as, in example below,

#wap to demonstrate static method class StaticMethod:

cnt=0

def init (self): type(self).cnt+=1

@staticmethod def disp():

return StaticMethod.cnt

print(StaticMethod.disp()) X=StaticMethod() print(X.disp()) print(StaticMethod.disp())

Here, in the above example we define disp() method as static method by using @staticmethod decorator (annotation) and we can call this method by using class name or even we can call it through instance of that class. Even we can check we cannot use ***self*** neither as argument nor in method’s statement in static method.

### Class methods

In python we also have one more decorator (annotation) to work with class method. It works like static method but it is slightly different than static method. As we have seen different types of methods in python, let’s differentiate three different types of methods.

|  |  |  |
| --- | --- | --- |
| Instance Method | Class Method | Static Method |
| It is simple methods that can works as member of class. To  call this method we need an object or instance. | It is also a class member but it cannot be called through instance of a class. | It is a member of class but like class method object or instance of a class not needed. |
| It works with ***self*** parameter. To call this type of methods we need object. | There is no ***self*** parameter, it works with ***cls*** parameter to point to the class not an object. It also calls like static method,  through the name of class. | There is no ***self*** no ***cls*** type of parameter. Because to call static method we need either name of class or instance, it has  the reference of class. |
| It can modify state of object and class. | It can’t modify the state of  object. But it can modify state of class by using cls. | It can’t modify object or class  state. Because it has no self no cls type of argument. |
| No flag or decorator is used. | @classmethod decorator is  used. | @staticmethod decorator is  used. |

Example of class method, static method and instance method.

#wap to demonstrate class instance and static method. class Methods:

def instance(self): print("Instance Method")

@classmethod

def classMethod(cls): print("Class Method")

@staticmethod

def staticMethod(): print("Static Method")

X=Methods() X.instance() Methods.classMethod() Methods.staticMethod()

O/P

>>>

Instance Method Class Method Static Method

>>>

So, here we can see the difference between these methods. To call instance method we have to use object. And to call class method and static method we have to use class name. even we can use name of object also. but we can see here for class method we use @classmethod decorator and for static method we use @staticmethod decorator.

**Access Specifiers (Modifiers):** We can define different class member and give them access specifiers also, these access specifiers are, Public, Private and Protected.

While working with Object – Oriented Programming we need to think in terms data security. For data security, we need access specifiers. Which data needs more security and which data needs least security. On the basis of data and it’s security we need to assign access specifiers or modifiers to the class members.

* **Private :** these types of attributes are used inside the class only in which they are defined.
* **Protected :** these types of attributes are used in class, but it works more than private and less than public access. It is used under certain condition only.
* **Public :** these are used anywhere freely.

|  |  |  |
| --- | --- | --- |
| **Naming** | **Type** | **Meaning** |
| Var | Public | We can use it anywhere inside or outside class. |
| \_var | Protected | We cannot use it outside class, even in subclass also. |
| var | Private | These types of attributes are invisible and inaccessible. |

In python there are no keywords to define variables or attribute as public, private or protected. Let’s take an example of different access specifiers,

>>> class Access:

def init (self):

self.pub='Public' self.\_pro='Protected' self. pri='Private'

>>> obj=Access()

>>> obj.pub 'Public'

>>> obj.\_pro 'Protected'

>>> obj. pri

Traceback (most recent call last):

File "<pyshell#9>", line 1, in <module> obj. pri

AttributeError: 'Access' object has no attribute ' pri'

>>>

Here we get an error while asking for the private attribute, because of the information hiding, private data is invisible and we ask to show, it gives an error there is no attribute pri. And even we cannot fetch private data outside class.

### Destructor

While working with Object Oriented Programming, we have constructor and when we think in terms of constructor, at that time we also think in terms of destructor. But there is no constructor or destructor in python. We have different methods to construct object we use init and to destruct object or instance del method is used. In other words we can say init method is used as constructor and del method is used as destructor.

#wap to demonstrate destructor

class DemoDest:

def init (self,name): print(name+" object created")

def del (self): print("Object Destroy")

print("X is creating") x=DemoDest("A") print("Y is creating") y=DemoDest("B")

del x

print("X is deleting") del y

print("Y is deleting")

O/P

>>>

X is creating

1. object created Y is creating
2. object created Object Destroy X is deleting Object Destroy Y is deleting

>>>

### Attributes

Attributes are known as properties of an object as we have seen that object has different attributes and it behave differently. Such as check in code snippets,

>>> class Cars: pass

>>> x=Cars()

>>> x.name='Tesla'

>>> x.familyCar=True

>>> x.sportsCar=False

>>> print(x.name) Tesla

>>> print(x.familyCar) True

>>>

Here, name, familyCar and sportsCar are three different attributes in class Cars. Even we can define attribute by using the name of class. Such as in code snippets,

>>> Cars.mileage='79km'

>>> print(x.mileage) 79km

>>>

If want to check internal work of object, i.e. how object works? , to assign and work with different attribute object creates dictionary by using dict , you can check by using below code snippet,

>>> x. dict

{'name': 'Tesla', 'familyCar': True, 'sportsCar': False}

It is not the proper way and it is not possible to define attribute by using object. Attributes are generally defined in class. If there is an undefined attribute, it will generate an attribute error. To avoid that error we can use **getattr()** it will define and give default value also by using this function.

>>> getattr(func,'x') 5

>>>

Here, in the getattr() we can assign third variable to give default value to the attribute such as,

>>> getattr(func,'x',9)

We can define an attribute through function also, define function and then by that function we can define an attribute.

>>> def func():

return 5+2

>>> func.x=5

>>> print(func.x) 5

>>> print(func()) 7

>>>

But the proper way to create and define the attribute is inside the class, we need to define different attribute same as we define in the example of *Fruit class var=’Mango’* is attribute of class.

### Abstraction and Encapsulation

**Data Abstraction** means show essential things those are necessary, but it is not possible for everything i.e. some data we cannot display in front of everyone is known as Data Encapsulation. **Data Encapsulation** works on data hiding and encapsulation bundles hidden data in different functions and provides security.

**Information Hiding** is the key thing to modularity. In which data attributes of a class invisible to clients of the class. And data accessed only through the object’s methods. In Java, C++ like languages provides enforcing information hiding. But python does not provide enforcing information hiding.

There is no way for implementing of a class to restrict access to the attributes of class instances. Example,

In general we make getter, setter function to provide encapsulation and abstraction. Let’s see in the example

print(lang.getName())

lang1=Languages("Python") #with parameter init call print(lang1.getName())

**O/P**

>>>

None Python

>>>

#without parameter init call

lang=Languages()

#wap to demonstrate abstraction and encapsulation class Languages:

def init (self,name=None): #name is defined with default value self.name=name

def setName(self,name): self.name=name

def getName(self): return self.name

Here, in the above example we define a class Languages and make a constructor i.e. our special method init with name parameter, we define this parameter with a default value ***None***. And we define two methods to get the name and to set the name. Now if we set the value for lang object by using setName() method,

lang.setName("Java") print(lang.getName())

#without parameter init call

lang=Languages()

Then output will be,

>>>

Java Python

>>>

So, here we can see the getName() and setName() methods encapsulate the data.

### Properties

Properties – to set a value or to retrieve value of any object, we need to work with properties. For example, if we are working with GUI and we take a label to display data, we need to set height and

width of that label, here height and width are properties of label. Getter and Setter are used in many Object Oriented Programming Languages, to make work easy and encapsulate data. They are also known as mutator methods. Getter and setters are nothing but a type of method, it works as properties. To get value using properties @property decorator (annotation) is used. To set value @<getter\_function>.setter decorator is used. Here getter is the name of function that you defined for setter.

Example of property,

#wap to demonstrate properties class Props:

def init (self,name): self.name=name

@property

def getProp(self): return self.name

@getProp.setter

def setProp(self,name): self.name=name

obj=Props("abc") obj.setProp="HELLO" print(obj.getProp)

O/P

>>> HELLO

>>>

This method is also works fine and even in newer version of python we can use obj.setProp like

attribute as well as like functions obj.setProp() both are working.

And even we have new another way to work with property. We can define functions like other programs, and just apply it in function properties.

#wap to demonstrate properties(2 way)

class Props:

def init (self,val=0): self.val=val

def getVal(self): return self.\_val

def setVal(self,val): self.\_val=val

val=property(getVal,setVal)

obj=Props() obj.val=456 print(obj.val)

O/P

>>> 456

>>>

This method also works fine, but the proper way to use and work with properties in pythonic way we need to use @property decorator (annotation).

### Inheritance

While working with OOP, as we have already known about OO programming comes from real world concepts and things, **Inheritance** is one of them. Like other Object Oriented Programming languages, Python also supports inheritance.

