History and Some Good Words

Python is a powerful programming language ideal for scripting and rapid application development. It is used in web development (like: Django and Bottle), scientific and mathematical computing (Orange, SymPy, NumPy) to desktop graphical user Interfaces (Pygame, Panda3D).

"Python's killer features – simple syntax that makes its code easy to learn and share, and its huge array of third-party packages – make it a good general purpose language. Its versatility is shown by its users and uses. The Central Intelligence Agency has employed it for hacking, Pixar for producing films, Google for crawling web pages and Spotify for recommending songs."

Out of top 50 programming language Python stands with C, C++ and Java. These four tops the list.

History of Python

Python is a fairly old language created by Guido Van Rossum. The design began in the late 1980s and was first released in February 1991.

In late 1980s, Guido Van Rossum was working on the Amoeba distributed operating system group. He wanted to use an interpreted language like ABC (ABC has simple easy-to-understand syntax) that could access the Amoeba system calls. So, he decided to create a language that was extensible. This led to design of a new language which was later named Python.

Why the name Python?

No. It wasn't named after a dangerous snake. Rossum was fan of a comedy series from late seventies. The name "Python" was adopted from the same series "Monty Python's Flying Circus".

Release Dates of Popular Versions

Version	Release Date
Python 1.0 (first standard release)	January 1994
Python 2.0 (Introduc+ed list comprehensions)	October 16, 2000
Python 3.0 (Emphasis on removing duplicative constructs and modules	December 3, 2008

Features of Python

- **Free and open-source**
- **Great Community Support**
- **Simple and Easier to Learn**
- **Simple Elegant Syntax**
- **Object-oriented Programming Language**
- **Huge Library Support to ease your work**
- **Portability :**
  You can move Python programs from one platform to another, and run it without any changes.
- **Extensible and Embeddable:**
  Suppose an application requires high performance. You can easily combine pieces of C/C++ or other languages with Python code.
- **A High-Level, Interpreted Language:**
  Unlike C/C++, you don't have to worry about daunting tasks like memory management, garbage collection and so on.

Reserve Keywords (33) in Python 3.7 (Latest Version)

and	**as**	**assert**	**break**	**class**	**continue**	**def**	**del**
elif	**else**	**except**	**False**	**finally**	**for**	**from**	**global**
if	**import**	**in**	**is**	**lambda**	**not**	**None**	**nonlocal**
or	**pass**	**raise**	**return**	**True**	**with**	**try**	**while**
yield							

All the keywords except True, False and None are in lowercase.

Other Important Keywords and Function which have their special meaning in programming

abs	**all**	**any**	**chr**	**dict**	**dir**	**eval**	**exit**	**file**	**float**
format	**input**	**int**	**max**	**min**	**next**	**object**	**open**	**print**	**quit**
range	**round**	**set**	**str**	**sum**	**tuple**	**type**	**vars**	**zip**	

Identifier

An identifier is a name given to entities like class, functions, variables, etc.

- The first character must be a letter or underscore (_).
- Additional characters may be alphanumeric or underscore.
- Names are case-sensitive.
- Identifier can be of any length.

Special Identifiers

Python also provides some special identifiers that use underscores. Their name will be of the form:

```
_xxx
__xxx__
__xxx
```

Comments

anything followed by hash (#) is comment

How to Create and run a Python Program?

1. Create a new .py file (e.g. myprogram.py) using any text editor that will contain the Python program (source code).
2. To run the code in command line

>>> **python myprogram.py**

or

>>> **python3 myprogram.py**

or

>>> **python2 myprogram.py**

depending upon version to version

Data Types in Python

- Numbers
- Strings
- Set
- List
- Tuple
- Dictionary

Import

`import math` or
`import library` or
`import user_defiend_module.py` etc are used same as header files in C.

Definitions inside a module can be imported to another module or the interactive interpreter in Python. We use the `import` keyword to do this.

Arithmetic operators

Operator	Meaning	Example
+	Addition	x + y
	unary plus	+5
-	Subtraction	x - y
	unary minus	-7
*	Multiply two operands	x * y
/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the right	x % y (remainder of x/y)
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	x**y (x to the power y)

Example Script: arith.py

```
x = 10
y = 3
#x + y = 13
print('x + y =',x+y)

#x - y = 7
print('x - y =',x-y)

#x * y = 30
print('x * y =',x*y)

#x / y = 3.3333333333333335
print('x / y =',x/y)

#x // y = 3
print('x // y =',x//y)

#x ** y = 1000
print('x ** y =',x**y)
```

Comparison Operators in Python Operator

>	x>y
<	x<y
==	x==y
!=	x!=y
>=	x>=y
<=	x<=y

Example Script: comp.py

```
x = 5
y = 7

# x > y is False
print('x > y  is',x>y)

# x < y is True
print('x < y  is',x<y)

# x == y is False
print('x == y is',x==y)

# x != y is True
print('x != y is',x!=y)

# x >= y is False
print('x >= y is',x>=y)

# x <= y is True
print('x <= y is',x<=y)
```

Some Important Interactive Commands

dir()
dir(__builtins__)
help()
len() #returns length of the parameter
type()
str()

Examples to be run in terminal:

Popular Operations on Data-types

Example Script: number.py

```
a=int(10)
b=10
c=float(10)
d=10.0
e=3+2j
print(a,type(a),'\n',b,type(b))
print(c,type(c),'\n',d,type(d),'\n',e,type(e))
```

Functions on Strings

Example Script: strFun.py

```
a="   Hello, How, are, you?   "
#removes leading and trailing white spaces
print(a.strip())

#removes trailing white space
print(a.rstrip())

#removes leading white space
print(a.lstrip())

#converts string to lower case
print(a.lower())

#converts string to upper case
print(a.upper())

#Search and Replace
print(a.replace("H",'X'))

#splits string into list
print(a.split())

#find the existing text and returns position or 2(in Python 2)
```

```
print(a.find('are'))

#return -1 for non existing text
print(a.find('You'))

#length of the string
print(len(a))
```

Bitwise Operator (Same as C)

&	**Bitwise AND**	**|**	**Bitwise OR**	**^**	**Bitwise XOR**
>>	**Right Shift**	**<<**	**left Shift**	**~**	**1's Compliment**

Example Script: bitWise.py

```
l=0&3
m=1<<10
n=32>>2
o=~5
p=6|2
q=6^6
print("----- Bit Wise Operations ------")
print("AND",l)
print("LS",m)
print("RS",n)
print("1's Compliment",o)
print("OR",p)
print("XOR",q)
```

Logical Comparison with 'is' and 'is not' ; Logical Search with 'in' and 'not in'

Example Script: logicalStr.py

```
o="hello"
p="hello"
#comparing strings logically
print(o is p)
print(o is not p)

#logical searching a member in list
mylist=["hllo", "how"]
print( "how" in mylist, "preltex" not in mylist)

#logical searching a member in Tuple
mytuple=("Hey", "preltex")
print( "preltex" in mytuple, "Hey" not in mytuple)

#logical searching a member in Dictionary (key only)
mydict={"M":"Monday", 'J':"January"}
print( "M" in mydict, "January" in mydict)
print( "Monday" in mydict, "J" in mydict)
```

Operations on List

Example Script myList.py

```
#creating a new list
k=list(("tea","coffee","lemon","old"))
print(k)

# append a new member at last
k.append("new")
print(k)

# removes a member
k.remove("old")
print(k)

#reverse the list
k.reverse()
print(k)

#creating a new list
kkk=list(("breakfst","lunch","dinner","new"))
```

```
#extending the list k with kkk
k.extend(kkk)
print(k)

#couting a particular member or string
print(k.count("new"))

#printing the index of first match
print(k.index("new"))

#insert the member at specified index
k.insert(1,"more")
print(k)

#pop-out a member at specified index
k.pop(5)
print(k)

#sorting the list
k.sort()
print(k)

#copy the list
kk=k.copy()
print(kk)
#clear the list, does not destroy only removes all members
k.clear()
print(k)
```

Operations on set

Set in Python is a data structure equivalent to sets in mathematics. Any immutable data type can be an element of a set: a number, a string, a tuple. Mutable (changeable) data types cannot be elements of the set.

Example Script : mySet.py

```
#creating a set
myset={"seb","kela","amrood"}

#alternate way to create a new set
s=set(("seb","kela","amrood"))
print(myset)
print(s)

#checking existance of a member in set
print("seb" in myset)

#comparing two indentical sets
print(myset is s)

#adding a new member
s.add("anar")
print(s)

#removing an existing member
s.remove("kela")
print(s)

#copying the set
newset=myset.copy();
print(newset)

#comparing two identical sets when copied
print(myset is newset)

#length of the set
print(len(s))
```

Exercise: try to use operations of list with set and find-out which one are compatible and which one are not.

Simple Operations on Dictionary

Example Script: dicOp.py

```
#Dictionary Operation

#creating a dictionary
dic={
   "India" :"New Delhi",
   "Haryana": "Chandigrah"
}
print(dic)

#printing a value using key
print(dic["India"])

#Alternate way to create a dictionary
di=dict(A="a",B="b",C="c")
print(di)

#adding a member to dictionary dic[key]=value
di["D"]="d"
print(di)

#deleting a member from dictionary using key
del(di["B"])
print(di)

#pop operation will delete the member based on key and returns its value
print(di.pop("C"))

#length of the dictionary
print(len(di))
```

Operations on Tuple

Example Script: myTuple.py

```
# Creating an empty tuple
myTuple = ()
print(myTuple)

#initializing
myTuple = (1, 2, 3)
print(myTuple)

# tuple with mixed datatypes
myTuple = (1, "Hello", 3.4)
print(myTuple)

# nested tuple
myTuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(myTuple)

# tuple can be created without parentheses known as tuple packing

myTuple = 3, 4.6, "dog"
print(myTuple)

# assigning values to variables from tuple, also known as tuple unpacking
a, b, c = myTuple
print(a)
print(b)
print(c)

#accessing the member
print(myTuple[0])
print(myTuple[2])

# See Result
myNewTuple = ("hello")
print(type(myNewTuple))

# Correct Result needs a comma at the end
myNewTuple = ("hello",)
print(type(myNewTuple))

# parentheses is optional
myNewTuple = "hello",
print(type(myNewTuple))
```

Accessing elements using Index

Example Script newTuple.py

```
myTuple = ('p','r','e','l','t','e','x')
# indexed from 1st to 3rd
#it always goes n-1th location
print(myTuple[1:4])

# up to element -3 (not included)
print(myTuple[:-3])

# from 4th to end
print(myTuple[4:])

# printing all elements: beginning to end
print(myTuple[:])
```

Exercise : try to use the similar operations on Strings and List etc

Some More Operations with List and Tuple

Example Script moreOp.py

```
#creating an integer list
myList=list((1,2,3,4))

#enumirating a list/tuple
enumList=enumerate(myList)
print(list(enumList))

#maths
print(sum(myList))
print(max(myList))
print(min(myList))

#Retuns true if all elements are True or list/tuple is empty
print(all(myList))

#Retuns true if any of the elements list/tuple are True;
print(any(myList))

#modify the list
myList.append(0)
#check now
print(all(myList))

#Clear the List
myList.clear()
#check now
print(all(myList))

#Return False if list/tuple is empty
print(any(myList))
```

Exercise: Try these operations with Tuple also

Some More Operations with Strings

Example Script moreOp2.py

```
# center(), ljust() and rjust()
a = "PreltexSolutions"

# Prints the string after centering with '.'
print ( a.center(30,'.'))

# Printing the string after ljust()
print ( a.ljust(30,'-'))

# Printing the string after rjust()
print ( a.rjust(30,'+'))
```

Operations like isalpha, isdigit, isnumeric, isspace

Example Script isOp.py

```
a=['a','b','c','7',' ',"456"] #last one is space
#checks if the character is alphabet: a-z or A-Z
print(a[0].isalpha())
#checks if the character/string is numeric
print(a[5].isnumeric())
#checks if digit
print(a[3].isdigit())
#checks if alpha-Numeric a-z A-Z 0-9
print(a[1].isalnum())
#checks if it is a white space
print(a[4].isspace())
```

Exercise: Try these operations with strings and tuple also

Format Strings

%d int
%f float/real
%s String
%c character
%% to print % symbol

Example Script: formatOp.py

```
x = 220/7
y = 456
z = "Hello How are you?"
w='a'
#float Values
print("Value of X: %f"%x)
print("Value of X: %0.2f"%x)
print("Value of X: %5.2f"%x)

#notice
print("Value of Y: %d"%x)
print("Value of Y: %5d"%x)

#int values
print("Value of Y: %d"%y)
print("Value of Y: %5d"%y)

#String
print("Value of Y: %s"%z)
print("Value of Y: %50s"%z)

#Character
print("Value of Y: %c"%w)
print("Value of Y: %5c"%w)
```

Syntax of print()

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

sep*= separator, by default it is one white space*
end*= end of line, by default it is new line*

Example Script: printOp.py

```
#use spe and end in print
#int object
a=10
#float object
b=20.3
#char object
c='a'
#string object
d="Hello"
#list object
e=list((1,2,3))
#tuple object
f=tuple((1,2,3))
#dictionary object
g=dict(a=1,b=2,c=3)

#default printing
print(a,b,c,d,e,f,g)

#printing with separator
print(a,b,c,d,e,f,g,sep='-')

#printing with separator and end character
print(a,b,c,d,e,f,g,sep=';',end='@')
```

List vs Tuple vs Dictionary vs Sets

All four are data structure in Python and it is always difficult to make a choice. Yet choosing the right data structure for our data is essential. Let's try:

Lists vs Tuples

Tuples are used to collect an immutable ordered list of elements. This means that:

- You can't add elements to a tuple. There's no **append()** or **extend()** method for tuples,
- You can't remove elements from a tuple. Tuples have no **remove()** or **pop()** method,
- You can find elements in a tuple since this doesn't change the tuple.
- You can also use the **in** operator to check if an element exists in the tuple.