### What is Inheritance?

Inheritance means one class can inherit from another class. A class can inherit attributes and behaviour and methods from one class to another class. There is a relationship between these classes like parent and child relationship. A class that inherits attribute from another class is known as super or parent or base class. And to which class it inherits is known as sub class or child class or derived class.

Syntax:

class DerivedClassName(BaseClassName): pass

Example,

#wap to demonstrate inheritance class Parent:

def init (self): print("Parent Class")

class Child(Parent): def init (self):

Parent. init (self) print("Child Class")

objChild=Child() O/P

>>>

Parent Class Child Class

>>>

### Overloading and Overriding

We have two different concepts overloading and overriding. Overloading and overriding both are the concept of polymorphism. Method overloading doesn’t support directly by python.

For example,

#wap to demonstrate overloading def add(a,b):

return a+b

def add(a,b,c): return a+b+c

#first form of add method add(5,2)

#second form of add method add(5,2,2)

O/P

>>>

Traceback (most recent call last):

File "L:/Python34/overload\_ride.py", line 9, in <module> add(5,2)

TypeError: add() missing 1 required positional argument: 'c'

>>>

It shows error for two arg. Add method because python doesn’t work with method overloading. It overwrites the method add with last added method. To avoid this error we can work differently in one method only there is no different form of methods.

Example,

#wap to demonstrate overloading def add(\*arg):

if len(arg)==1: return arg

else:

return arg[0]+arg[1] print(add(5))

print(add(5,2))

O/P

>>> (5,)

7

>>>

To achieve Method Overriding, we need to work with inheritance it’s meaning is when we have parent class method and want to redefine it in child class, we need to work with overriding. In

method overriding we need to define a method in parent class and need to change it’s definition in child class as per the need.

Example,

#wap to demonstrate method overriding. class Employee:

def init (self,eid,enm): self.eid=eid self.enm=enm

def str (self):

return str(self.eid)+"<===>"+self.enm

class Salary(Employee):

def init (self,eid,enm,sal): super(). init (eid,enm) self.sal=sal

def str (self):

return super(). str ()+"<===>"+str(self.sal)

obj=Salary(123,"abc",50000) print(obj)

O/P

>>> 123<===>abc<===>50000

>>>

We can check in the above example of method overriding, str method it is same as toString() of java, it returns any object values or attribute in string. We override both str () and init () methods from parent class Employee to Salary. We just redefine it and we also use super() to call

init () or str () from super class, and execute super class statements’ otherwise it will not return super class value in subclass.

### Multi level Inheritance

As we all know about, different types of inheritance. Now we are going to work with multi level inheritance. In this approach, we have more than two levels and more than one level of parent class, such as,



Grand Parent Class

Parent Class

Child Class

Example,

#wap to demonstrate multi level inheritance class GrandParent:

def init (self): print("Grad Parent Class")

class Parent(GrandParent): def init (self):

super(). init () print("Parent Class")

class Child(Parent): def init (self):

super(). init () print("Child Class")

obj=Child() O/P

>>>

Grad Parent Class Parent Class Child Class

>>>

Here we can see the different levels of class and inheritance. GrandParent class is a base or super class of Parent class, Parent class is a base of Child class. So, Child class has properties of both Parent class and GrandParent class.

### Hierarchical Inheritance

In this type of inheritance we have one parent class and more than one child class, is known as hierarchical inheritance such as,

Parent

Class

Child

Class1

Child

Class2

Here, we can see we have parent class and it has two child class i.e., parent class shares it’s properties between child class1 and child class2.

Example,

#wap to demonstrate hierarchical inheritance. class Parent:

def init (self): print("Parent Class")

class Child1(Parent): def init (self):

super(). init () print("Child 1 Class")

class Child2(Parent): def init (self):

super(). init () print("Child 2 Class")

ch1=Child1() ch2=Child2()

O/P

>>>

Parent Class Child 1 Class Parent Class Child 2 Class

>>>

Here, we initialize Child1 class object ch1 and Child2 class object ch2.

### Multiple Inheritance

Python supports Multiple inheritance like C++. In multiple inheritance there are two parent of a child class. Such as,

Parent

Class1

Parent

Class2

Child

Class

But it is not possible in real world as per some programming languages such as, Java and C#. So, Java and C# do not support multiple inheritance. But it works through interface to achieve multiple inheritance.

Example,

#wap to demonstrate multiple inheritance. class Parent1:

def init (self): print("Parent 1 Class")

class Parent2:

def init (self): print("Parent 2 Class")

class Child(Parent1,Parent2): def init (self):

Parent1. init (self) Parent2. init (self) print("Child Class")

ch=Child() O/P

>>>

Parent 1 Class

Parent 2 Class Child Class

>>>

### Hybrid Inheritance

When we use two different types of inheritance in our application we can say it is hybrid inheritance. For example, if we work with multi level inheritance with hierarchical inheritance, we can say it is Hybrid Inheritance.

Example,

#wap to demonstrate hybrid inheritance. class Parent:

pass

class Child(Parent): pass

class Child1(Parent): pass

class Child2(Child,Parent): pass

print("Child is subclass of Parent:",issubclass(Child,Parent)) print("Child1 is subclass of Parent:",issubclass(Child1,Parent)) print("Child2 is subclass of Parent:",issubclass(Child2,Parent)) print("Child2 is subclass of Child2:",issubclass(Child2,Child))

O/P

>>>

Child is subclass of Parent: True Child1 is subclass of Parent: True Child2 is subclass of Parent: True Child2 is subclass of Child2: True

>>>

Even we can work with Multipath inheritance while working with python, is also known as Diamond Problem. Now, how multipath inheritance works? Or what is diamond problem?

A

B

C

D

Multiple inheritance example,

#wap to deonstrate diamond problem class A:

def disp(self): print("A class")

class B(A):

def disp(self): print("B class")

class C(A):

def disp(self): print("C class")

class D(B,C): pass

obj=D() obj.disp()

O/P

>>>

B class

>>>

### Exception Handling

Exception handling is an error handling technique. An exception is an error, will generate while executing program. Sometimes it depends on user’s behaviour, to manage that error exception handling is used. Exception doesn’t occur frequently. Like other programming languages such as, C++, Java, PHP, C# etc. Python also supports exception handling to deal with different run – time error or problem occur while execution of program.

While running our program, we face different problems such as, file find or open error at that moment exception occurs and that interrupt the flow of program. To manage the flow of program we need to work with exception handling by putting piece of code, known as exception handler.

In python, exception handling is much similar to Java. Like Java programming python also supports

***try*** block, but in case of catch python has ***except*** keyword.

Example,

#wap to demonstrate exception handling by using try x=int(input("Enter first no:"))

y=int(input("Enter second no:")) try:

z=x/y print(z)

except ZeroDivisionError: print("Can't divide by zero")

O/P

**On Success**

>>>

Enter first no:5 Enter second no:2 2.5

>>>

**On Fail**

>>>

Enter first no:5 Enter second no:0 Can't divide by zero

>>>

### Multiple except block

Sometimes, multiple type of error may occur in our program execution. So, we need to manage all different errors for that we can use multiple except in our program.

Example,

#wap to demonstrate exception handling by using multiple except block try:

x=int(input("Enter first no:")) y=int(input("Enter second no:")) z=x/y

print(z) except ValueError:

print("Please, Enter valid interger") except ZeroDivisionError:

print("Can't divide by zero")

O/P

>>>

Enter first no:abc

Please, Enter valid interger

>>>

Even we can use multiple except like below example,

#wap to demonstrate exception handling by using multiple except block try:

x=int(input("Enter first no:")) y=int(input("Enter second no:")) z=x/y

print(z)

except (ValueError,ZeroDivisionError):

print("Plz. Check, there is invalid interger or divide by zero")

Or we can create an object of that exception as example below.

#wap to demonstrate exception handling by using multiple except block try:

x=int(input("Enter first no:")) y=int(input("Enter second no:")) z=x/y

print(z)

except ValueError as e: print("Error:",e)

except ZeroDivisionError as e1: print("Exception:",e1)

Or sometimes we don’t know type of exception in advance, at that time we need to use except keyword like below example,

#wap to demonstrate exception handling by using multiple except block try:

x=int(input("Enter first no:")) y=int(input("Enter second no:")) z=x/y

print(z)

except ZeroDivisionError as e1: print("Exception:",e1)

except:

print("Exception occur")

O/P

>>>

Enter first no:abc Exception occur

>>>

It is also possible to create custom – exception or User Defined Exception in python. Like example below,

#wap to demonstrate custom exception class MyException(Exception):

pass

try:

x=int(input("Enter number:")) if x==0:

raise MyException else:

print(x) except MyException:

print("Plz. Enter valid number")

O/P

>>>

Enter number:0

Plz. Enter valid number

>>>

Here in the above example, we can see the use of ***raise*** statement. ***raise*** statement is used to indicate the exception is to be raised.

### Else clause

In python one more facility we have i.e. else clause with try – except statement. We can use else clause after all except blocks. else clause will be executed if there is no exception throws by the try statement. For example,

#wap to demonstrate exception handling by using else clause try:

x=int(input("Enter first no:")) y=int(input("Enter second no:")) z=x/y

print(z)

except ZeroDivisionError as e1: print("Exception:",e1)

except:

print("Exception occur") else:

print("No Exception!!!")

O/P

>>>

Enter first no:5 Enter second no:2 2.5

No Exception!!!

>>>

### Clean – up actions

So far we handle exceptions in different ways, but there are some circumstances in which we need to work in any cases whether our work is to be done nicely, or an exception occurs. Such as, while handling file operations, we open a file and perform operations but if we find error at that time exception occurs but we need to close file in all the circumstances. So we have clean – up actions in python, with finally clause we can clean up our exception.

Example,

#wap to demonstrate exception handling by using finally clause try:

x=int(input("Enter first no:")) y=int(input("Enter second no:")) z=x/y

print(z) finally:

print("Bye..Bye...")

In above example, we use try and finally only without except keyword. We can use try – except and finally also.

#wap to demonstrate exception handling by using finally clause try:

x=int(input("Enter first no:")) y=int(input("Enter second no:")) z=x/y

print(z) except:

print("Exception occur") finally:

print("Bye..Bye...")

O/P

>>>

Enter first no:5 Enter second no:0 Exception occur Bye..Bye...

>>>

>>>

Enter first no:5 Enter second no:2 2.5

Bye..Bye...

>>>

### Some built – in exceptions

|  |  |  |
| --- | --- | --- |
| BaseException | | This class is Base class of all Exception class. |
| Exception | | This is base class for all built – in non – system – exiting, all user  defined function should be derived from this class. |
| ArithmeticError | | Three types of ArithmeticError occur   * OverflowError * ZeroDivisionError * FloatingPointError |
| BufferError | | Raised when buffer related operations cannot be performed. |
| LookupError | | Raised when a key or index used on mapping or sequence is invalid or not found.   * KeyError * IndexError |
| **Concrete Exception**  Those types of exceptions usually raised. | | |
| AssertionError | When assert statement failed. | |
| AttributeError | When attribute reference not find or assignment fails. | |
| EOFError | Raised when I/O functions are at End Of File while reading data. Or  some functions return empty or end of file directly. | |
| FloatingPointError | Raised when floating point operation fails. | |
| ImportError | Import statement is unable to load. | |
| ModuleNotFoundError | Raised when module not found when try to import. | |
| IndexError | When specified index is out of range. | |
| KeyError | When mapping key is not found. | |
| KeyboardInterrupt | When user press interruption key such as delete or ctrl + c | |
| MemoryError | When operation is out of memory. | |
| NameError | When local or global name is not found, variable name is not  declared. | |
| NotImplementedError | When derived classes override methods. | |
| OSError([arg]) | When system function returns system related value. | |
| OverflowError | When arithmetic operation is out of range. | |
| RecursionError | When interpreter detects that the maximum recursion depth is  exceed. | |
| ReferenceError | When a weak reference proxy is used to access an attribute of the  referent after garbage collection. | |
| RuntimeError | When no other exception applies. | |

|  |  |
| --- | --- |
| SyntaxError | When parser encounters syntax error. Or when read standard  input. |
| SystemError | When the interpreter finds an internal error. |
| SystemExit | When sys.exit() function is called. |
| TypeError | When inappropriate type of object is used. |
| ValueError | When built – in operation receives an argument that has right type  but has invalid value. |
| ZeroDivisionError | When second argument is zero of division or modulo operation. |

There are lots of other built in exceptions in python.

### Assert statement

Python assert statement is used to test condition while debugging. If condition is true, it does nothing and program works nicely. But if condition is false, it generates ***AssertionError***.

Syntax,

assert\_statement : : = “assert” expression1 [“,” expression2]

Here, expression1 is test the condition and the optional expression2 is an error message that displayed if assertion is failed.

The above syntax is equivalent to

if debug :

if not expression1: raise AssertionError(expression2)

Example,

#wap to demonstrate assert statement def max(a,b):

assert(a>b),"B should not be greater than A" if(a>b):

return "A is Max"

print(max(5,2))

print(max(2,5))

O/P

>>>

A is Max

Traceback (most recent call last):

File "L:/Python34/assert\_state.py", line 8, in <module> print(max(2,5))

File "L:/Python34/assert\_state.py", line 3, in max assert(a>b),"B should not be greater than A"

AssertionError: B should not be greater than A

>>>

### Searching Algorithms

There are lots of different algorithms, at present we are working with Searching Algorithms such as,

* Linear Search
* Binary Search

### Linear Search

Let’s start with linear search; linear search is also known as sequential search. When data items are stored in a collection such as lists, they are stored in linear or sequential fashion. Each data items

stored in a position related to others. In Python lists have relative positions are index values for each data items. Index values are ordered, so we can visit each data item in sequence. And in linear search, start searching with first item and move to next item and same procedure works until end of list once we find the key value, it stops searching and if we can’t find key value, it will send message “No Key value found”.

Linear search is used to find element in an unsorted list. In this searching technique, search requires n comparisons in worst case so, time complexity for ***worst case*** for linear search is ***O(n)***.

For best case we can say that we find element in first comparison, so ***best case time complexity is O(1)***. Time complexity for ***average case is O(n/2)*** if item is present in list. But if item is not present in the list, time complexity for ***best case***, for ***worst case*** and for ***average case*** is ***O(n)***.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Value | 54 | 26 | 93 | 55 | 58 | 10 | 15 | 68 | 93 | 28 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Start

Like above figure, linear search works in sequence.

**Algorithm of Linear Search**

Linear\_search(list,n,element)

Where, list – represents list of elements.

n – represents size of list (optional argument) element – represents value to be searched in the list.

Step – 1: [initialize]

K=1

Flag = 1

Step – 2: repeat through step – 3 for k=1,2,3, ..., n Step – 3: if list[k]==element

Flag=0

Output “Successful element found at location k”

Step – 4: if flag

Output “Can’t Search” Exit

Step – 5: End

Example,

#wap to demonstate linear search def linear(lst,element):

flag=False

for i in range(0,len(lst)): if lst[i]==element:

flag=True

print("%d Element Found at %d location"%(lst[i],i)) break

if not flag:

print("Element not Found")

lst=[1,2,3,5,12,455,34,56,68]

element=12 linear(lst,element)

Example2,

#wap to demonstate linear search def linear(lst,element):

flag=False

for i in range(0,len(lst)): if lst[i]==element:

flag=True

print("%d Element Found at %d location"%(lst[i],i)) break

if not flag:

print("Element not Found")

lst=[]

n=int(input("Enter Size of list")) for i in range(0,n):

data=int(input("Enter data into the list:")) lst.append(data)

element=int(input("Enter element that you want to find from list:")) linear(lst,element)

### Binary Search

When we have sorted list, and we want to search an element at that time we can use binary search. In linear searching technique, searching is performed in sequential manner. So, we need to start from the first element and compare n-1 elements. In binary search algorithm, searching will starts from the middle element in an ordered list. If the element that we find is matched with middle element, it completes our search. If it is not a middle element, check whether the element that we want to find is smaller than middle element, check only left lower half portion. It emits entire right portion i.e. upper half portion. And if the element that we find into the list is greater than the middle element, check right upper half portion.

See the process how it works? We want to find 34 from the below list. Let’s start with middle element. i.e. 48 in below example. Now check 34 is smaller than 48 so use lower half.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9 | 12 | 22 | 34 | 46 | 48 | 55 | 56 | 67 | 70 | 81 |

**Algorithm**

Binary\_search(element,n,list) Where,

Element – which we want to find from the list N – size of list [optional argument]

List – list of elements Step – 1: [initialize]

Low=0 [first index of list] High= n [last index of list] Flag=1

Step – 2: repeat through step - 4 while(low<=high) Step – 3: mid =(low+high)/2

Step – 4: if(element<list[mid])Then

High=mid-1

Else if(element >=list[mid])Then Low=mid+1

Else if(element==list[mid])Then

Output “Found element at location” mid

Flag=0

Return

Step – 5: if(flag)Then

Output “Can’t find element”

Return

Example of binary search,

#wap to demonstrate binary search in iterative way def binary\_search(lst,element):

flag=False low=0 high=len(lst)-1

while(low<=high and not flag): mid=(low+high)//2

if lst[mid]==element: flag=True

elif lst[mid]>element: high=mid-1

elif lst[mid]<element: low=mid+1

return flag

lst=[1,3,8,12,15,23,36,48,52]

element=3 flag=binary\_search(lst,element) if flag:

print("Element Found") else:

print("Not Found")

Example of binary search in recursive way,

#wap to demonstrate binary search in recursive way

def binary\_search(lst,element): if len(lst)==0:

return False else:

mid=len(lst)//2

if lst[mid]==element: return True

else:

if lst[mid]>element:

return binary\_search(lst[:mid],element) elif lst[mid]<element:

return binary\_search(lst[mid+1:],element)

lst=[1,3,8,12,15,23,36,48,52]

element=36 print(binary\_search(lst,element))

Binary search algorithm takes exactly **O(log n)**, as it eliminates half portion of list. Thus running time of a binary search is proportional to **log n**.