So, if you're defining a constant set of values, use a tuple instead of a list. It will be faster than working with lists and also safer. The tuples contain "write-protect" data.

Example Script: ListVsTuple.py

```
#list vs tuple
#creating a list
a=[1,2,3]

#creating a tuple
b=('a','b','c')

#printing
print(a,type(a))
print(b,type(b))

#modifying list
a[2]=5
print(a)

#try to modify tuple; first run above code and then uncomment last two line and run
#b[2]='x'
#print(b)
```

Lists vs Dictionaries

- A list store an ordered collection of items, so it keeps some order. Dictionaries don't have any order.
- Dictionaries are known to associate each key with a value, while lists just contain values.
- Use a dictionary when you have an unordered set of unique keys that map to values.

***Note** that, because you have keys and values that link to each other, the performance will be better than lists in cases where you're checking membership of an element.*

Lists vs Sets

- Just like dictionaries, sets have no order in their collection of items. Not like lists.
- Set requires the items contained in it to be hashable, lists store non-hashable items.
- Sets require your items to be unique and immutable.
- Duplicates are not allowed in sets, while lists allow for duplicates and are mutable.
- You should make use of sets when you have an unordered set of unique, immutable values that are hashable.

Hashable:

in Python all immutable built-in objects are hashable

Non-Hashable:

in Python all mutable built-in objects are non-hashable

Hashable	Floats	Integers	Tuples	Strings	frozenset()
Non-Hashable	Lists	Sets	Dictionaries		

Python append() vs extend() Methods

extend() method takes an iterable (list, set, tuple or string, all objects which have sequential indexes starting from zero), and adds each element of the iterable to the list one at a time.

append() method, on the other hand, adds its argument to the end of the list as a single item, which means that when the append() function takes an iterable as its argument, it will treat it as a single object.

Example Script : ExtenExpan.py

```
#creating a list
a=[1,2,3,]

#making a copy
b=a.copy()

#printing lists
print(a)
print(b)

#append and extend methods
a.append([4,5])
b.extend([4,5])

#printing changed lists
print(a)
print(b)

#creating new list
c=[8,9,10]

#alternatively append and extend lists
a.append(c)
b.extend(c)

#printing changed lists
print(a)
print(b)
```

frozenset() Method

Frozen set is just an immutable version of a Python set object. While elements of a set can be modified at any time, elements of frozen set remains the same after creation. Due to this, frozen sets can be used as key in Dictionary or as element of another set. But like sets, it is not ordered (the elements can be set at any index).

Syntax:

frozenset(iterable) *where: iterable is an object, like list, set, tuple etc.*

Example Script:frorzSet.py

```
#creating a list
mylist = ['a', 'b', 'c']
print(mylist)

#extending a list when not frozen
mylist.extend(['d','e'])

#print new list
print(mylist)

#creating a frozen lsit
freezMyList = frozenset(mylist)
print(freezMyList)

#now uncomment following line and try to run
#freezMyList.extend(['f'])
print(freezMyList)
```

Methods of copying or clone in Python

- You can slice your original list and store it into a new variable: newList = oldList[:]
- You can use the built-in list() function: newList = list(oldList)
- Using copy library:
  - With the copy() method: newList = copy.copy(oldList)
  - If your list contains objects and you want to copy those as well, you can use copy.deepcopy(): copy.deepcopy(oldList)

Example Script: CopyCopy.py

```
#various methods of copy
a=[1,2,3]
#slice original list into new
b=a[:]
print(a)
print(b)

#using list function
c=list(a)
print(c)

#using copy library (import copy)
import copy
d=copy.copy(a)
print(d)

#alternatively
import copy as z
y=z.copy(a)
print(y)

#using copy.deepcopy(oldList) when list contains objects
x=[1,2,3,[4,5],['a','b'],"hello"]
e=copy.deepcopy(x)
print(e)
```

Spliting A Python List Into evenly sized Chunks

- we can resort to the zip() function in combination with iter()
- **iter()** is an iterator over a sequence (***like i in for loop***).
- **([iter(myList)] * N)** produces a list with ***N*** listiterator objects: each list iterator is an iterator of myList.
- the * that is passed to the zip() function before anything else unpacks the sequence into arguments so that you're passing the same iterator three times to the zip() function, and it pulls an item from the iterator each time.

How it works Lets take N=3 and x=[1,2,3,4,5,6,7,8,9]**:**

1. You will have three list iterator objects, which you can think of as:
   [1,2,3,4,5,6,7,8,9], [1,2,3,4,5,6,7,8,9], [1,2,3,4,5,6,7,8,9]
2. The first time, zip() will take one element of the list sequentially, which leaves you with:
   [1][2][3]
3. next time:
   [1, 4], [2, 5], [3, 6]
4. and then:
   [1, 4, 7], [2, 5, 8], [3, 6, 9]
5. finally all three are zipped and form like:
   [1, 2, 3], [4, 5, 6], [7, 8, 9]

Example Script: splitList.py

```
# Creating a List
myList = [1,2,3,4,5,6,7,8,9,10,11,12]
print(myList)

# Split into chunk of N elements zip(*[iter(myList)]*N)
newList = zip(*[iter(myList)]*3)

# Use `list()` to print the result of `zip()`
print(list(newList))
```

Formatted Output using argument Positions and use of `format` method

Example Script: formatOut1.py

```
#similar to shell script $1 $2 etc
firstName  = input("Enter First Name > ")

lastName  = input("Enter Last Name > ")

Orgnization  = input("Enter Organization Name > ")

print('{0}, {1} works at {2}'.format(firstName, lastName, Orgnization))

print('{1}, {0} works at {2}'.format(firstName, lastName, Orgnization))

print('FirstName {0}, LastName {1} works at {2}'.format(firstName, lastName, Orgnization))

print('{0}, {1} {0}, {1} works at {2}'.format(firstName, lastName, Orgnization))
```

Accessing Output strings arguments by name

Example Script: formatOut2.py

```
#Accessing Output strings arguments by name
name = input("Enter Name > ")

marks = input("Enter marks > ")

print('{x} got {p}% Marks'.format(x=name.title(), p=marks))
```

Number Format

Display Number in various Int format

Example Script: formatNum.py

```
# Taking Input
number = int(input("Enter number > "))

# 'd' is for integer number formatting
print("The number is:{:d}".format(number))

# 'o' is for octal number formatting, binary and hexadecimal format
print('Output number in octal format : {0:o}'.format(number))

# 'b' is for binary number formatting
print('Output number in binary format: {0:b}'.format(number))

# 'x' is for hexadecimal format
print('Output number in hexadecimal format: {0:x}'.format(number))

# 'X' is for hexadecimal format in Capital Letters
print('Output number in HEXADECIMAL: {0:X}'.format(number))
```

Display Number in various Real/float format

Example Script: formatFloat.py

```
# Taking Input
number  = float(input("Enter float Number > "))

#'f' is for float number arguments
print("Output Number in The float type :{:f}".format(number))

# padding for float numbers
print('padding for output float number{:5.2f}'.format(number))

#'e' is for Exponent notation
print('Output Exponent notation{:e}'.format(number))

#'E' is for Exponent notation in UPPER CASE
print('Output Exponent notation{:E}'.format(number))
```

Example Script: moreOnDict.py

```
#important operation on Dictionary
#creating a dictionary
d={'M':'Monday','T':'Tuesday','W':'Wednesday','Th':'Thrusday','F':'Friday','Sa':'Saturday'
,'Su':'Sunday'}

#creating a list of keys
k=d.keys()

#creating a list of values
v=d.values()

#Printing
print(d)
print(k)
print(v)

#lookup a key , with default value (if key is not found)
print(d.get('T','Not a Day'))
print(d.get('X','Not a Day'))
```

Any block in Python starts with a colon :

Note: *Indentation/spacing is very important in Python, equal spacing makes statements in same block*

if Block

```
if test:                    # note the colon ':'.
       statement1           # The statements following the
       statement2           #if are indented and executed
       statement3           # if the test is True.
```

if-else block Block

```
if test:                    # note the colon ':'.
       DoThis()             # DoThis() ... if test=True
else:                       # note the colon after 'else'
       DoThat()             # DoThat() ... if test=False
```

if-elif-else Block

```
If test:
       do this
elif another-test:
       otherwise do this
else:
       do that
```

Note: *that the indentation/spacing of the* ***if*** *and its corresponding* ***elif*** *and* ***else*** *keywords must all be the same.*

Example Script: ifelif.py

```
#if-elif-else
x=float(input("Enver First Number > "))
y=float(input("Enver Second Number > "))
if x > y :
   print(x," is greather than ", y)
elif x < y :
   print(x," is less than ",y)
else:
   print(x," is equal to ",y)
```

Different Types of Test we can do:

Comparisons

- var1 > var2 # greater than
- var1 >= var2 # greater than equal to
- var1 < var2 # Less than
- var1 <= var2 # less than equal to
- var1 == var2 # equal to
- var1 != var2 # not equal to

Sequence (list, tuple, or dictionary) membership

- var in sequence
- var not in sequence

Sequence length

- len(x)>0 # sequence has entries?

Has a value?

- var # not None (or zero or '')

Boolean value

- myObject # myObject ==True?

- not myObject # myObject ==False?
- var1 and var2 #logical AND
- var1 or var2 #logical OR

Validation

- var.isalpha()
- var.isalnum()
- var.isnumeric()
- var.isdigit()
- var.isdecimal()
- var.isspace()
- var.isprintable()
- var.isupper()
- var.islower()
- var.istitle()
- var.isidentifier()

Calculations

- (price*quantity) > 100.0 # cost>100?
- (cost-budget) > limit-budget # overbudget?

Example Script:ifis.py

```
#if and use of isxxxx() test
var=input("Enter any character > ")
if var.isalnum():
    print(var," is either alphabet or Number")
if var.isalpha():
    print(var," is an alphabet")
if var.isdigit():
    print(var," is a digit")
if var.isnumeric():
    print(var," is a Number")
if var.isdecimal():
    print(var," is a Decimal Number")
if var.isspace():
    print(var," is a White Space")
if var.isprintable():
    print(var," is a Printable Character")
if var.isupper():
    print(var," is an Upper Case")
if var.islower():
    print(var," is a Lower Case")
if var.istitle():
    print(var," is a Title Case")

#this is used to check if the identifier with valid name exist or not
#in this example this is always true as we are testing for var
if var.isidentifier():
    print(var," is a an identifier")
```

Exercise: Re-Write above program such that it employs nested if, if-else and ie-elif-else block

Exercise: Write a program to check if given number is positive, negative or zero, and if it is positive then check further weather it is even or odd.

Example Script for Logical Operations: logical.py

```
x=10
y=20
z=0
#logical AND
print ( "Logical AND".center(50,'-'))
if x and y:
    print("True")
else:
    print("False")

if y and z:
    print("True")
else:
    print("False")

#logical OR
print ( "Logical OR".center(50,'-'))
if x or y:
    print("True")
else:
    print("False")

if y or z:
    print("True")
else:
    print("False")

#logical not
print ( "Logical NOT".center(50,'-'))
if not x:
    print("True")
else:
    print("False")

if not y:
    print("True")
else:
    print("False")

if not z:
    print("True")
else:
    print("False")

#logical check for non-zero
print ( "Check if Valueis Non-Zero".center(50,'-'))
if x:
    print("True")
else:
    print("False")

if x:
    print("True")
else:
    print("False")

if z:
    print("True")
else:
    print("False")
```

Example Script for Search in String: findStr.py

```
myStr=input("Enter a Line > ")
findStr=input("Enter a Search Word > ")
if len(myStr)>0 and len(findStr)>0:
    if findStr in myStr:
        print(findStr," is availabe in ", myStr)
    if findStr not in myStr:
        print(findStr," is notavailabe in ", myStr)
else:
    print("Error: Either the String or the Search word or both are empty")
```

Taking multiple inputs

Using split() method; Syntax:

input().split(separator, maxsplit) #default separator is white space

Example Script: multiple_input.py

```
# taking four inputs at a time
#try using different objects, numbers, strings, list, tuple, dictionary etc
w, x, y, z = input("Enter four values: ").split()

print("First Value: ", w)
print("Second Value: ", x)
print("Third Value: ", y)
print("Fourth Value: ", z)
```

Example Script: multi_inp_sep.py

```
# taking four inputs separated by comma at a time
w, x, y, z = input("Enter four values : ").split(',')

print("First Value: ", w)
print("Second Value: ", x)
print("Third Value: ", y)
print("Fourth Value: ", z)
```