### Sorting Techniques

Now we move to sorting techniques, we have lots of different sorting techniques. i.e. arrange list of items in an order either in ascending or in descending order. So, let’s start with different sorting techniques. In this book, we will learn,

* Bubble sort
* Insertion sort
* Shell sort
* Selection sort
* Quick sort
* Merge sort
* Heap Sort

Let’s start with,

### Bubble sort

It is a very simple and easy technique. However, it is not efficient for large list compare to other sorting methods. But it works fine with smaller list. It compares adjacent elements and swap elements those are out of order.

#### Algorithm,

Bubble\_sort(list,n)

Where, n – size of list [optional argument] List – lust of elements

Step – 1: [initialize]

i=1

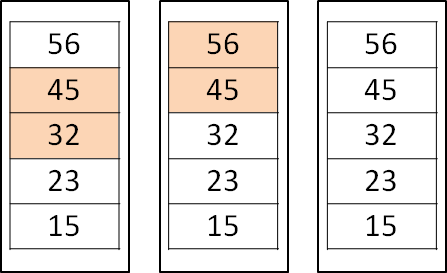
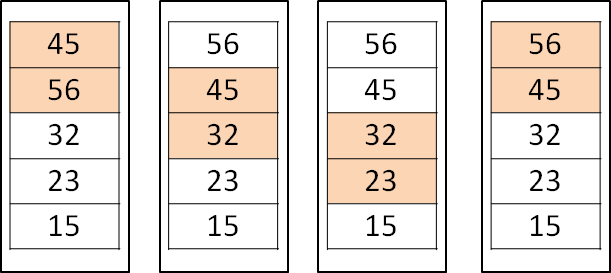
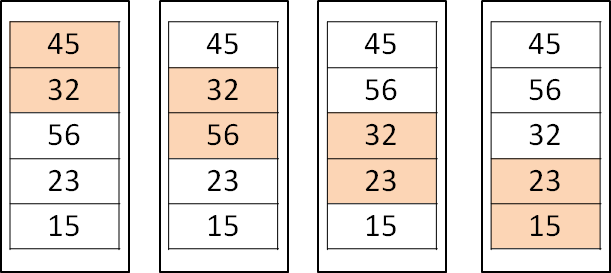
Step – 2: repeat through step – 5 while(i<n) Step – 3: j=1

Step – 4: repeat through step – 5 while(j<n-i) Step – 5: if list[j]<list[j+1]

Temp=list[j+1] List[j+1]=list[j] List[j]=temp

Step – 6: Exit

Operations of Bubble Sort



This much comparison takes place while working with bubble sort. So, it takes too much of time for execution. So, that we cannot use bubble sort for huge list. As we have seen in the algorithm outer loop passes the control to inner loop and it performs n-i comparison and outer loop executes n times. So, the total number of comparison becomes n2 times so the time complexity for bubble sort is O(n2).

Example,

#wap to demonstrate bubble sort. def bubble\_sort(lst):

for i in range(0,len(lst)-1):

for j in range(0,len(lst)-1-i): if lst[j]<lst[j+1]:

temp=lst[j+1] lst[j+1]=lst[j] lst[j]=temp

return lst

lst=[45,56,21,32,11,47]

print(bubble\_sort(lst))

### Insertion Sort

One more sorting technique we are going to learn is insertion sort. It is also a simple method. Insertion sort is also works fine with small number of elements. The example of insertion sort is sort a hand of playing card. To sort playing card, first thing is to find the correct position of a card, we remove a card at a time and compare with each card already in hand, from right to left.

#### Algorithm,

Insertion\_sort(list,n)

Where list – list of elements

N – size of list [optional argument] Step – 1: [initialize]

List[0]=0

Step – 2: repeat through Step – 3 to 5 for i=1, 2, 3, 4, n

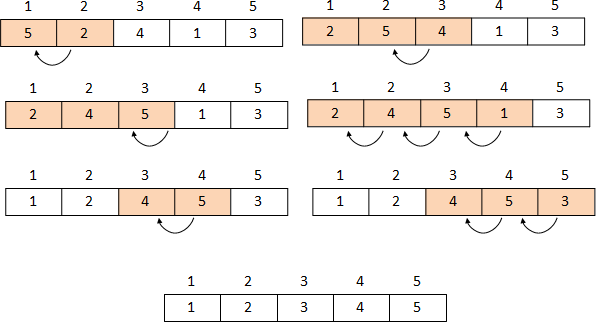
Step – 3: temp=list[i]

Pointer=i-1

Step – 4: while(temp < list[pointer])

List[pointer+1]=list[pointer] Pointer=pointer-1

Step – 5: list[pointer]=temp Step – 6: exit



**Operation of Insertion Sort**

Example,

#wap to demonstrate insertion sort. def insertion\_sort(lst):

for i in range(1,len(lst)): cur\_val=lst[i] pointer=i

while pointer>0 and lst[pointer-1]>cur\_val: lst[pointer]=lst[pointer-1] pointer=pointer-1

lst[pointer]=cur\_val return lst

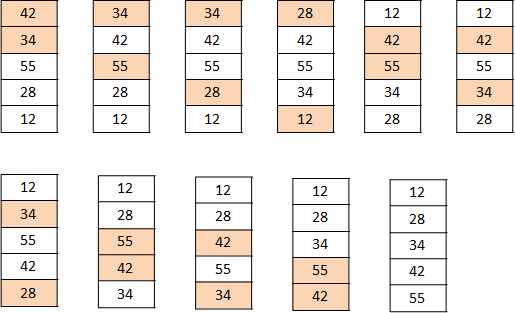
lst=[12,5,35,27,1,8]

print(insertion\_sort(lst))

The maximum number of comparison is n-1. So, time complexity of insertion sort is O(n2). In best case, we need only one comparison and it is only possible with already sorted list.

### Selection Sort

Selection sort is an extended version of bubble sort. Selection sort starts from first element and searches the entire list. Select smallest or largest value as per the order and place it on first place, then check for the second place and so on. The process continues until the complete list is sorted.



#### Operation Process of Selection Sort

Algorithm,

Selection\_sort(list,n)

Where, list – list of elements

N – size of list [optional argument] Step – 1: current = 0 [initialize]

Step – 2: repeat through step – 7 while(current < size) Step – 3: j = current+1

Step – 4: repeat through step – 6 while(j<size) Step – 5: if(list[current]>list[j])

Temp = list[current] List[current]=list[j] List[j]=temp

Step – 6: j=j+1

Step – 7: current = current+1 Step – 8: exit

Example,

#wap to demonstrate selection sort. def selection\_sort(lst):

for current in range(0,len(lst)-1): j=current+1

while j<len(lst):

if lst[current]>lst[j]: temp=lst[current] lst[current]=lst[j] lst[j]=temp

j=j+1 current=current+1

return lst

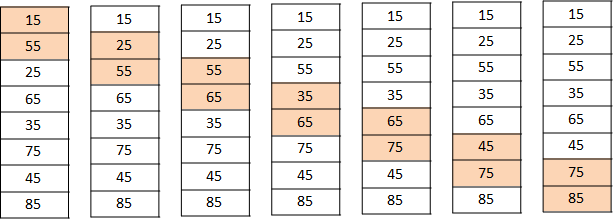
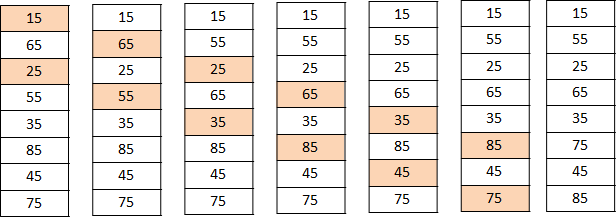
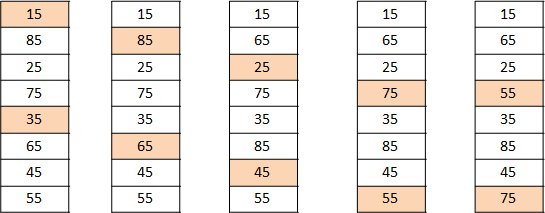
lst=[45,32,56,23,78,12]

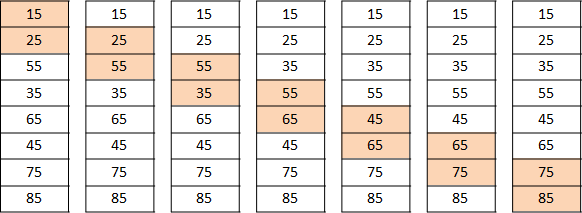
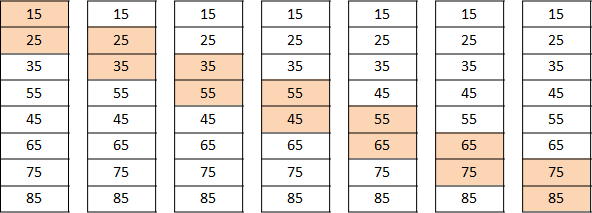
print(selection\_sort(lst))

While analyzing this sorting technique again it executes loop n-1 times. So, it takes O(n2) time to sort list.

### Shell Sort

Shell sort, this method is developed by Donald Shell. So, the name is Shell sort. It compares element by a specific distance. i.e. a gap. Process continues until the elements compared with gap and all elements are arranged in order.



**Operation process for Shell Sort**

Algorithm,

Shell\_sort(list,n)

Where, list – list of elements

N – size of list [optional argument] Step – 1: [initialize]

Gap=size/2

Step – 2: repeat through step – 6 while(gap=gap/2) Step – 3: swap = false

Step – 4: repeat through step – 6 while(swap)

Step – 5: repeat through step – 6 for i=0,1,2,3,4,. i<(size-gap)

Step – 6: if(list[i]>list[i+gap])

Temp=list[i] List[i]=list[i+gap] List[i+gap]=temp Swap=true

[end of for loop] [end of while loop]

[end of outer while loop] Step – 7: output

Step – 8: exit

Example,

#wap to demonstrate shell sort. def shellSort(lst):

gap=len(lst)//2 while(gap>0):

swap=False while(swap!=True):

for i in range(0,len(lst)-gap): if lst[i]>lst[i+gap]:

temp=lst[i] lst[i]=lst[i+gap] lst[i+gap]=temp swap=True

gap=gap//2 return lst

lst=[45,23,34,12,67,25,78]

print(shellSort(lst))

While analyzing Shell sort it takes time to execute n times and gap is reduces by half in every iteration so, time complexity of Shell Sort is O(n2).

Before we start quick sort or merge sort, we need to clear divide and conquer technique.

### Divide – And – Conquer

There is lots of different algorithmic paradigm. Such as,

* Greedy programming,
* Dynamic programming,
* Divide and conquer,
* Brute – force programming,
* Back tracking,
* Branch and bound etc.

At present we are going to learn Divide and conquer technique.

Divide and conquer is a technique in algorithms. It divides problem into smaller sub problems and independently solve each of the sub problems. And then combine the sub problems solutions so again we can get the proper solutions of the main original problem. With this technique we conquer the problem i.e. after dividing the main problem into sub problems, conquering means recursively solve the sub problems. And then we combine it. So it reduces the complexity by factor. It is very useful and generally used programming technique. We use this technique in different problems such as, binary search, quick sort, merge sort etc.

### Quick Sort

Quick Sort is developed by British computer scientist Tony Hoare in 1959 and published in 1961. The quick sort uses divide and conquer technique to sort the entire list. In this sorting method first of all we need to select pivot value. There are lots of different ways to select pivot. Based on pivot value we can further divide our list into two different sub lists. Left sub list and right sub list, left sub list contains elements that are smaller than pivot and right sub list contains elements that are greater than the list. It works recursively. This process continues until list is sorted.

Algorithm,

Quick\_sort(list,first,last) Where list – list of elements

First – represents the position of the first element in the list.

It’s value will be changed during execution

Last – represents the position of the last element in the list. It’s

value will be changed during execution Step – 1: [initialize]

Low=first High = last

Pivot=list[(low+high)/2] [we consider middle value as

pivot]

Step – 2: repeat through step – 7 while(low<=high) Step – 3: repeat while(list[low]<pivot)

Low=low+1

Step – 4: repeat while(list[high]>pivot)

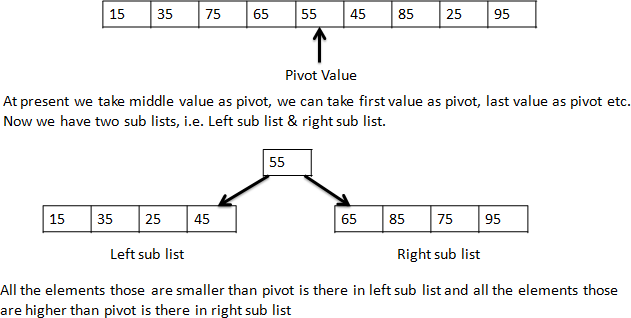
High=high-1 Step – 5: if(low<=high)

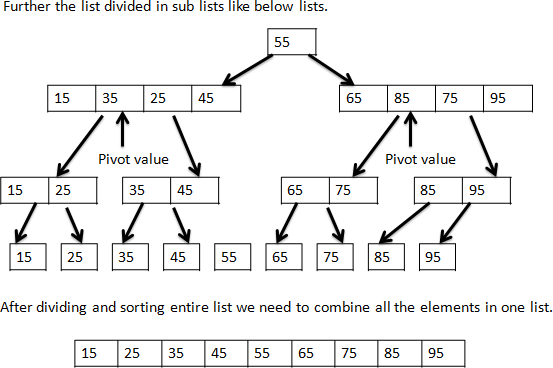
Temp=list[low] List[low]=list[high] List[high]=temp Low=low+1

High=high-1 Step – 6: if(first<high)

Quick\_sort(list,first,high) Step – 7: if(low<last)

Quick\_sort(list,low,last)





#### Operations for quick sort

Quick Sort uses divide and conquer technique and it works recursively so it reduces the time complexity. Time complexity of Quick Sort is O(n log n) time in Average case, but in worst case time complexity of Quick Sort is O(n2).

Example,

#wap to demonstrate quick sort. def quick\_sort(lst,first,last):

low=first high=last

pivot=lst[(low+high)//2] while(low<=high):

while(lst[low]<pivot): low=low+1

while(lst[high]>pivot): high-=1

if(low<=high): temp=lst[low] lst[low]=lst[high] lst[high]=temp low+=1

high-=1 if(first<high):

quick\_sort(lst,first,high) if(low<last):

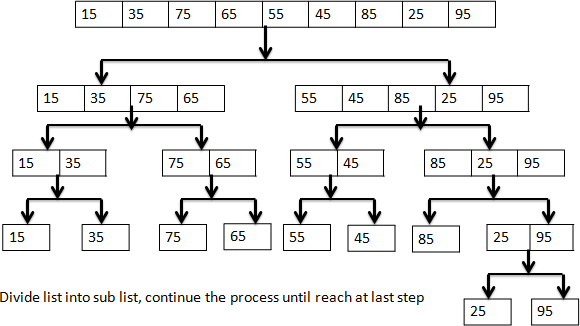
quick\_sort(lst,low,last) return lst

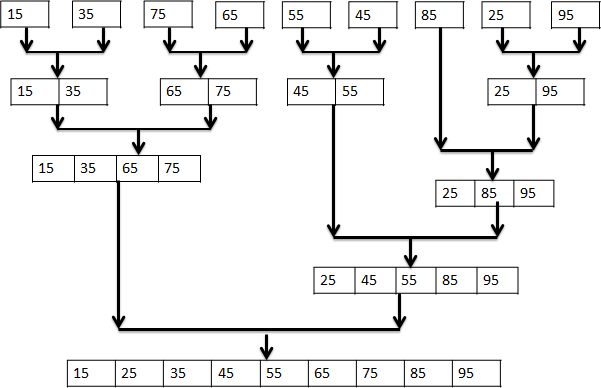
lst=[65,25,15,75,35,55,85,45,95]

print(quick\_sort(lst,0,len(lst)-1))

### Merge Sort

Merge sort is also works on divide – and – conquer technique. Merge sort is a recursive algorithm it splits list in half. If list is empty or has one item, it is already sorted. To split our list we need at least two elements in the list. And it recursively invoke a merge sort on both sub lists. Then we need to merge all those sub lists and combine them together into a single list.





#### Operation of Merge Sort

Then we need to merge all the elements into a single sorted list, as we can see in the above figure. Algorithm,

Merge\_sort(list,size)

Where list – list of elements

Size – size of list [optional argument] Step – 1: if(size>1)

[get middle index value and divide list in two sub lists.] Mid=size/2

Left=list[0,mid] Right=list[mid+1,size]

Step – 2: [call recursively]

Merge\_sort(left,mid) Merge\_sort(right,size)

Step – 3: [initialize]

i=j=k=0

Step – 4:[start merging from here] repeat through step – 3 while(i< size of left and j<size of right)

if(left[i]<right[j]) list[k]=left[i] i=i+1

else

list[k]=right[j] j=j+1

k=k+1

Step – 5: repeat through while(i<size of left)

list[k]=left[i] K=k+1

I=i+1

Step – 6: repeat through while(j<size of right)

list[k]=right[j] k=k+1

j=j+1

Step – 7: return

In the above algorithm we define two different functions; one is for sorting and another for merging. There are different ways to work with merge sort. We are working here as per the operation that mentioned earlier.