Example Script: multi_in_format.py

```
# multiple input and format method
a, b = input("Enter two values: ").split()

print("First number is {}".format(a))
print("Second number is {}".format(b))
print("First number is {} and second number is {}".format(a, b))
```

USING LIST COMPREHENSION

See how multiple inputs are taken using list:

Example Script: multi_inp_list.py

```
# taking multiple inputs using list and split them
w,x,y,z = [int(i) for i in input("Enter four values: ").split()]

print("First Value: ", w)
print("Second Value: ", x)
print("Third Value: ", y)
print("Fourth Value: ", z)
```

Example Script: multi_inp_list_split.py

```
# taking multiple inputs using list and split them using semicolon
w,x,y,z = [int(i) for i in input("Enter Four values separated by \';\' : ").split(';')]

print("First Value: ", w)
print("Second Value: ", x)
print("Third Value: ", y)
print("Fourth Value: ", z)
```

Example Script: multi_inp_list_format.py

```
# taking multiple inputs using list and format method
a, b = [int(i) for i in input("Enter two values: ").split()]

print("First number is {}".format(a))
print("Second number is {}".format(b))
print("First number is {} and second number is {}".format(a, b))
```

Mathematical functions : import math

import math

This module is always available. It provides access to the mathematical functions defined by the C standard. These functions cannot be used with complex numbers; use the functions of the same name from the cmath module if you require support for complex numbers

Number-theoretic and representation functions

math.ceil(*x*) : Return the ceiling of *x*, the smallest integer greater than or equal to *x*. If *x* is not a float, delegates to x.__ceil__(), which should return an Integral value.

math.copysign(*x, y*) : Return a float with the magnitude (absolute value) of *x* but the sign of *y*. On platforms that support signed zeros, copysign(1.0, -0.0) returns *-1.0*.

math.fabs(*x*) : Return the absolute value of *x, where x is real/float*.

math.factorial(*x*) : Return *x* factorial. Raises ValueError if *x* is not integral or is negative.

math.floor(*x*): Return the floor of *x*, the largest integer less than or equal to *x*. If *x* is not a float, delegates to x.__floor__(), which should return an Integral value.

math.fmod(*x, y*) : Return fmod(x, y), as defined by the platform C library. Note that the Python expression x % y may not return the same result.

math.frexp(*x*) : Return the mantissa and exponent of *x* as the pair (m, e). *m* is a float and *e* is an integer such that x == m * 2**e exactly. If *x* is zero, returns (0.0, 0), otherwise 0.5 <= abs(m) < 1. This is used to "pick apart" the internal representation of a float in a portable way.

math.fsum(*iterable*)
Return an accurate floating point sum of values in the iterable. Avoids loss of precision by tracking multiple intermediate partial sums,
for better understanding run following script

```
import math
#taking 0.1 10 times in a list
a=[0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1]

#normal sum
print(sum(a))

#using fsum
print(math.fsum(a))
```

math.gcd(*a*, *b*)
Return the greatest common divisor of the integers *a* and *b*. If either *a* or *b* is nonzero, then the value of gcd(a, b) is the largest positive integer that divides both *a* and *b*. gcd(0, 0) returns 0.

math.isfinite(*x*) : Return True if *x* is neither an infinity nor a NaN, and False otherwise. (Note that 0.0 *is* considered finite.)

math.isinf(*x*) : Return True if *x* is a positive or negative infinity, and False otherwise.

math.isnan(*x*) : Return True if *x* is a NaN (not a number), and False otherwise.

math.ldexp(*x, i*) : Return x * (2**i). This is essentially the inverse of function frexp().

math.modf(*x*) : Return the fractional and integer parts of *x*. Both results carry the sign of *x* and are floats.

math.remainder(*x, y*)
For finite *x* and finite nonzero *y*, this is the difference x - n*y, where n is the closest integer to the exact value of the quotient x / y. If x / y is exactly halfway between two consecutive integers, the nearest *even* integer is used for n. The remainder r = remainder(x, y) thus always satisfies abs(r) <= 0.5 * abs(y).

math.trunc(*x*)
Return the Real value *x* truncated to an Integral (usually an integer). Delegates to x.__trunc__().

Note that frexp() and modf() have a different call/return pattern than their C equivalents: they take a single argument and return a pair of values, rather than returning their second return value through an 'output parameter' (there is no such thing in Python).

For the ceil(), floor(), and modf() functions, note that *all* floating-point numbers of sufficiently large magnitude are exact integers. Python floats typically carry no more than 53 bits of precision (the same as the platform C double type), in which case any float *x* with abs(x) >= 2**52 necessarily has no fractional bits.

Power and logarithmic functions

math.exp(*x*)

Return *e* raised to the power *x*, where *e* = 2.718281… is the base of natural logarithms. This is usually more accurate than math.e ** x or pow(math.e, x).

math.expm1(*x*)

Return *e* raised to the power *x*, minus 1. Here *e* is the base of natural logarithms. For small floats *x*, the subtraction in exp(x) - 1 can result in a significant loss of precision; the expm1() function provides a way to compute this quantity to full precision: ***Try following code***

```
import math
#using exp function
print(math.exp(1e-2)-1)

#using expm1 function; see the diff in output
print(math.expm1(1e-2))
```

math.log(*x*[, *base*]) : With one argument, return the natural logarithm of *x* (to base *e*). With two arguments, return the logarithm of *x* to the given *base*, calculated as log(x)/log(base).

math.log1p(*x*) : Return the natural logarithm of *1+x* (base *e*). The result is calculated in a way which is accurate for *x* near zero.

math.log2(*x*) : Return the base-2 logarithm of *x*. This is usually more accurate than log(x, 2).

math.log10(*x*) : Return the base-10 logarithm of *x*. This is usually more accurate than log(x, 10).

math.pow(*x, y*) : Return x raised to the power y. Exceptional cases follow Annex 'F' of the C99 standard as far as possible. In particular, pow(1.0, x) and pow(x, 0.0) always return 1.0, even when x is a zero or a NaN. If both x and y are finite, x is negative, and y is not an integer then pow(x, y) is undefined, and raises ValueError.

Unlike the built-in ** operator, math.pow() converts both its arguments to type float. Use ** or the built-in pow() function for computing exact integer powers.

math.sqrt(*x*) : Return the square root of *x*.

Trigonometric functions

math.acos(*x*) : Return the arc cosine of *x*, in radians.

math.asin(*x*) : Return the arc sine of *x*, in radians.

math.atan(*x*) : Return the arc tangent of *x*, in radians.

math.atan2(*y, x*) : Return atan(y / x), in radians. The result is between -pi and pi. The vector in the plane from the origin to point (x, y) makes this angle with the positive X axis. The point of atan2() is that the signs of both inputs are known to it, so it can compute the correct quadrant for the angle. For example, atan(1) and atan2(1, 1) are both pi/4, but atan2(-1, -1) is -3*pi/4.

math.cos(*x*) : Return the cosine of *x* radians.

math.hypot(*x, y*) : Return the Euclidean norm, sqrt(x*x + y*y). This is the length of the vector from the origin to point (x, y).

math.sin(*x*) : Return the sine of *x* radians.

math.tan(*x*) : Return the tangent of *x* radians.

Angular conversion

math.degrees(*x*): Convert angle *x* from radians to degrees.

math.radians(*x*) :Convert angle *x* from degrees to radians.

Hyperbolic functions

Hyperbolic functions are analogs of trigonometric functions that are based on hyperbolas instead of circles.

math.acosh(*x*) : Return the inverse hyperbolic cosine of *x*.

math.asinh(*x*): Return the inverse hyperbolic sine of *x*.

math.atanh(*x*): Return the inverse hyperbolic tangent of *x*.

math.cosh(*x*): Return the hyperbolic cosine of *x*.

math.sinh(*x*): Return the hyperbolic sine of *x*.

math.tanh(*x*): Return the hyperbolic tangent of *x*.

Special functions

math.erf(*x*) : Return the error function at *x*. The erf() function can be used to compute traditional statistical functions such as the cumulative standard normal distribution:

math.erfc(*x*) : Return the complementary error function at *x*. The complementary error function is defined as 1.0 - erf(x). It is used for large values of *x* where a subtraction from one would cause a loss of significance.

math.gamma(*x*) : Return the Gamma function at *x*.

math.lgamma(*x*): Return the natural logarithm of the absolute value of the Gamma function at *x*.

Constants

math.pi : The mathematical constant *π* = 3.141592…, to available precision.

math.e : The mathematical constant *e* = 2.718281…, to available precision.

math.tau : The mathematical constant τ = 6.283185…, to available precision. Tau is a circle constant equal to 2π, the ratio of a circle's circumference to its radius.
math.inf : A floating-point positive infinity. (For negative infinity, use -math.inf.) Equivalent to the output of float('inf').
math.nan : A floating-point "not a number" (NaN) value. Equivalent to the output of float('nan').

List of Important Functions in Math Module

Function	Description
ceil(x)	Returns the smallest integer greater than or equal to x.
copysign(x, y)	Returns x with the sign of y
fabs(x)	Returns the absolute value of x
factorial(x)	Returns the factorial of x
floor(x)	Returns the largest integer less than or equal to x
fmod(x, y)	Returns the remainder when x is divided by y
frexp(x)	Returns the mantissa and exponent of x as the pair (m, e)
fsum(iterable)	Returns an accurate floating point sum of values in the iterable
isfinite(x)	Returns True if x is neither an infinity nor a NaN (Not a Number)
isinf(x)	Returns True if x is a positive or negative infinity
isnan(x)	Returns True if x is a NaN
ldexp(x, i)	Returns x * (2**i)
modf(x)	Returns the fractional and integer parts of x
trunc(x)	Returns the truncated integer value of x
exp(x)	Returns e**x
expm1(x)	Returns e**x - 1
log(x[, base])	Returns the logarithm of x to the base (defaults to e)
log1p(x)	Returns the natural logarithm of 1+x
log2(x)	Returns the base-2 logarithm of x
log10(x)	Returns the base-10 logarithm of x
pow(x, y)	Returns x raised to the power y
sqrt(x)	Returns the square root of x
acos(x)	Returns the arc cosine of x
asin(x)	Returns the arc sine of x
atan(x)	Returns the arc tangent of x
atan2(y, x)	Returns atan(y / x)
cos(x)	Returns the cosine of x
hypot(x, y)	Returns the Euclidean norm, sqrt(x*x + y*y)
sin(x)	Returns the sine of x
tan(x)	Returns the tangent of x
degrees(x)	Converts angle x from radians to degrees
radians(x)	Converts angle x from degrees to radians
acosh(x)	Returns the inverse hyperbolic cosine of x
asinh(x)	Returns the inverse hyperbolic sine of x
atanh(x)	Returns the inverse hyperbolic tangent of x
cosh(x)	Returns the hyperbolic cosine of x
sinh(x)	Returns the hyperbolic cosine of x
tanh(x)	Returns the hyperbolic tangent of x
erf(x)	Returns the error function at x
erfc(x)	Returns the complementary error function at x
gamma(x)	Returns the Gamma function at x
lgamma(x)	Returns the natural logarithm of the absolute value of the Gamma function at x
pi	Mathematical constant, the ratio of circumference of a circle to it's diameter (3.14159...)
e	mathematical constant e (2.71828..

Exercise: Try to use these functions and write some code.

Example Script: myMath.py

```
import math
x=22/7
#ceiling function
print(math.ceil(x))

#floor function
print(math.floor(x))

#float absolute function returns answer in float
print(math.fabs(x))

#Real value x truncated to an Integral/integer
print(math.trunc(x))
```

```
#return real and int part as pair
print(math.modf(x))

#return (mantissa,exponent) where mantissa is real and exponent is integer; x == m * 2**e
print(math.frexp(2))

#verify above result by math.ldexp(x, i)
print(math.ldexp(0.5, 2))

#floating remainder function, returns remainder in float
print(math.fmod(25,7))

#alternatively modulus operator returns remainder in integer
print(25%7)

#copysign(value,sign)
print(math.copysign(x,-0.0))

#factorial of n, here n=5
print(math.factorial(5))

#try math.remainder(x, y)and observe result
print(math.remainder(5,3))

#Nan Functions
print(math.isnan(5.3))
print(math.isnan(0))
#important float('nan')
print(math.isnan(float('nan')))

#try math.isinf(x)
print(math.isinf(0.0/5))
#positive infinity
print(math.isinf(float('inf')))
#negative infinity
print(math.isinf(float('-inf')))

#Greatest common divisor
print("GCD: ",math.gcd(5,15))

#pow(X,y)
print(math.pow(2,5))

#constant values
print("------Constant Values-------")
print(math.pi)
print(math.tau)
print(math.e)

print("-----Hyperbolic Functions-------")
y=math.pi/4;
print("val: ", y)
print(math.cosh(y))
print(math.sinh(y))
print(math.tanh(y))
print(math.acosh(1.32246))
print(math.asinh(0.86867))
print(math.atanh(0.65579))

print("------Angular Conversion-------")
print(math.degrees(y))
print(math.radians(30))


print("-----Trigonometric Functions-------")
y=math.pi/4;
print(math.cos(y))
print(math.sin(y))
print(math.tan(y))
print(math.acos(0.707))
print(math.asin(0.707))
print(math.atan(0.99999999999999))

print("------ Log Functions-------")
print(math.log10(100))
print(math.log2(16))
```

while loop : *very much similar to C*

Syntax:

```
loop Initializer
while test_conditon:
      do something...
      loop iterator
```

Example Script: while_loop.py

```
x=1
while x<=5:
    print("loop no: ",x)
    x+=1
```

break *and* ***continue*** *Statement:*

Example Script: while_break_cont.py

```
while True:
    x=input("Enter q to quit > ")
    if len(x)==0:
        continue
    if x.lower()=='q':
        print("Loop Terminated")
        break;
    else:
        continue
```

Working with Lists and Tuples

Example Script: while_list.py

```
#working with a list
a=[1,2,3,4,5,6,7]

i=0
while i<len(a):
    print("a[",i,"] = ",a[i])
    i+=1


#working witha a tuple
b=("seb","kela","santra","amrood")

i=0
while i<len(b):
    print("b[",i,"] = ",b[i])
    i+=1
```

Example Script: sum_of_n.py

```
#sum of first n natural numbers
print("sum of first n natural numbers")
n=int(input("Enter an Integer Number >"))

# initialize sum and counter
sum = 0
i = 1

while i <= n:
    sum = sum + i
    i+=1        # update counter

# print the sum
print("The sum is", sum)
```

while-else

Loops can have an else statement with loops in python. When condition becomes false, else is executed.
If the else statement is used with a for loop, the else statement is executed when the loop has exhausted iterating the list.