While analyzing the merge sort it works on any number of elements with any type of processors or system, time complexity of merge sort is O(n log n) in worst case, best case and in average case also.

But if it ask for extra memory it takes O(n) time. So, we can say merge sort is more powerful than other sorting techniques.

Example,

#wap to demonstrate merge sort. def merge\_sort(lst):

if len(lst)>1:

mid = len(lst)//2 left = lst[:mid] right = lst[mid:]

merge\_sort(left) merge\_sort(right)

i=0 j=0 k=0

while i < len(left) and j < len(right): if left[i] < right[j]:

lst[k]=left[i] i=i+1

else:

lst[k]=right[j] j=j+1

k=k+1

while i < len(left): lst[k]=left[i] i=i+1

k=k+1

while j < len(right): lst[k]=right[j] j=j+1

k=k+1 print("Merging ",lst)

lst=[65,25,15,75,35,55,85,45,95]

merge\_sort(lst) print(lst)

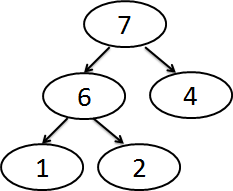
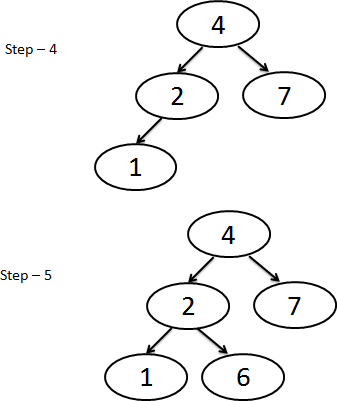
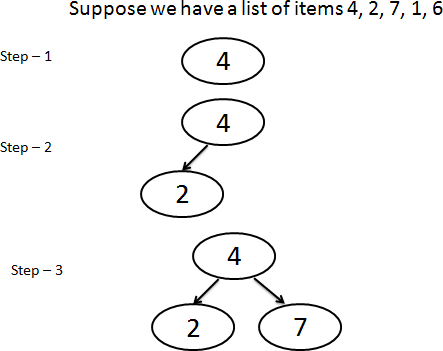
### Heap Sort

Heap sort is based on complete binary tree. Complete binary tree means a binary tree in which every level except last level completely filled and all parent nodes have exactly two children and it has height *h* and it has 2h+1-1 nodes. In complete binary tree every parent node has a value less than or equals to any of its children value.

While analyzing heap sort it takes O(n log2 n) for n values.

Now to heapify we swap 7 and 4 and 2 and 6

First step is to build heap

There are 2 steps for Heap Sort.

* First step is to build heap
* Second step is to sorting.

Algorithm to heapify,

Heapify(list,n,i)

Where, list – list of elements N – size of heap

I – index to heapify sub tree rooted on index i.

Step – 1: [initialize]

Largest = i [initialize largest as root] Left = 2\*i + 1 [initialize left node] Right = 2\*i + 2 [initialize right node]

Step – 2: [check for the left child of root exists or not and is greater than the root]

If(left<n and list[i]<list[left]) Largest = left

Step – 3: [check for the right child of root exists or not and is greater than the root]

If(right<n and list[largest]<list[right]) Largest = right

Step – 4: [change the root if needed]

If(largest != i)

List[i]=list[largest] List[largest]=list[i] Heapify(list,n,largest) [heapify the root]

Heap\_sort(list,size)

Where, list – list of elements

Size – size of list [optional argument] Step – 1: [build maximum heap]

Repeat for i=size,size-1, size-2,. ,-1

Heapify(list,n,i) Step – 2: [extract one by one element]

Repeat for i=size-1, size-2, size-3,..., 0 List[i]=list[0]

List[0]=list[i] Heapify(list,i,0)

Example,

#wap to demonstrate heap sort. def heapify(lst,n,i):

large=i left=2\*i+1 right=2\*i+2

if(left<n and lst[i]<lst[left]): large=left

if(right<n and lst[large]<lst[right]): large=right

if large!=i:

lst[i],lst[large]=lst[large],lst[i] heapify(lst,n,large)

def heap\_sort(lst): n=len(lst)

for i in range(n,-1,-1): heapify(lst,n,i)

for i in range(n-1,0,-1): lst[i],lst[0]=lst[0],lst[i] heapify(lst,i,0)

return lst

lst=[4,2,7,1,6]

print(heap\_sort(lst))

### Hash table

Let us start with hashing; in data structure we have seen lists to store collections of items. Now we will be going to learn a data structure that we can search in O(1) time and this concept is knowing as hashing.

A hash table uses hashing and it is a collection of items through which we can search it easily. Each item store on a particular slot (position on which item is stored) which has integral name starting with 0 index. For example, we have a hash table with 10 items, so we have slot 0, slot 1, slot 2, ,

slot 9. Initially hash table is empty so python has special value for empty slot is *None*.



The mapping between item and slot where that item belongs in the hash table is called the hash function. The hash function will take any item from hash table and return an integer in the range of slot names. Even we can calculate the hash value by using the below formula.

h(item) = item/table size

So, for above example we have 10 values in table so table size is 10 and we store different values in this table like,

|  |  |
| --- | --- |
| **Item** | **Hash value** |
| 12 | 12%10 = 2 |
| 54 | 54%10 = 4 |
| 26 | 26%10 = 6 |

Like that, but hash value should be in the range of table size. The searching operation is O(1), as constant time is required for searching.

Example,

#wap to demonstratet hashtable. hash\_table=[None]\*10 print(hash\_table)

def hash(key):

return key%len(hash\_table)

def insert(hash\_table,key,val): hash\_key=hash(key) hash\_table[hash\_key]=val

insert(hash\_table,10,'ABC') print(hash\_table) insert(hash\_table,25,'PQR') print(hash\_table)

References:

Introduction to Algorithms, Second Edition by Thomas H. Cormen Expert Data Structures with C by R.B. Patel <http://docs.python.org/>

<https://www.geeksforgeeks.org/>

<https://www.python-course.eu/object_oriented_programming.php>

<https://realpython.com/python3-object-oriented-programming/>

<https://www.programiz.com/python-programming/>

<http://interactivepython.org/courselib/static/pythonds/>

**Ch.3 Plotting using PyLab and More Algorithms**

In this chapter we are going to explore plotting and algorithms and data structure more. As we have seen in previous chapter we have different types of techniques to solve problems, such as divide and conquer, dynamic programming, brute force, branch and bound etc. And we have seen divide and conquer technique also.

Now in this chapter we are going to learn about dynamic programming.

### Dynamic Programming

Dynamic programming was developed by Richard Bellman in 1950s. Dynamic programming typically applies to **optimization problem**. This technique is used for solving any problem efficiently by overlapping the problem. Dynamic programming is something like divide and conquers technique, in dynamic programming optimal solution can be found by combining optimal solutions to local subproblems. Divide – and – conquer algorithms divides the problem in independent subproblems and solve sub problems recursively, and then combine their solutions to solve the subproblems.

Where dynamic programming is used when subproblems are not independent. In dynamic programming algorithm, all the subproblems solves every subproblem just once and then solution of subproblems saves in an answer table. In optimization problem we have lots of different solutions and each solution has an answer with optimal value i.e. either maximum or minimum, we can call such solution as **optimal solution**.

### Fibonacci revisited

First of all, we look at Fibonacci solution by recursive way.

#wap to demonstrate Fibonacci by using recursive call def Fibonacci(n):

if n==0:

return 0 if n==1 or n==2:

return 1

else:

return Fibonacci(n-1)+Fibonacci(n-2)

print(Fibonacci(9),end=' ')

In the above solution we use recursive way, to solve Fibonacci. In the above code we call the function in function itself. And it executes again and again for same values. Here for this problem we can use dynamic programming and in above example we return value each time so we can save the value and then look it up rather than calculate it each time by using dynamic programming. It is known as **momeization**(to save values in answer table). And is the key thing in dynamic programming.

Example of revised Fibonacci series,

#wap to demonstrate fibonacci revised version using dynamic programming def Fibonacci\_revise(n):

ans=[0]\*1000 if n==0:

return 0 elif n==1 or n==2:

return 1

else:

res=Fibonacci\_revise(n-1)+Fibonacci\_revise(n-2) ans[n]=res

return res

print(Fibonacci\_revise(6),end=' ')

In the above code we are working with list, after getting answer value we save that value to the list and we use it for the next iteration we can say this process is memorization process.