Example Script: while_else.py

```
#while-else
i = 0
while i <= 5:
    print("while loop >", i)
    i+=1
else:
    print("while else")
```

Fibonacci sequence for first n terms

Example Script:while_fibo.py

```
# Fibonacci sequence for first n terms
# Up to where?
n = int(input("Enter the Nth term for Fibonacci Series >"))
# pre initializing first two terms
n1 = 0
n2 = 1
i = 0

# check if the number of terms is valid
while n<=0:
   n=int(input("Please Enter Value greater than 0 > "))
else:
   print("Fibonacci sequence for ",n," terms:")
   while i < n:
       print(n1,end=', ')
       next = n1 + n2

       # updating the terms n1 and n2
       n1 = n2
       n2 = next
       i += 1
```

Armstrong Number

abcd... = $a^n + b^n + c^n + d^n + ...$

Example: 153 = 1*1*1 + 5*5*5 + 3*3*3

Numbers: *1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407, 1634, 8208, 9474, 54748,..*

Example Script: while_armstrong.py

```
#Armstrong Number of N Digits
number=int(input("Enter a Number > "))

# to find number of digits  change number variable to string
nth = len(str(number))

# initialize sum
sum = 0

# find the sum of the nth power of each digit
temp = number

while temp > 0:
   digit = temp % 10
   sum += digit ** nth
   temp //= 10

# display the result
if number == sum:
   print(number,"is an Armstrong number")
else:
   print(number,"is not an Armstrong number")
```

Armstrong numbers between Range

Example Script: while_armstrong_range.py

```
# Program to find Armstrong numbers between Range
lnum = int(input("Enter Lower Number: "))
hnum = int(input("Enter Higher Number: "))

if(lnum>hnum):
    #swap the numbers
    lnum = lnum+hnum
    hnum = lnum-hnum
    lnum = lnum-hnum

num=lnum
while num <= hnum:
        # order of number
        nth = len(str(num))
        # initialize sum
        sum = 0
        #find the sum of the nth power of each digit
        temp = num
        while temp > 0:
            digit = temp % 10
            sum += digit ** nth
            temp //= 10

        if num == sum:
            print(num)

        num += 1
```

Prime Number

Example Script: while_prime.py

```
# While Prime Number Program
print("Prime Number Checker".center(50,'-'))

#taking an initial value to enter into while loop
n=1

#taking a valid input
while n<2:
    n=int(input("Enter a Number Greater than 1> "))

p=1                #assume the number is prime
i=2                #initializing the loop with 2
while i<n/2:
    if n%i==0:  #test for divisibility
        p=0
        break
    i += 1

if p==1:
    print("%d is a Prime Number"%n)
else:
    print("%d is Not a Prime Number"%n)
```

Factorial

Example Script: while_fact.py

```
# While Factorial of a Number Program
print(" Find The Factorial".center(50,'-'))
n=0       #initial value to zero

#taking a valid input
while n<1:
    n=int(input("Enter a Number Greater than 0> "))


fact=1             #initialize factorial to zero

i=n
while i>0:
    fact *= i
    i -= 1

#printing the result
print("Factorial of {} is {}".format(n,fact))
```

Syntax:

```
for variable in sequence:
        do something
```

Example Script: forLoop.py

```
#for loop example
#creating a list
myList=['a','b','c','d']

for i in myList:
    print(i,end=" ")
```

Range Function

range(initial value, final value) #final value is excluded

Example Script: forRange1.py

```
#for in range simple example

for i in range(1,10):
    print(i)
```

Example Script: forRange2.py

```
#for in range example 2
#creating a list
myList=['a','b','c','d']

for i in range(len(myList)):
    print(myList[i],end=" ")
```

for - else

Example Script: for_else.py

```
#for else example
for i in range(1,10):
    print(i)
else:
    print("Loop is Closed")
```

Prime Numbers in Range

Example Script: for_primeRange.py

```
# Prime Number in Range
#setting initial value to enter into loop
lnum=1
hnum=1
print("Prime Number in Range".center(50,'-'))

#taking valid inputs
while lnum<2:
    lnum = int(input("Enter Lower Number Greater Than 1: "))
while hnum<2:
    hnum = int(input("Enter Higher Number Greater Than 1: "))

#correcting the Range
if(lnum>hnum):
    #swap the numbers
    lnum = lnum+hnum
    hnum = lnum-hnum
    lnum = lnum-hnum


print("Prime Numbers between",lnum," and ",hnum,":")

for num in range(lnum,hnum + 1):
```

```
        #using for-else loop
        for i in range(2,num):
            if (num % i) == 0:
               break
        else:
            print(num)
```

Factors of a Number

Example Script : for_NumFactors.py

```
# factors of a number
num=1
while num<2:
    num=int(input("Enter a Number Greater than 2 > "))

print("Factors of",num,":")
for i in range(1, num + 1):
       if num % i == 0:
           print(i)
```

Working with Dictionaries

Example Script: forDict.py

```
#for loop and working with Dictionaries

#creating a dictionary
states = {
        'H': "Haryana",
        'P':"Punjab",
        'HP':"Himachal Pradesh",
        'UP':"Uppar Pradesh",
        'B':"Bihar"
    }

#printing keys directly
print("Printing Keys".center(50,'-'))
for k in states:
    print(k)

#printing keys Alternatively using .keys() method
print("Alternatively Printing Keys".center(50,'-'))
for k in states.keys():
    print(k)

#printing values using .values() method
print("Printing Values".center(50,'-'))
for v in states.values():
    print(v)

#printing key and values using .items()
for k, v in states.items():
    print("State Code: {0} \t Name: {1}".format(k, v))
```

Example Sctipt: for_fact.py

```
#factorial of a Number

#taking initial values
fact=1
n=0

#taking valid input
while n<1:
    n=int(input("Enter a Number Greater than 1 >"))

for i in range(1,n+1):
    fact *= i

#printing the Result
print(fact)
```

For loop working with Lists

Example Script: for_squareInRangeList.py

```
#Exaple of creating a list with special purpose
#taking inputs
print("List of Squares between Range".center(50,'-'))
lnum = int(input("Enter First Number: "))
hnum = int(input("Enter Second Number: "))

#correcting the Range
if(lnum>hnum):
    #swap the numbers
    lnum = lnum+hnum
    hnum = lnum-hnum
    lnum = lnum-hnum

#creating empty list
squares=[]
print("List of Squares between {} and {}".center(50,'-').format(lnum,hnum))
for i in range (lnum,hnum+1):
    squares.append(i*i)

print(squares)

#alternatively
print("AlternativelyList of Squares between {} and {}".center(50,'-').format(lnum,hnum))
mySquare=[i*i for i in range(lnum,hnum+1)]
print(mySquare)
```

Counting the Vowels/any character

Example Script: forVowelCounter.py

```
# Count of each vowel in a string using dictionary
# creating string of vowels
vowels = 'aeiou'

# creating a blank string
myStr =""
while len(myStr)==0:
    myStr=input("Enter your String > ")

# make string case free using casefold() method
myStr = myStr.casefold()

# creating a dictionary where each vowel is a key with value = 0
vowelCount = {}.fromkeys(vowels,0)

# Counting the vowels
for char in myStr:
   if char in vowelCount:
       vowelCount[char] += 1

#printing the result
print(vowelCount)
```

Alternatively

Example Script: foranyChar.py

```
# creating string of desigred charactres
SearchString = 'aeiou'

myStr =""
while len(myStr)==0:
    myStr=input("Enter your String > ")

# make string case free using casefold() method
myStr = myStr.casefold()

# Counting the vowels/desired charactres
CountCharacters = {x:sum([1 for char in myStr if char == x]) for x in SearchString}

#printing the result
print(CountCharacters)
```

A function is a piece of program code that is:

- A self-contained coherent piece of functionality.
- Callable by other programs and modules.
- Passed data using arguments (if required) by the calling module.
- Capable of returning results to its caller (if required).

A function has two important keywords:

- **def** # used to create a new function / define a function
- **return** # used to return back to caller with either with none or as appropriate

Note: A function would work just fine without a return statement but it is good practice to use it

Return Statement Examples:

---

return	# None
return True	# True
return False	# False
return r1, r2, r3	# returns three values/Results
return dict(a=v1,b=v2)	# returns a dictionary similarly a list or tuple etc
return a+b+c	#returns result of an expression

Example Script: myFunction1.py

```
#def is keyword use to write a new function
def myFunction():
    #body of the function
    print("hello")
    #return statement (optional)
    return

print("Calling a Function".center(50,'-'))
#calling a function
myFunction()

print("Calling a Function in Loop".center(50,'-'))
#calling a function in loop
for i in range(1,6):
    myFunction()

print("What does a function return".center(50,'-'))
#what does a function return in Python
print("{}".format(myFunction()))
```

Imp: myFunction() may contain arguments in '()'

Example Sctipt: mySum.py

```
#creating new function
def mySum(a,b): #list of arguments
    #return result of expression
    return a+b

print("Calling a Function with literal Values".center(50,'-'))
#calling a function
print("{}".format(mySum(10,25)))

print("Calling a Function using Inputs".center(50,'-'))
#taking inputs as string
x =input("Enter a Number > ")
y =input("Enter another Number > ")

#taking two variables and intialise them to zero as default value
```

```
#by doing this if we enter empty while taking inputs, it will not prompt an error
if len(x)==0:
    var1=0
else:
    var1=int(x) #type casting variable

if len(y)==0:
    var2=0
else:
    var2=int(y) #type casting variable

print("Sum of both numbers".center(50,'-'))
print("{}".format(mySum(var1,var2)))
```

User Defined function to find out the length of a String

Example Script: myStrLen.py

```
#Creating a user defined function
def myLen(myString):
    strLen=sum([1 for i in myString])
    return strLen

#using a Fixed Sting directly
print("Length of My String is {}".format(myLen("How are You?")))

#using a Fixed Sting using a variable
strVar="How are You?"
print("Length of My String is {}".format(myLen(strVar)))

#Taking input
print("Length of My String is {}".format(myLen(input("Enter a String > "))))
```

Recursion or Recursive Function

Example Script: myFact.py

```
#factorial using recursion
def myFact(num):
   if num == 1 or num == 0:
       return num
   else:
       return num*myFact(num-1)

#taking inputs
x=int(input("Enter a Number > "))

#calling function
n=myFact(x)

#printing results with auto assignment
print("Factorial of {} is {}".format(x,n))
```

main() in Python

what is __name__ = __main__ ?

underscore underscore **name** *underscore underscore* = *underscore underscore* **main** *underscore underscore*

We can decide whether we want to run the script. Or that we want to import the functions defined in the script.

- When we run the script containing, the __name__ variable equals __main__. The Python interpreter runs the "source file" as the main program, it sets the special variable (__name__) to have a value ("__main__").
- When we execute the main function, it will then read the "if" statement and checks whether **__name__** does equal to **__main__.**
- When we import the script, it will contain the name of the script. The Python interpreter reads a source file, it will execute all the code found in it.
- In Python "**if__name__== "__main__"** allows you to run the Python files either as **reusable** modules or **standalone** programs.

Example Script as standalone: name_main.py

```
#crating a user defined function
def myFunction():
    print ('__name__ = ' + __name__)
    return

#creating a main function
def main():
    myFunction()
    return

#if we run the script following statement results in True
#if we use this script in other script using import then
#following statement resutls in False
if __name__ == '__main__':
    main()
```

Result: __name__ = __main__

Example Script as import to previous script: myNameMain.py

```
import name_main as nm
nm.myFunction()
```

Result: __name__ = name_main #name of the imported script

We can Directly create a main function and can also use it

Example Script : mainOnly.py

```
#creating a main function
def main():
    print("This is main")
    return

#calling a main function
main()
```

Example Script: mainOnlyImport.py

```
#importing mainOnly
import mainOnly as myMain
myMain
```

alternatively:

```
#importing mainOnly
import mainOnly as myMain
#if __name__ == '__main__':
#    myMain
```

OR other way is:

```
#importing mainOnly
import mainOnly as myMain
myMain.main()
```

Exercises:

Create a menu which first takes a range of numbers and then it ask if you want to

- **Check prime numbers in range**
- **Check Armstrong numbers in range**
- **Square in range**

and then print accordingly

each option must be implemented with functions

Default Value of Arguments in a Function

Example Script: defaultValueFunction.py

```
#defalut Value Argument Function
def myFunc(a=0,b=0,c=0):
    return a+b+c

#let's take three integer objects
x = 15
y = 10
z = -2
print("calling function with no argument".center(50,'-'))
print("Output without Argument : {} ".format(myFunc()))

print("\n"+"calling function with all three arguments".center(50,'-'))
print("Output with all three Arguments : {} ".format(myFunc(x,y,z)))

print("\n"+"calling function with any two arguments".center(50,'-'))
print("Output with any two Arguments : {} ".format(myFunc(x,y)))
print("Output with any two Arguments : {} ".format(myFunc(y,z)))
print("Output with any two Arguments : {} ".format(myFunc(z,x)))

print("\n"+"calling function with only one argument".center(50,'-'))
print("Output with only one Argument : {} ".format(myFunc(x)))
print("Output with only one Argument : {} ".format(myFunc(y)))
print("Output with only one Argument : {} ".format(myFunc(z)))
```

Creating a Dictionary using Function

Example Script: dictFunction.py

```
#creating an empty dictionary
mydict={}

def dictFunc(k,v):
    mydict.update({k:v})
    return mydict

#creating an exit character and empty key and value objects
x=""
myKey=""
myVal=""

while x.casefold() != 'q':
    #clearing the exit character and other variables
    x=""
    myKey=""
    myVal=""

    #printing dictionary
    if len(mydict.keys())==0:
        print("Dictionary is Empty")
    else:
        print("Used Keys are : {}".format(mydict.keys())+"\n")

    #taking unique key input
    while len(myKey)==0 or myKey in mydict.keys():
        myKey=input("Enter a Unique Key > "+"\n")

    #taking value
    while len(myVal)==0:
        myVal=input("Enter a Value for {} :".format(myKey)+"\n")

    #calling dictionary function
    dictFunc(myKey,myVal)

    #prompt for exit or continue
    x=input("Press Enter to Continue or q to quit > ")


print("\n"+"Dictionay is:".center(50,'-'))
print(mydict)
```

File Modes:

Mode	Description
'r'	Open a file for reading. (default)
'w'	Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
'a'	Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.
't'	Open in text mode. (default)
'x'	Open a file for exclusive creation. If the file already exists, the operation fails.
'b'	Open in binary mode.
'+'	Open a file for updating (reading and writing)

There can be combination of Modes like: wb, rb, ab, wt, w+, r+, a+ etc.

Opening a file with any mode/s is similar to C

fp = open("test.txt",mode = 'r',encoding = 'utf-8')

or

with open("test.txt",mode = 'r',encoding = 'utf-8') as fp

Other Python File Methods

Method	Description
close()	Close an open file. It has no effect if the file is already closed.
detach()	Separate the underlying binary buffer from the TextIOBase and return it.
fileno()	Return an integer number (file descriptor) of the file.
flush()	Flush the write buffer of the file stream.
isatty()	Return True if the file stream is interactive.
read(*n*)	Read almost *n* characters form the file. Reads till end of file if it is negative or None.
readable()	Returns True if the file stream can be read from.
readline(*n*=-1)	Read and return one line from the file. Reads in at most *n* bytes if specified.
readlines(*n*=-1)	Read and return a list of lines from the file. Reads in at most *n* bytes/characters if specified.
seek(*offset*, *from*)	Change the file position to *offset* bytes, in reference to *from* (start=0, current=1, end=2).
seekable()	Returns True if the file stream supports random access.
tell()	Returns the current file location.
truncate(*size*=None)	Resize the file stream to *size* bytes. If *size* is not specified, resize to current location.
writable()	Returns True if the file stream can be written to.
write(*s*)	Write string *s* to the file and return the number of characters written.
writelines(*lines*)	Write a list of *lines* to the file.

Let's Start: first create a file *demo.txt*

Example Script: readFile.py #Reading the Entire file

```
#file Read program
# fp = open("demo.txt", mode = 'r') # encoding is optional
fp = open("demo.txt", mode = 'r', encoding = 'utf-8')

#reading entire file
print(fp.read())

#closing the file
fp.close()
```

Example Script: readNextLine.py #default first line

```
#Reading next line program
fp = open("demo.txt", mode = 'r') # encoding is optional

#reading first line of the file
print(fp.readline())
#try using end="" to avoid extra new line character
#print(fp.readline(),end="")
```

```
#reading next line of the file
print(fp.readline())

#closing the file
fp.close()
```

Example Script: readMultipleLine.py

```
#Reading line by line program
fp = open("demo.txt", mode = 'r') # encoding is optional

for i in range(0,9):
    print(fp.readline(),end="")

#closing the file
fp.close()
```

Example Script: numberOfLines.py

```
#couting number of lines in a file
fileName='demo.txt'
numLines = 0

with open(fileName, 'r') as fp:
    for line in fp:
        numLines += 1
print("Number of lines in {} are: {}".format(fileName,numLines))
fp.close()
```

Example Script: readSingleLine.py #default first line

```
#Reading line by line program
fileName='demo.txt'
numLines = 0
with open(fileName, 'r') as fp:
    for line in fp:
        numLines += 1
fp.close()

fp = open(fileName, mode = 'r')
for i in range(0,numLines):
    print(fp.readline(),end="")
fp.close()
```

Example Script: *readingNcharacters.py*

```
#reading N characters
fileName='demo.txt'
fp=open(fileName, 'r')

print("Is file {} Readable ? : {} ".format(fileName,fp.readable()))
print("Is file {} Seekable ? : {} ".format(fileName,fp.seekable()))
print("Is file {} Writable ? : {} ".format(fileName,fp.writable()))
print("Is file Stream of {} interactive with tty? : {} ".format(fileName,fp.isatty()))

#reading first N characters
print(fp.read(5))

#or reading next N
print(fp.readline(5))

#fp.seek(position, from where= 0=start/1=relative to current/2=end of file)
fp.seek(0,0)    #seeking 0th char of firt line
print(fp.readline())

fp.seek(0,1) #seeking 0th char of next line or seeking from current pos
print(fp.readline())

#size of the file
fp.seek(0,2)    #seeking last character
print("file contains {} characters".format(fp.tell()))
print("Size of file is {} bytes".format(fp.tell()))
```

Creating to file and Writing to it

Example Script: fileWrite.py

```
#creating a file and writing to it
myFile="new.txt"

#opening in write mode
fp=open(myFile,'w')
print("File Descriptor assigned is {} ".format(fp.fileno()))

#creating a blank string variable/object and escape char
myStr=""
esc=""

while esc.casefold() != 'q':

    #re-assigning empty value
    myStr=""
    esc=""

    #taking input
    myStr=input("Enter a Line to File > "+"\n")
    fp.write(myStr+"\n") # try without newline character
    #or Try this line
    #fp.writelines(myStr+"\n")

    esc=input("Press Enter to Continue and q to Quit > "+"\n")

fp.close()
```

Appending a File:

Example Script: appendFile.py

```
#taking an existing file
myFile="new.txt"

#opening in append mode
fp=open(myFile,'a+')
print("File Descriptor assigned is {} ".format(fp.fileno()))

#creating a blank string variable/object and escape char
myStr=""
esc=""

while esc.casefold() != 'q':

    #re-assigning empty value
    myStr=""
    esc=""

    #taking input
    myStr=input("Enter a Line to File > "+"\n")
    fp.write(myStr+"\n") # try without newline character
    #or Try this line
    #fp.writelines(myStr+"\n")

    esc=input("Press Enter to Continue and q to Quit > "+"\n")

fp.close()
```

Example Script: listFile.py

```
#Writing a list to a file
#file name
myFile="listFile.txt"
#creating a list
myList=["Once there was a Crow.", "He was very thirsty.", "He flew eveywhere in search of water"]

#wrting list into file
fp = open(myFile, "w")

for line in myList:
    fp.write(line+"\n")

fp.close()
```

Alternatively using writelines()

```
#Writing a list to a file
#file name
myFile="listFile.txt"

#creating a list
myList=["Once there was a Crow.\n","He was very thirsty.\n","He flew here and there in search of water\n"]

#wrting list into file
fp = open(myFile, "w")

#Writing all lines at once
fp.writelines(myList)
fp.close()
```

Dealing with Image File

JPEG headers contain information like height, width, number of color (grayscale or RGB) etc.

Marker Name	Marker Identifier	Description
SOI	0xd8	Start of Image
APP0	0xe0	JFIF application segment
APPn	0xe1 – 0xef	Other APP segments
DQT	0xdb	Quantization Table
SOF0	0xc0	Start of Frame
DHT	0xc4	Huffman Table
SOS	0xda	Start of Scan
EOI	0xd9	End of Image

SOF0 (Start Of Frame 0) marker:

Field	Size	Description
Marker Identifier	2 bytes	0xff, 0xc0 to identify SOF0 marker
Length	2 bytes	This value equals to 8 + components*3 value
Data precision	1 byte	This is in bits/sample, usually 8 (12 and 16 not supported by most software).
Image height	2 bytes	his must be > 0
Image Width	2 bytes	This must be > 0
Number of components	1 byte	Usually 1 = grey scaled, 3 = color YcbCr or YIQ 4 = color CMYK
Each component	3 bytes	Read each component data of 3 bytes. It contains, (component Id(1byte)(1 = Y, 2 = Cb, 3 = Cr, 4 = I, 5 = Q), sampling factors (1byte) (bit 0-3 vertical., 4-7 horizontal.), quantization table number (1 byte)).

Remarks: JFIF uses either 1 component (Y, greyscaled) or 3 components (YCbCr, sometimes called YUV, colour).

Example Script: imgPixles.py

```
#working with image file
#creating a function to deal with image
def myImg(filename):
   # open image for reading in binary mode
   with open(filename,'rb') as img:

       # height of image (in 2 bytes) is at 164th position
       img.seek(163)

       # read the 2 bytes
       a = img.read(2)

       # calculate height (2 bytes)
       height = (a[0] << 8) + a[1]

       # next 2 bytes is width
       a = img.read(2)

       # calculate width (2 bytes)
       width = (a[0] << 8) + a[1]

   print("Image Resolution is {} x {}: ".format(width,height))

myImg("abc.jpg")
```

Important Functions related to Directories

Example Script: osDir.py

```
#important Function Related to Directory
import os

#a String Object for New Dir Name
newDir='preltex'
updatedDir='Preltex'

#print current working directory using os.getcwd
print("Print Current Working Directory".center(50,'-'))
print(os.getcwd())

print("\n"+"Create New Directory if does not exists".center(50,'-'))
#creaking a new directory
#os.mkdir('name_of_dir')
if os.path.isdir(newDir): #testing if directory already exists
    print("Directory Exists")
else:
    os.mkdir(newDir)
    print("Directory {} Created".format(newDir))

#change working directory
print("\n"+"Change Working Directory to {}".center(50,'-').format(newDir))
os.chdir(newDir)
#print current working directory
print(os.getcwd())

print("\n"+"Resolving Path to Parent Dir".center(50,'-'))
#resolving path of current directory
from pathlib import Path
d = Path().resolve().parent
#also try following code
#print(d)
#print(d.parent)
#print(d.parent.parent)

#changing directory to parent directory
os.chdir(d)
print(os.getcwd())

print("\n"+"Rename {} to {}".center(50,'-').format(newDir,updatedDir))
#renaming an existing directory
os.renames(newDir,updatedDir)
#testing
if os.path.isdir(newDir): #testing if directory already exists
    print("Direcoty {} Exists".format(newDir))
elif os.path.isdir(updatedDir): #testing if directory name is updated
    print("Now Direcoty is {} ".format(updatedDir))

print("\n"+"Creating List of Directories and Files".center(50,'-'))
#creating list of files and directories of current directory
dirList=os.listdir(os.getcwd())
#printing list of files and directories in a directory
for x in dirList:
    print(x)

print("\n"+"Checking Before Removing Directory".center(50,'-'))
#testing before removing
if os.path.isdir(updatedDir): #testing if directory already exists
    print("Directory {} Exists".format(updatedDir))
```

```
print("\n"+"Removing Directory".center(50,'-'))
#Removing Directory
os.rmdir(updatedDir)

print("\n"+"Checking After Removing Directory".center(50,'-'))
#testing after Removal
if os.path.isdir(updatedDir): #testing if directory already exists
    print("Direcoty {} Exists".format(updatedDir))
else:
    print("Directory {} is Removed".format(updatedDir))
```

Important Functions related to Files

Example Script: osFile.py

```
#important Function Related to Files
import os
#creating a new file name and updated file name objects
newFile="myFile.txt"
updatedFile="myUpFile.txt"

print("Creating A New File".ljust(50,'.'))
#creating a file if does not exist
if os.path.isfile(newFile):
    print("File {} Already Exists".format(newFile))
else:
    fp=open(newFile,'w')
    fp.write("Hello There\nHow are You?")
    fp.close()
    print("File Successfully Created")

#rename a file
if os.path.isfile(newFile):
    os.rename(newFile,updatedFile)
    print("File Renamed from {} to {}".format(newFile,updatedFile))
else:
    print("File already Renamed to {}".format(updatedFile))

#test if operation is done
if os.path.isfile(updatedFile):
    print("File {} Exists".format(updatedFile))

#removing a file
if os.path.isfile(updatedFile):
    print("Removing {}".ljust(50,'.').format(updatedFile))
    os.unlink(updatedFile)
    #or try this
    #os.remove(updatedFile)


#test if remove operation is done
if os.path.isfile(updatedFile):
    print("File {} Exists".format(updatedFile))
else:
    print("File {} Removed".format(updatedFile))
```

Important Terms about Object Oriented Programming in Python

1. Class:
A class in OOPs is a blueprint through which objects are created. It is a container which has two important components:

- **Attribute/s :** A Class Attribute is a Python variable that belongs to class
- **Method/s:** A method is nothing but a function associated with class

2. Object:
It is an instance of the class which actually implements the blueprint of the class. for sake of simplicity we can understand an object as variable which has all properties of the class it belongs to. An object has two properties:

- **Attribute** : An instance/object **attribute** is a **Python** variable belonging to one, and only one, object.
- **Behavior**: Behavior is defined by the method of a class, that how it behaves with different data set/values.