### 0/1 Knapsack problem

Before we start with the 0/1 Knapsack problem let’s understand what the problem is actually?

It is also known as rucksack problem, in which we have set of items each with weight and values. Number of items are there in a collection so total weight is less than or equals to a given limit and total value is as large as possible. Problem arises when we have fixed – size knapsack and must be fill with valuable items.

There are two different versions of Knapsack problem,

**0/1 Knapsack problem:** items are indivisible; you either take an item or not. Some special instances can be solved with dynamic programming.

**Fractional Knapsack problem:** items are divisible; you can take any fraction of an item. So, here we are going to solve 0/1 Knapsack problem.

We want the best optimal solution of the knapsack problem.

So, we can follow some steps to solve the 0/1 knapsack problem.

1. Each item represented by *value* and *weight* attributes.
2. The knapsack can accommodate items with total weight i.e. capacity of knapsack not more than capacity(total weight *w*).
3. A vector *item*, represents collection of items. The size of vector *item* is up to the length of *n*.
4. A vector V, of length of n, represents whether item is taken or not. Such as, if *V[i]=1* than item *item[i]* is taken, if *V[i]=0* than item is not taken.
5. Find maximum *V*.

But there is a condition,

Dynamic programming, provides a practical solution for 0/1 Knapsack problem with reasonable amount of time.

Example,

#wap to demonstrate 0/1 Knapsack problem. def knapsack(value, weight, capacity):

n = len(value) - 1

m = [[-1]\*(capacity + 1) for \_ in range(n + 1)] return knapsack\_helper(value, weight, m, n, capacity)

def knapsack\_helper(value, weight, m, i, w): if m[i][w] >= 0:

return m[i][w]

if i == 0:

q = 0

elif weight[i] <= w:

q = max(knapsack\_helper(value, weight,m, i - 1 , w - weight[i])+ value[i],knapsack\_helper(value, weight,m, i - 1 , w))

else:

q = knapsack\_helper(value, weight,m, i - 1 , w) m[i][w] = q

return q

val=[60,100,120] wt=[10,20,30]

W=50

print(knapsack(val,wt,W))

As, we have discussed in the above steps one can test the above code with 3 different items with different values and weight. And we will find the optimal solution of the 0/1 knapsack problem.

### Dynamic Programming and Divide – and – conquer programming

Generally, in both methods we can solve problems by dividing the problem into the independent sub problems. But there are some differences.

As we all know about divide and conquer technique used independent sub problems and get solution of them and then combine it, whereas in dynamic programming we use memoization technique in which we save the result of sub problem.

There are some more differences in divide and conquer technique we divide the original problem in to sub problem so sub problem is smaller than original one. For example, in merge sort we divide the list of elements is smaller than the original list. Where, in dynamic programming sub problem is not much smaller than original one it is slightly smaller than the original problem. Such as in Fibonacci we execute any step suppose 5th step is not smaller than 6th step.

In the matter of efficiency, in divide and conquer algorithms do not depend on the structuring the algorithm so the same problems are solved repeatedly. Dynamic programming is efficient only when the number of sub problems is significantly smaller than total number of problems.

### PyLab

Generally, we all communicate by different languages but when we are communicating with a person who is not aware with our language, at that time we communicate by sign or images. But python provides this facility to work with visual data. It is simple and easy to manage visual data in python.

At present we are starting with PyLab, PyLab is a python standard module, to provide many facilities of MATLAB. MATLAB is a high level technical language to provide interactive IDE to develop

algorithms, data visualization, data analysis, numeric calculations etc. At present we are going to learn some facilities for plotting data.

PyLab is embedded with Matplotlib module of python. Through PyLab we can import some portion of Matplotlib and NumPy.

Now lets’ start with PyLab. First of all install PyLab, but we can’t directly install PyLab directly as we know that PyLab is embedded with Matplotlib we need to install Matplotlib.

To install Matplotlib, first of all we need to install NumPy and SciPy. If you are working with Windows just download *.whl* files of NumPy, SciPy and Matplotlib. And after downloading these *.whl* files run the below command.

pip install numpy-<version>.whl pip install scipy-<version>.whl

pip install matplotlib-<version>.whl

After successful installation of all above module, we can work with PyLab. So, let’s start ploting.

import pylab as plt

run the above command to import the pylab library or

import matplotlib.pylab as plt

to work with pylab, run the below code.

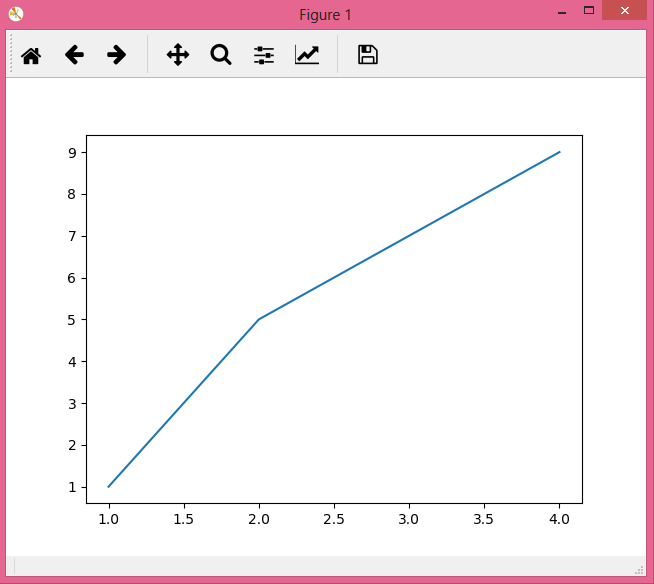
>>> import pylab as plt

>>> plt.plot([1,2,3,4],[1,5,7,9])

[<matplotlib.lines.Line2D object at 0x01A2D990>]

>>> plt.show()

This will show you the graph,



Here, we can see the graph it creates line 2D graph for our data i.e.[1,2,3,4],[1,5,7,9]. By using plot() we create a plot on which we can put our graph as per the data and by using show() we can show the graph. For plotting we need to pass x-coordinate and y-coordinate and both are of same length. Here, in the above example [1,2,3,4] is x-coordinate and [1,5,7,9] is y-coordinate.

We can also create multiple figures at the same time and also save as a *.png* file.

>>> import pylab

>>> pylab.figure (1)

<matplotlib.figure.Figure object at 0x02460930>

>>> pylab.plot([1,2,3,4],[1,5,3,7])

[<matplotlib.lines.Line2D object at 0x07FC2CB0>]

>>> pylab.figure(2)

<matplotlib.figure.Figure object at 0x07FC2C50>

>>> pylab.plot([1,3,5,7],[2,4,6,8])

[<matplotlib.lines.Line2D object at 0x07BA3530>]

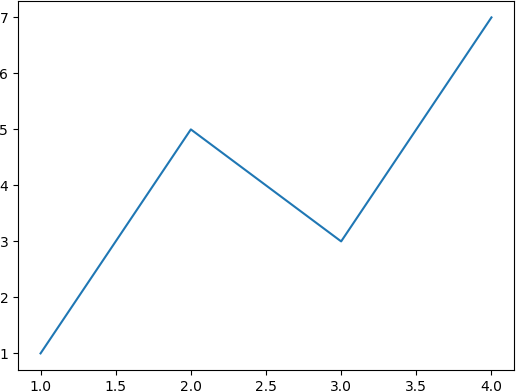
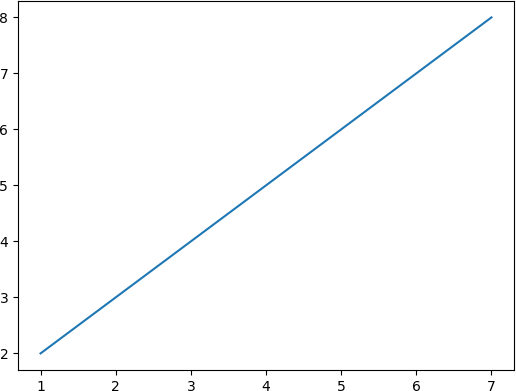
>>> pylab.savefig('abc')

>>> pylab.figure (1)

<matplotlib.figure.Figure object at 0x02460930>

>>> pylab.savefig('fig\_1')

>>> pylab.show()



**Figure abc.png Figure fig\_1.png**

In the above example, we use pylab.figure(n) here, n is a number. When we execute this method is sets the current figure to the figure number n that we set in the figure().

And savefig() is used to save the graph. We need to set the name of file. So, in the above example we set pylab.savefig(‘abc’) it will create and save abc.png file.