3. self:
As we call a function with parameters, we have to provide list of parameters/arguments to it. Likewise when we create an object or we call a method from object's class, then the first argument passed to that method or attribute initializer is object (as it is like a variable) itself. so self keyword represents object itself.

4. __init__ : (*similar to **new** constructor in Java***)**
It is a reserved and special method of Python to initialize the attributes of the class. We can understand it as a constructor and it is automatically called when an object of a class is created. When __init__ is run/called class attributes are initialized and assigned to object.

To check class of any object use type() method

Example Script : type.py

```
#Creating Objects of Different Classes
w=True
x=10
y=2.3
z='hello'
myList=['a','b']
myTuple=(1,2,3)
mySet={'a',1,'hello'}
myDict={'a':'apple','b':'ball'}

#Checking Classes
print(" Checking Class using type()".center(50,'-'))
print(type(w))
print(type(x))
print(type(y))
print(type(z))
print(type(myList))
print(type(myTuple))
print(type(mySet))
print(type(myDict))

print("\n"+"Alternatively Checking Class".center(50,'-'))
print(w.__class__.__name__)
print(x.__class__.__name__)
print(y.__class__.__name__)
print(z.__class__.__name__)
print(myList.__class__.__name__)
print(myTuple.__class__.__name__)
print(mySet.__class__.__name__)
print(myDict.__class__.__name__)
```

This script shows that everything in Python (or any OOP Language) is an object of some class

Creating a Class and an Object

Example Script: firstClass.py

```
#Creating first calss
class myClass:
    pass # pass is used to tell the Python Language interpreter
         # that this code block will be written later

#creating an object/instance of myClass
x=myClass()

#checking if it works or not
print(type(x))
print(x.__class__.__name__)
```

***Imp:** Python uses **class** keyword to create a new class like it uses **def** for creating new function/method.*

Creating a Full Class

Example Script: MyCar.py

```
#Creating a Class MyCar( )
class MyCar:
   "Car Blueprint"
   #attributes of the class is defined as
   def __init__(self, model, color, company, speedLimit):
      self.color = color.title()
      self.company = company.title()
      self.speedLimit = speedLimit
      self.model = model.title()

   #methods of the class MyCar defines behavior of the car
   def start(self): #takes one argument i.e. object of the class
      return "started".title()

   def stop(self): #takes one argument i.e. object of the class
      return "stopped".title()

   def accelarate(self): #takes one argument i.e. object of the class
      return  "accelarating...".title()

   def changeGear(self, gear): #takes Two arguments: object, gear value
      return "gear changed to {}".title().format(gear)

#creating objects and initializing them with arguments
print(" Creating Objects ".center(50,'-'))
maruthiSuzuki = MyCar("ertiga", "white", "suzuki", 60)
audi = MyCar("A6", "red", "audi", 80)

#printing object properties for Maruti
print("\n"+" Printing Object Properties for Martuti ".center(50,'-'))
print("Company : "+maruthiSuzuki.company)
print("Model : "+maruthiSuzuki.model)
print("Color : "+maruthiSuzuki.color)
print("Top Speed : ",maruthiSuzuki.speedLimit)
print(maruthiSuzuki.start())
print(maruthiSuzuki.accelarate())
print(maruthiSuzuki.changeGear("Three"))
print((maruthiSuzuki.stop()))

#printing object properties for Audi
print("\n"+" Printing Object Properties for Audi ".center(50,'-'))
print("Company : "+audi.company)
print("Model : "+audi.model)
print("Color : "+audi.color)
print("Top Speed : ",audi.speedLimit)
print(audi.start())
print(audi.accelarate())
print(audi.changeGear(5))
print((audi.stop()))

#printing class
print("\n"+" Printing Class ".center(50,'-'))
print(type(maruthiSuzuki))
print(maruthiSuzuki.__class__.__name__)
```

Take Another Example of Employee Class where:

Attributes of employees are:

- First Name
- Last Name
- Salary

Methods are:

- Full Name
- email-Id

Example Script: **EmployeeClass.py**

```
#Creating an employee class
class Employee:
    "attributes are defined here"
    def __init__(self,firstName,lastName,salary):
        self.firstName = firstName.title()
        self.lastName = lastName.title()
        self.salary = salary

    #"behaviors are defined here"
    def fullName(self):
        return '{} {}'.format(self.firstName,self.lastName)

    def email(self):
        return
'{}.{}@preltex.in'.format(self.firstName.casefold(),self.lastName.casefold())

#creating objects
emp1 = Employee('suresh','mehta',25000)
emp2 = Employee('ramesh','dinker',25500)
emp3 = Employee('ananat','gupta',65000)

#printing attributes
print("\n"+" Printing Attributes ".center(50,'-'))
print(emp1.firstName)   #attributes do not have () brackets at the end
print(emp2.lastName)
print(emp3.salary)

print("\n"+" Accessing Methods ".center(50,'-'))
print(emp1.fullName())
print(emp1.email())
print(emp2.fullName())
print(emp2.email())
print(emp3.fullName())
print(emp3.email())

print("\n"+" Printing Class ".center(50,'-'))
print(emp1.__class__.__name__)
print(emp2.__class__.__name__)
print(emp3.__class__.__name__)

#printing object locations
print(" Printing Object Locations ".center(50,'-'))
print(emp1)
print(emp2)
print(emp3)

print("\n"+" Trying Method without () ".center(50,'-'))
print(emp1.fullName)
print(emp1.email)
print(emp2.fullName)
print(emp2.email)
print(emp3.fullName)
print(emp3.email)

#now match the addresses with object-addresses
```

Python built-in class attributes with description

__doc__ : *Returns the class documentation string, if defined.*

__module__ : *Return the name of the module in which the class is defined.*

__name__ : *Return the name of the class.*

__dict__ : *Returns a dictionary of classes namespace.*

Example Script: StudentClass.py

```
class Student:
  "A Student Class with Important Attributes"

  #Constructor to initialize object with attributes
  def __init__(self,fname,lname,gender,age):
      self.fname = fname.title()
      self.lname = lname.title()
      self.age = age
      if gender.casefold()=='m':
          self.gender = 'Male'
      elif gender.casefold()=='male':
          self.gender = 'Male'
      elif gender.casefold()=='f':
          self.gender = 'Female'
      elif gender.casefold()=='female':
          self.gender = 'Female'
      else:
          self.gender = 'Not-Disclosed'

  #Defining methods
  def FullName(self):
      return '{} {}'.format(self.fname,self.lname)

  def email(self):
      return '{}.{}@myschool.com'.format(self.fname.casefold(),self.lname.casefold())

print(" Printing Builtin Class Attributes ".center(100,'-'))
print ('Student.__name__ = ',Student.__name__)
print ('Student.__doc__ = ',Student.__doc__)
print ('Student.__module__ = ',Student.__module__)
print ('Student.__dict__ = ',Student.__dict__)

#creating an object:
stu1 = Student('karan','arora','m',19)

print("\n"+" Try using Class Attributes with its Object ".center(100,'-'))
#__name__ is bound to class and not with objects
#print ('stu1.__name__ = ',stu1.__name__)
print ('stu1.__doc__ = ',stu1.__doc__)
print ('stu1.__module__ = ',stu1.__module__)
print ('stu1.__dict__ = ',stu1.__dict__)

print("\n"+" Let\'s Access Methods ".center(100,'-'))
print(stu1.email())
print(stu1.FullName())

print("\n"+" Let\'s Try Attributes of Objects ".center(100,'-'))
print(stu1.fname)
print(stu1.lname)
print(stu1.gender)
print(stu1.age)
```

Exercise: code a class Bike in Python with

Attributes: Company, Model, Color, Top Speed, NM-Torque, Engine-CC etc

Behavior: Start the engine, Stop the engine Speed up, Change gear, Stall

Exercise: code a class Restaurant-Customer in Python with and offer a customer menu based on time input

Attributes: name, time of entry in 24 hrs format

Behavior: breakfast, lunch, dinner, snacks etc

Inheritance:
Inheritance is the basic and most powerful property of any OOP. It is about creating a new class with or without modification to an existing class. Usually the new or derived class is known as child class and old or base class is known as parent class.

Inheritance Syntax:

```
class ParentClass:
      Body of Parent class

class ChildClass(ParentClass):
      Body of Child class
```

Import to refer: *While creating a Parent Class, it must be generic and should have expansion capabilities.*

Let's Take an Example of Polygon which can have n sides; using polygon we can write any child class for say triangle.
Example Script: Polygon.py

```
#Creating a Polygon Class as Parent Class
class Polygon:
    "The Polygon Class"
    # initializing attributes
    def __init__(self, noOfSides):
        self.n = noOfSides
        self.sides = [0 for i in range(noOfSides)]

    #creating methods
    def inputSides(self):
       self.sides = [float(input("Enter Side No "+str(i+1)+" : ")) for i in range(self.n)]

    def dispSides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])

#Creating Triangle as Derived or Child Class-------------------------
class Triangle(Polygon): # see the parameter
    "Triangle Area and Perimeter"

    #initializing attributes of Triangle objects
    def __init__(self):
        Polygon.__init__(self,3)

    #defining methods
    #Area method
    def Area(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        import math
        return math.sqrt((s*(s-a)*(s-b)*(s-c)))

    #Perimeter Method
    def Perimeter(self):
        a, b, c = self.sides
        # calculate the perimeter
        return a + b + c
#Triangle Class Ends here---------------------------------------

#Creating Rectangle as Derived or Child Class--------------------
class Rectangle(Polygon): # see the parameter
    "Rectangle Area and Perimeter"

    #initializing attributes of Rectangle objects
    def __init__(self):
        Polygon.__init__(self,2)

    #defining methods
    #Area method
    def Area(self):
        a, b, = self.sides
        return (a*b)

    #Perimeter Method
```

```
    def Perimeter(self):
        a, b, = self.sides
        # calculate the perimeter
        return (2*(a+b))
#Rectangle Class Ends here---------------------------------------

#Creating Object for Triangle
print(" Creating Object for Triangle ".center(100,'-'))
t=Triangle()
t.inputSides()
t.dispSides()
print("Area of Triangle is : {}".format(t.Area()))
print("Perimeter of Triangle is : {}".format(t.Perimeter()))

#Creating Object for Rectangle
print("\n"+" Creating Object for Rectangle ".center(100,'-'))
r=Rectangle()
r.inputSides()
r.dispSides()
print("Area of Rectangle is : {}".format(r.Area()))
print("Perimeter of Rectangle is : {}".format(r.Perimeter()))

print("\n"+" Checking Class ".center(100,'-'))
print(t.__class__.__name__)
print(r.__class__.__name__)
```

Method Overriding

As we can see __init__ method is defined in both parent as well as child class. While creating object of a child class, __init__ method of the child class is preferred over base or parent class. There can be any other method also with same name and has the same preference. For say there could have been an Area method in Polygon class also, but while creating an instance/object of Triangle or Rectangle, the object would have preferred Area method of Triangle or Rectangle over the Polygon. This is known as method overriding. Proper way to do so is:

```
Polygon.__init__(self,3) is equivalent to super().__init__(3)
```

Built-in functions to check inheritance

- **isinstance()**
- **issubclass()**

Example Script: Super.py

```
#Creating a Polygon Class as Parent Class
class Polygon:
    "The Polygon Class"
    # initializing attributes
    def __init__(self, noOfSides):
        self.n = noOfSides
        self.sides = [0 for i in range(noOfSides)]

    #creating methods
    def inputSides(self):
        self.sides = [float(input("Enter Side No "+str(i+1)+" : ")) for i in
range(self.n)]

#Creating Triangle as Derived or Child Class-------------------------
class Triangle(Polygon): # see the parameter
    "Triangle Area and Perimeter"

    #initializing attributes of Triangle objects and overriding Polygon __init__
    def __init__(self):
        super().__init__(3)

    #Overriding Parent method
    def inputSides(self):
        self.sides = [float(input("Enter Side No "+str(i+1)+" : ")) for i in
range(self.n)]
    #Triangle Class Ends here---------------------------------------

#creating an object
t=Triangle()
print(isinstance(t,Triangle))
print(isinstance(t,Polygon))
print(isinstance(t,float))
print(isinstance(t,object))
```

```
print(issubclass(Polygon,Triangle))
print(issubclass(Triangle,Polygon))
```

Multiple Inheritance

Like C++ in Python also a Child or derived class can inherit properties of multiple parent or base classes. Multiple inheritance is forbidden in Java. A attribute is searched first in the current class. If not found, the search continues into parent classes in depth.

Syntax:

```
class Parent1:
  pass
class Parent2:
  pass
class NewChild(Parent1,Parent2):
  pass
```

Multilevel Inheritance

When a child class having a Parent class and that Parent class also having a Parent class and so on... is known as multilevel inheritance. What properties a derived/child class can have depends upon its level of inheritance in hierarchy.

Syntax:

```
class MasterParent:
  pass
class Parent(MasterParent):
  pass
class Child(Parent):
  pass
```

Method Resolution Order (MRO)

Every class in Python is derived from the **class** object. It is the most base type in Python. So all other classes, either built-in or user-defines, are derived classes and all objects are instances of object class. It ensures that a class always appears before its parents and in case of multiple parents, the order is same as tuple of base classes.
MRO of a class can be viewed as the **__mro__** attribute (Returns Tuple) or **mro()** method (Returns List).

Example Script: MultiDrived.py #A Multiple Inheritance

```
#A Multi-Derived Class
class ParentA:
    pass
class ParentB:
    pass
class Child(ParentA,ParentB):
    pass
#Testing for MRO : Child, ParentA, ParentB, Object
print(Child.__mro__)
print(Child.mro())
```

Example Script: MultLevel.py #A Multi-Level Inheritance

```
#A Multi-Level Class
class SuperParent:
    pass
class Parent(SuperParent):
    pass
class Child(Parent):
    pass

#Testing for MRO : Child, Parent, SuperParent, Object
print(Child.__mro__)
print(Child.mro())
```

Example Script: MultiMulti.py #A Multiple-Multi-Derived Inheritance

```
#A Multi Level, Multi Derived Class
class A:    pass
class B:    pass
class C:    pass
class AB(A,B):
    pass
class ABC(AB,C):
    pass
class Child(ABC):
    pass
#Testing for MRO : Child, ABC, A,B,C, Object
print(Child.__mro__)
print(Child.mro())
```

Operator Overloading

When an operator behaves differently with different data-types (objects of different classes). For example, the + operator performs arithmetic addition on two numbers, merge two lists and concatenate two strings.

Class functions that begins with double underscore __ are called special functions in Python. To overload the + sign, we will need to implement __add__() function in the class. Similarly __str__() method in class make sure that how things will get printed.

Binary Operators	
Operator	**Sepcial Method**
+	__add__(self, other)
–	__sub__(self, other)
*****	__mul__(self, other)
/	__truediv__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)
******	__pow__(self, other)

Comparison Operators	
Operator	**Sepcial Method**
<	__lt__(self, other)
>	__gt__(self, other)
<=	__le__(self, other)
>=	__ge__(self, other)
==	__eq__(self, other)
!=	__ne__(self, other)

Assignment Operators	
Operator	**Sepcial Method**
-=	__isub__(self, other)
+=	__iadd__(self, other)
***=**	__imul__(self, other)
/=	__idiv__(self, other)
//=	__ifloordiv__(self, other)
%=	__imod__(self, other)
****=**	__ipow__(self, other)

Unary Operators	
Operator	**Sepcial Method**
+	__pos__(self, other)
~	__invert__(self, other)
–	__neg__(self, other)

Example Script : OperatorOverloadingComplexNumber.py

```
#Operator Overloading addition of two complex numbers using +
#Creating a Complex Number Class
class CompNum:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

     # adding two objects
    def __add__(self, num):
        r=self.x + num.x
        i=self.y + num.y
        if i==0:
            return r
        else:
            return r,i

    def __str__(self):
        return self.x, self.y

#Creating Objects
a=CompNum(1,3)      #passing two arguments
b=CompNum(7,2)

c="Hello!"+" How? are you?" #adding strings

d=CompNum(5)        #passing only one argument
e=CompNum(6)

#prining Results
print("Addition : {}".format(a+b))
print("Addition : {}".format(d+e))
print("{}".format(c))
```

Encapsulation: Data and Method Hiding

Encapsulation is a property of an Object Oriented Programming Language, which allows a class to hide its data and methods by restricting its direct access/ modification outside the class itself. Although we can indirectly modify data of an encapsulated variable and can also access encapsulated method with the help of other methods in the class.

- An encapsulated variable/ data/ method is known as private.
- Non-encapsulated are public variables/ data/ methods.

How to create a Private variable or Method in Python?
Using double underscore
private variable : __a=10
private method: __myFun():

Encapsulated Variable
Example Script: encapVar.py

```
#creating an encapsulated variable in myClass
class myClass:
    #encapsulated data variable
    __x=20
    __y="Oh this is Encapsulated!"

    #creating a public function
    def myFunc(self):
        return '{} {}'.format(self.__x,self.__y)

#creating object
a=myClass()

#accessing the Public function myFunc()
print(a.myFunc())

#now try to run following code un-commenting one line at a time
#This will give AttributeError which tell us that encapsulated variable is not seen
outside

#print(myClass.__x)
#print(myClass.__y)
#print(a.__x)
#print(a.__y)
```

Encapsulated Method
Example Script: envapMethod.py

```
#creating an encapsulated method in myClass
class myClass:
    #encapsulated function
    def __myPrivate(self):
        return "Oh This is a Encapsulated"

    #creating a public function
    def myFunc(self):
        return self.__myPrivate()

#creating object
a=myClass()

#accessing the Public function myFunc()
print(a.myFunc())

#now try to run following code un-commenting one line at a time
#This will give AttributeError which tell us that encapsulated variable is not seen outside

#print(myClass.__myPrivate)
#print(a.__myPrivate)
```

Indirectly modifying value of a private/encapsulated variable in-side class

```
#creating an encapsulated variable in myClass
class myClass:
    #encapsulated data variable
    __x=20

    #creating a Public modify method
    def modifyEncap(self,updatedVal):
        self.__x = updatedVal

    #creating a public function
    def dispVal(self):
        return self.__x

#creating object
a=myClass()

#accessing the Public function dispVal()
print(a.dispVal())

#Modifying value using modifyEncap() function
a.modifyEncap(50)

#Again accessing the Public function dispVal()
print(a.dispVal())
```

Important:

According to the official Python documentation, **__repr__** is a built-in function used to compute the "official" string reputation of an object, while **__str__** is a built-in function that computes the "informal" string representations of an object.

Example Script:

```
#Creating a Class which has __repr__ and __str__ built-in methods
class myClass:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __repr__(self):
        return "__repr__ Method of myClass a:%s b:%s" % (self.a, self.b)

    def __str__(self):
        return "__str__ Method of myClass a is %s and b is %s" % (self.a, self.b)

# Creating Object and initializing
t = myClass(1024, 'Hello')

# Printing Results
print(t)          # This calls __str__()
print([t])        # This calls __repr__()
print({t})        # This calls __repr__()
```

if no __str__ method is defined then __repr__ method is used.

Example Script:

```
#Creating a Class which has __repr__ and __str__ built-in methods
class myClass:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __repr__(self):
        return "__repr__ Method of myClass a:%s b:%s" % (self.a, self.b)

# Creating Object and initializing
t = myClass(1024, 'Hello')

# Printing Results
print(t)          # This calls __str__()
print([t])        # This calls __repr__()
print({t})        # This calls __repr__()
```

Abstraction

Abstraction is a concept or an idea not associated with any specific instance. So in Object Oriented Programming it is a concept which allows a class to have a method which is only declared and not defined. Precisely an abstract method does not have any implementation details.

Abstract classes are written when we know what to do, but don't know how to do it. At this point we write abstract class with abstract method (one or more). Any abstract class cannot be instantiated, so we must extend them with derived (or child) class to implement abstract methods.

Python already have an abstract Module ***abc*** which have and Abstract class **ABC** and an abstract method with name ***abstractmethod***. So we need to import ABC and abstractmethod from this module, to do so we need to write following line before we start writing any abstract class:

```
from abc import ABC,abstractmethod
```

Let's Create One:

Example Script: AbstractClass.py

```
#importing abstract class ABC and abstractmethod from abc.py
from abc import ABC,abstractmethod
#creating an abstract class by extending ABC from abc.py
class myClass(ABC):
    "Abstract Class"
    @abstractmethod #must be written
    def dispaly(self):
        pas

#creating a child class which can extend myClass and impliment display
class Child(myClass):
    "Impliment display method"
    def dispaly(self):
        return "Abstraction is Implemented"

#Creating objects
a=Child()

#accesing method display
print(a.dispaly())

#Try to instantiate myClass() by creating some object
#Now try to un-comment following code
#b=myClass()
```

What happen when there are more than one @abstractmethod?

Either we have to implement all @abstractmethod in extended child class or if we extend them through different derived classes then following example is important:

Example Script: abstractMore.py

```
#importing abstract class ABC and abstractmethod from abc.py
from abc import ABC,abstractmethod
#creating an abstract class by extending ABC from abc.py
class myClass(ABC):
    "Abstract Class"
    @abstractmethod #must be written
    def display(self):
        pas

    @abstractmethod
    def sqr(self,a): #to square a number
        pass

#creating a child class which can extend myClass and implement display
class DispChild(myClass):
    "Implement display method"
    def display(self):
```

```
        return "Abstraction is Implemented"

    # must keep the sqr method and may not necessarily implement it
    def sqr(self,a):
        pass

#creating a child class which can extend myClass and implement sqr
class SqrChild(myClass):
    "Implement sqr method"
    def sqr(self,a):
        return (a*a)
    # must keep the display method and may not necessarily implement it
    def display(self):
        pas

#Creating objects of DispClass()
a=DispChild()
#accessing method display
print(a.display())


#Creating objects of SqrClass()
b=SqrChild()
#accessing method sqr
print(b.sqr(5))

#Try to instantiate myClass() by creating some object
#Now try to un-comment following code
#b=myClass()
#print(a.sqr())
#print(b.display())
```

We can Implement more than one extended/child class from an abstract class.

Example AnimalType.py

```
#importing abstract class ABC and abstractmethod from abc.py
from abc import ABC,abstractmethod
#Creating an Abstract Class Animal
class Animal(ABC):
    @abstractmethod
    def Type(self):
        pass

#Creating a child Class Cow from Animal
class Cow(Animal):
    #implementing Abstract Method Type
    def Type(self):
        return "Domestic"


#Creating a child Class Dog from Animal
class Dog(Animal):
    #implementing Abstract Method Type
    def Type(self):
        return "Pet"

#Creating a child Class Lion from Animal
class Lion(Animal):
    #implementing Abstract Method Type
    def Type(self):
        return "Wild"

#Creating Objects
Rita=Cow()
Tomy=Dog()
Alex=Lion()

#Printing Types of Animals
print(Rita.Type())
print(Tomy.Type())
print(Alex.Type())
```

Polymorphism in Python, Overriding and Overloading

Polymorphism is an object-oriented programming concept that refers to the ability of a variable, function or object to take on multiple forms. A language that features polymorphism allows developers to program in the general rather than program in the specific. Polymorphism is more like operator overloading, where one operator behaves differently.

Polymorphism lets us define methods in the child class that have the same name as the methods in the parent class. This process of re-implementing a method in the child class is known as Method Overriding.
In nutshell Polymorphism means same function(method) name but different behavior for different inputs.

Method Overriding

- Two different classes have same method name but different behavior.
- A Parent class and a Child class have same method name, so an object of child class overrides the method of parent class and executes the method of its own class (i.e. child class).

Example Script: methodOverriding.py

```
#Parent Class
class Parent:
    "This is a Parent Class"
    def Disp(self):
        return "This is a Parent Class Disp Method"

#Child Class
class Child(Parent):
    "This is a Child Class extended from Parent Class"
    def Disp(self):
        return "This is a Child Class Disp Method"

#Printing Results for Parent Class
p=Parent()
c=Child()

#Printing Results
print(p.__class__.__name__)
print(p.Disp())

#Printing Results for Child Class
print(c.__class__.__name__)
print(c.Disp())
```

What if Child class does not have a Disp method

Example Script:

```
#Parent Class
class Parent:
    "This is a Parent Class"
    def Disp(self):
        return "This is a Parent Class Disp Method"

#Child Class
class Child(Parent):
    "This is a Child Class extended from Parent Class"

#Creating Objects
p=Parent()
c=Child()

#Printing Results for Parent Class
print(p.__class__.__name__)
print(p.Disp())

#Printing Results for Child Class
print(c.__class__.__name__)
print(c.Disp())
```

Variable Overriding; Example Script: VarOverriding.py

```
#Parent Class
class Parent:
    "This is a Parent Class"
    val=50

#Child Class
class Child(Parent):
    "This is a Child Class extended from Parent Class"
    val=100

#Printing Results for Parent Class
p=Parent()
c=Child()
#Printing Results
print(p.__class__.__name__)
print(p.val)
print(c.__class__.__name__)
print(c.val)
```

*Similarly if Child class does not have the attribute **val**, then automatically **val** or Parent class is executed*

Method Overloading : When same method behaves differently with different parameters

Example Script: methodOverloading.py

```
#Method Overloading

class Hello:
    "Say Hello!"
    def sayHello(self, name=None):
        if name==None:
            print("Hello")
        else:
            print("Hello {}".format(name))

#Creating object
ram=Hello();
ram.sayHello()            #Calling sayHello() without parameters
ram.sayHello("Ram")       #Calling sayHello() with parameters
```

Example Script: MethodOverLoading_Polygon.py

```
#Polygon Method Overloading
class Polygon:
    "Area of any Polygon based on parameters"

    def Area(self,*args):
        count=0
        for i in args:
            count += 1

        #Assuming for Square
        if count==1:
            import math
            return math.pow(args[0],2)

        #Assuming for Rectangle
        if count==2:
            return args[0]*args[1]

        #assuming for Triangle
        if count==3:
            s=(args[0]+args[1]+args[2])/2
            import math
            return math.sqrt((s*(s-args[0])*(s-args[1])*(s-args[2])))

#Creating objects
sqr=Polygon()
rect=Polygon()
trig=Polygon()

#Printing Area
print("Area of Squr  is {}".format(sqr.Area(3)))
print("Area of Rect  is {}".format(rect.Area(3,4)))
print("Area of Trig  is {}".format(trig.Area(3,4,5)))
```

There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

Syntax Errors:

They are also known as parsing errors. The parser repeats the offending line and displays a little 'arrow' pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow.