Now we will take another example,

>>> import pylab as plt

>>> plt.plot([1,2,3,4],[2,5,3,6],color='red',linewidth=5) [<matplotlib.lines.Line2D object at 0x07039FB0>]

>>> plt.show()

Here, in the above example we use color and linewidth attributes to set graph color and it is line graph on the plot.

>>> import pylab as py

>>> py.title('Population Growth') Text(0.5,1,'Population Growth')

>>> py.xlabel('Population in Crores') Text(0.5,0,'Population in Crores')

>>> py.ylabel('Years') Text(0,0.5,'Years')

>>> py.plot([100,105,110,115,120],[2001,2005,2010,2015,2020],'ro')

[<matplotlib.lines.Line2D object at 0x06AA9CF0>]

>>> py.show()

In the above example, we set the title of the graph, and axes labels. And by using

>>> py.plot([100,105,110,115,120],[2001,2005,2010,2015,2020],'ro')

We indicate the color and linestyle of the graph by default color of the graph is blue and linestyle is –

* 1. solid line but when we write ‘ro’ it means red color and o means solid dots. Even we can use such as
     + g^ : means green color triangle shape marks
     + bs : means blue color square shape marks
     + r- : means red color (single dash) line graph
     + r-- : means red color(double dash) break line etc.

The above example, we can code like this also

#wap to demonstrate pylab import pylab as plt yrs=2000

pop=100 x=[]

y=[]

for i in range(1,20,+2): y.append(yrs) x.append(pop) pop+=5

yrs+=5 plt.title('Population Growth')

plt.xlabel('Population in Crores') plt.ylabel('Years')

plt.plot(x,y) plt.show()

Here, in the above example we create two lists x and y and plotting on it.

In pylab there are lots of different methods for graph and different types of graph also available in matplotlib. You can get the tutorials of matplotlib official site. <https://matplotlib.org/users/tutorials.html>

There are different parameters with different functions to set fontsize, linewidth etc. See the below example.

>>> import pylab as plt

>>> plt.title('Test Data',fontsize='xx-large') Text(0.5,1,'Test Data')

>>> plt.plot([1,5,10,15],[2,4,6,8],linewidth=20) [<matplotlib.lines.Line2D object at 0x07C48CF0>]

>>> plt.xlabel('X - Axis',fontsize='medium') Text(0.5,0,'X - Axis')

>>> plt.show()

And even we can customize plot with lots of different settings and “rc” also known as rc settings. Some rc settings we can see here,

#to set line width

#to set font size for title

#to set size of numbers on x-axis #to set marker size

pylab.rcParams[‘lines.linewidth’]=10 pylab.rcParams[‘axes.titlesize’]=18 pylab.rcParams['xtick.labelsize'] = 12 pylab.rcParams[‘lines.markersize’]=15

<https://matplotlib.org/users/index.html>you can get more tutorials from this link of official site. Even you can find matplotlib from sourceforge.net also. from below link, you can get the customizing related tutorial. <http://matplotlib.sourceforge.net/users/customizing.html>

### Plotting Mortgages, an Extended Example

At the time of economical collapse, sometimes people have to put some of their properties on mortgage. Such as, land mortgage. So, we are going to discuss about how to plot this mortgages? Let’s start to explore mortgage. First of all we need to create class for mortgage and different functions to perform some task. Such as, payment, loan months, rate growth etc.

#wap to demonstrate plotting mortgage. import pylab

def findPayment(loan, r, m):

"""Assumes: loan and r are floats, m an int Returns the monthly payment for a mortgage of size loan at a monthly rate of r for m months""" return loan\*((r\*(1+r)\*\*m)/((1+r)\*\*m - 1))

This method is used to find the payment of monthly mortgage depends on the loan amount, rate and month.

class Mortgage(object):

"""Abstract class for building different kinds of mortgages""" def init (self, loan, annRate, months):

"""Create a new mortgage""" self.loan = loan

self.rate = annRate/12.0 self.months = months self.paid = [0.0] self.owed = [loan]

self.payment = findPayment(loan, self.rate, months) self.legend = None #description of mortgage

def makePayment(self): """Make a payment"""

self.paid.append(self.payment)

reduction = self.payment - self.owed[-1]\*self.rate self.owed.append(self.owed[-1] - reduction)

def getTotalPaid(self):

"""Return the total amount paid so far""" return sum(self.paid)

def str (self): return self.legend

def plotPayments(self, style):

pylab.plot(self.paid[1:], style, label = self.legend) def plotBalance(self, style):

pylab.plot(self.owed, style, label = self.legend) def plotTotPd(self, style):

"""Plot the cumulative total of the payments made"""

totPd = [self.paid[0]]

for i in range(1, len(self.paid)): totPd.append(totPd[-1] + self.paid[i])

pylab.plot(totPd, style, label = self.legend) def plotNet(self, style):

"""Plot an approximation to the total cost of the mortgage over time by plotting the cash expended minus the equity acquired by paying off part of the loan"""

totPd = [self.paid[0]]

for i in range(1, len(self.paid)): totPd.append(totPd[-1] + self.paid[i])

#Equity acquired through payments is amount of original loan

# paid to date, which is amount of loan minus what is still owed equityAcquired = pylab.array([self.loan]\*len(self.owed)) equityAcquired = equityAcquired - pylab.array(self.owed)

net = pylab.array(totPd) - equityAcquired pylab.plot(net, style, label = self.legend)

class Fixed(Mortgage):

def init (self, loan, r, months): Mortgage. init (self, loan, r, months)

self.legend = 'Fixed, ' + str(r\*100) + '%'

class FixedWithPts(Mortgage):

def init (self, loan, r, months, pts): Mortgage. init (self, loan, r, months) self.pts = pts

self.paid = [loan\*(pts/100.0)]

self.legend = 'Fixed, ' + str(r\*100) + '%, ' + str(pts) + ' Points'

class TwoRate(Mortgage):

def init (self, loan, r, months, teaserRate, teaserMonths): Mortgage. init (self, loan, teaserRate, months) self.teaserMonths = teaserMonths

self.teaserRate = teaserRate self.nextRate = r/12.0

self.legend = str(teaserRate\*100)+ '% for ' + str(self.teaserMonths)+ ' months, then ' + str(r\*100) + '%'

def makePayment(self):

if len(self.paid) == self.teaserMonths + 1: self.rate = self.nextRate

self.payment = findPayment(self.owed[-1], self.rate,self.months

- self.teaserMonths)

Mortgage.makePayment(self)

In the Mortgage class, we define methods plotPayments and plotBalance are simple and one

– liners, and in which we use pylab.plot. when a graph has multiple plots, it is useful to produce a key to identify what each plot represents.

And even we use label keyword as an argument to associate a string with the plot. A key pylab.legend represents legend key for graph.

plotTotPd method represents total of payment that has made.

plotNet method represents approx total cost of the mortgage.

def plotMortgages(morts, amt): styles = ['b-', 'b-.', 'b:'] #Give names to figure numbers payments = 0

cost = 1

balance = 2

netCost = 3 pylab.figure(payments)

pylab.title('Monthly Payments of Different $' + str(amt)+ ' Mortgages') pylab.xlabel('Months')

pylab.ylabel('Monthly Payments') pylab.figure(cost)

pylab.title('Cash Outlay of Different $' + str(amt) + ' Mortgages') pylab.xlabel('Months')

pylab.ylabel('Total Payments') pylab.figure(balance)

pylab.title('Balance Remaining of $' + str(amt) + ' Mortgages') pylab.xlabel('Months')

pylab.ylabel('Remaining Loan Balance of $') pylab.figure(netCost)

pylab.title('Net Cost of $' + str(amt) + ' Mortgages') pylab.xlabel('Months')

pylab.ylabel('Payments - Equity $') for i in range(len(morts)):

pylab.figure(payments) morts[i].plotPayments(styles[i]) pylab.figure(cost) morts[i].plotTotPd(styles[i]) pylab.figure(balance) morts[i].plotBalance(styles[i]) pylab.figure(netCost) morts[i].plotNet(styles[i])

pylab.figure(payments) pylab.legend(loc = 'upper center') pylab.figure(cost) pylab.legend(loc = 'best') pylab.figure(balance) pylab.legend(loc = 'best')

def compareMortgages(amt, years, fixedRate, pts, ptsRate,varRate1, varRate2, varMonths):

totMonths = years\*12

fixed1 = Fixed(amt, fixedRate, totMonths)

fixed2 = FixedWithPts(amt, ptsRate, totMonths, pts)

twoRate = TwoRate(amt, varRate2, totMonths, varRate1, varMonths) morts = [fixed1, fixed2, twoRate]

for m in range(totMonths): for mort in morts:

mort.makePayment() plotMortgages(morts, amt)

compareMortgages(amt=200000, years=30, fixedRate=0.07,pts = 3.25, ptsRate=0.05,varRate1=0.045, varRate2=0.095, varMonths=48) pylab.show()