Example:

```
while True print('Hello world')
               ^
SyntaxError: invalid syntax
```

Exceptions (Run time errors):

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal. Most exceptions are not handled by programs.

Example:

```
x=5
y=0
print("{}",format(x/y))
```

This Code will Generate exception: **ZeroDivisionError: division by zero**

Example:

```
x=5
print("{}".format(x+z))
```

This Code will Generate exception: **NameError: name 'z' is not defined**

Example:

```
x=5
print("{}".format(x+"Hello"))
```

This Code will Generate exception: **TypeError: unsupported operand type(s) for +: 'int' and 'str'**

Above ZeroDivisionError, NameError, TypeError are built-in exceptions. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions.

Exception Handling:

exception **Exception**

All built-in, non-system-exiting exceptions are derived/extended from this class. All user-defined exceptions should also be derived from this class. It is possible to write programs that handle selected exceptions.

Most of the run-time-errors are due to wrong or invalid user inputs. we need to write an efficient code in such a way that Python interpreter does not throw an exception in a way it does, this is called exception handling. To handle an exception we need to find the error, following process will help us to understand this:

- ✓ Spotting an Error (may be by Built-in exception)
- ✓ Take the Caution where user input can go wrong **(try Statement)**
- ✓ Write a block to fix the error if thrown by the Python interpreter **(except-Block)**
- ✓ Write a block where everything is going good **(else-Block)**
- ✓ Write a final block which executes weather your code throw an exception or not **(finally-Block)**

This is How it will Work:

try: First (try to) run this code, to get an exception.
except: Run this code when an exception is thrown.
else: Run this code when no exception is thrown.
finally: Always run this code

Example Script: FirstExceptionHandlingProgram.py

```
#Our First Exception Handling Code
x=int(input("Enter A Number : "))
#try to enter y as 0 (zero)
y=int(input("Enter Other Number : "))
try:
    x/y # We try here to catch the exception
except Exception:
    print("Ops! You Are Trying an Illegal Division")
else:
    print("{} divided by {} is : {} ".format(x,y,x/y))
finally:
    print("Exception Handling is Successful")
```

How to print Interpreter generated Exception?

Example Script: printError.py

```
#Printing Python Generated Exception
x=int(input("Enter A Number : "))
#try to enter y as 0 (zero)
y=int(input("Enter Other Number : "))
try:
    x/y # We try here to catch the exception

#Creating object of class Exception
except Exception as err:
    print("Ops! You Are Trying an Illegal Division")
    #printing error
    print("Python Error >>> {}".format(err))
else:
    print("{} divided by {} is : {} ".format(x,y,x/y))
finally:
    print("Exception Handling is Successful")
```

Handling Specifically a built-in exception here ZeroDivisionError

Example Script: Handling ZeroDivisionError.py

```
#ZeroDivisionError Exception Handling Code
x=int(input("Enter A Number : "))
#try to enter y as 0 (zero)
y=int(input("Enter Other Number : "))
try:
    x/y # We try here to catch the exception

#See the name of Built-in Exception and its Object
except ZeroDivisionError as err:
    print("Ops! You Are Trying an Illegal Division")
    #printing error
    print("Python Error >>> {}".format(err))
else:
    print("{} divided by {} is : {} ".format(x,y,x/y))
finally:
    print("Exception Handling is Successful")
```

What if we enter a Non-Int Value: Python will Throw Value Error Exception

Example Script: MultipleExceptionHandling.py

```
#Multiple Exception Handling
try:
    #try to enter non int value
    x=int(input("Enter A Number : "))
    y=int(input("Enter Other Number : "))
    x/y # We try here to catch the exception
except ZeroDivisionError:
    print("Ops! You Are Trying an Illegal Division")
except ValueError:
    print("Ops! Invalid Value. Only Integers are expected")
except Exception as err:
    print("Python Says >>> {}".format(err))
else:
    print("{} divided by {} is : {} ".format(x,y,x/y))
finally:
    print("Exception Handling is Successful")
```

We can Also Write Multiple Exception in Single Block:

Example Script: MultiExpt.py

```
#Multiple Exception Handling in Single Block
try:
    #try to enter non int value
    x=int(input("Enter A Number : "))
    y=int(input("Enter Other Number : "))
    x/y # We try here to catch the exception
#Multiple Exception in One Block
except (ZeroDivisionError, ValueError):
    print("Operation Failed Due to Invalid Entry")
#for all other exceptions
except Exception as err:
    print("Python Says >>> {}".format(err))
```

```
else:
    print("{} divided by {} is : {} ".format(x,y,x/y))
finally:
    print("Exception Handling is Successful")
```

Raising Exceptions

We can explicitly raise exception using ***raise*** keyword:

Example Script: raiseException.py

```
#raising Exception explicitly
while True:
    try:
        #try to enter a negative Number or a non int value
        a = int(input("Enter a Positive Integer Number: "))
        if a < 0:
            #raising exception with custom error message
            raise ValueError("We Expect a Positive Integer Number Only!")
    except ValueError as err:
        print("Python Error >>> {}".format(err))
    except KeyboardInterrupt:
        #try to close program using ctrl+C in terminal
        print("Unexpected End of Program")
    else:
        print("{} is Positive Number".format(a))
        break
    #finally:                    #finally is an optional block
```

Nested Try and Except with File handling

Example: tryExceptWithFileHandling.py

```
#Try-Except block with file-Handling
myFile="hello.txt"
try:
    #creating a file handler
    fp=open(myFile,'r',encoding='utf-8')
except Exception as err:
    print("Something went wrong... Python Says >>> {}".format(err))
else:
    print(fp.read())
finally:
    #always close file in finally block
    fp.close()
```

Now Try This:

```
#Try-Except block with file-Handling
myFile="hello.txt"
try:
    #creating a file handler
    fp=open(myFile,'r',encoding='utf-8')
except Exception as err:
    print("Something went wrong... Python Says >>> {}".format(err
else:
    print(fp.read())
finally:
    #always close file in finally block
    try:
        fp.close()
    except NameError:
        pass
```

User Defined Custom Exception Handling

Example Script: CustomExceptionHandling.py

```
#Creating a custom user-Defined Exception
class myExcept(Exception):
   "Base Class"
   pass

class NegatieValueError(myExcept):
    pass

class PositiveValueError(myExcept):
    pass

while True:
```

```
    try:
       a = int(input("Enter a number: "))
       if a < 0:
           raise NegatieValueError
       if a > 0:
           raise PositiveValueError
    except NegatieValueError:
        print("You Have Entered a Negative Value")

    except PositiveValueError:
        print("You Have Entered a Positive Value")
    except ValueError:
        print("Program Expect only integer Inputs")
    else:
        print("Congratulations You Have Entered Correct Value")
        break
print("We ar Out of While")
```

Built-in Exceptions

Exception	Cause of Error
AssertionError	Raised when assert statement fails.
AttributeError	Raised when attribute assignment or reference fails.
EOFError	Raised when the input() functions hits end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raise when a generator's close() method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits interrupt key (Ctrl+c or delete).
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in local or global scope.
NotImplementedError	Raised by abstract methods.
OSError	Raised when system operation causes system related error.
OverflowError	Raised when result of an arithmetic operation is too large to be represented.
ReferenceError	Raised when a weak reference proxy is used to access a garbage collected referent.
RuntimeError	Raised when an error does not fall under any other category.
StopIteration	Raised by next() function to indicate that there is no further item to be returned by iterator.
SyntaxError	Raised by parser when syntax error is encountered.
IndentationError	Raised when there is incorrect indentation.
TabError	Raised when indentation consists of inconsistent tabs and spaces.
SystemError	Raised when interpreter detects internal error.
SystemExit	Raised by sys.exit() function.
TypeError	Raised when a function or operation is applied to an object of incorrect type.
UnboundLocalError	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
UnicodeError	Raised when a Unicode-related encoding or decoding error occurs.
UnicodeEncodeError	Raised when a Unicode-related error occurs during encoding.
UnicodeDecodeError	Raised when a Unicode-related error occurs during decoding.
UnicodeTranslateError	Raised when a Unicode-related error occurs during translating.
ValueError	Raised when a function gets argument of correct type but improper value.
ZeroDivisionError	Raised when second operand of division or modulo operation is zero.

RegEx:

Starting from UNIX, most of the programming language support regular expressions. Basic rules of regular Expression are same. A regular expression, regex or regexp (sometimes called a rational expression) is a sequence of characters that define a search pattern. Usually this pattern is used by string searching algorithms for "find" or "find and replace" operations on strings, or for input validation. It is a technique developed in theoretical computer science and formal language theory.

The concept arose in the 1950s when the American mathematician Stephen Cole Kleene formalized the description of a regular language. The concept came into common use with Unix text-processing utilities. Since the 1980s, different syntaxes for writing regular expressions exist, one being the POSIX standard and another, widely used, being the Perl syntax.

Regular Expressions uses meta-characters to design search pattern. They have special meanings.

Common Meta-Characters being used in RegExp

Meta-Character	Description	Example
[]	A set of characters	"[a-m]"
\	Signals a special sequence	"\d"
.	Any character (except newline character)	"he..o"
^	Starts with	"^hello"
$	Ends with	"world$"
*	Zero or more occurrences	"aix*"
+	One or more occurrences	"aix+"
{}	Exactly the specified number of occurrences	"al{2}"
|	Either or , Alternation, OR Logic	"falls|stays"

Special Sequences	Description	Example
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"
\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s"
\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the specified characters are at the end of the string	"Spain\Z"

Set	Description
[arn]	Returns a match where one of the specified characters (a, r, or n) are present
[a-n]	Returns a match for any lower case character, alphabetically between a and n
[^arn]	Returns a match for any character EXCEPT a, r, and n
[0123]	Returns a match where any of the specified digits (0, 1, 2, or 3) are present
[0-9]	Returns a match for any digit between 0 and 9
[0-5][0-9]	Returns a match for any two-digit numbers from 00 and 59
[a-zA-Z]	Returns a match for any character alphabetically between a and z, lower case OR upper case
[+]	In sets, +, *, ., |, (), $,{} has no special meaning, so [+] means: return a match for any + character in the string

Quantifiers	Description
+	Matches the preceding pattern element one or more times.
?	Matches the preceding pattern element zero or one time.
?	Modifies the *, +, ? or {M,N}'d regex that comes before to match as few times as possible.
*****	Matches the preceding pattern element zero or more times.
{M,N}	Denotes the minimum M and the maximum N match count. N can be omitted and M can be 0: {M} matches "exactly" M times; {M,} matches "at least" M times; {0,N} matches "at most" N times. x* y+ z? is thus equivalent to x{0,} y{1,} z{0,1}.

import re

Python have module re to work with regular expressions.

Designing a regular expression pattern: ^P..L..X$

This pattern matches all the 7 character combinations which Start with 'P' followed by any two characters, then' L' followed by any two characters and then 'X'.

Let's Try our First Program with re.match(pattern, string):

```
#Testing a match using match(pattern, string) method
import re
inputString="PRELTEX"
searchString='^P..L..X$'

if re.match(searchString,inputString):
    print("Match Found")
else:
    print("Match Fail")
```

Searching in a List:

```
import re
inputList=['PRELTEX','PRRLTTX','PTTTTTX']
searchString='^P..L..X$'
for inputString in inputList:
    if re.match(searchString,inputString):
        print("Match-Pass")
    else:
        print("Match-Fail")
```

Matching Valid Mobile No

```
#matching valid mobile No using match(pattern, string) method
import re
inputList=['+919896010011','0919896010011','91-9896010011','+9198960-10011','+91-98960-
10011']
searchString='\+?0?91-?[7-9][0-9]{4}-?[0-9]{5}'
for inputString in inputList:
    if re.match(searchString,inputString):
        print("{} Match-Pass".format(inputString))
    else:
        print("{} Match-Fail".format(inputString))
```

re.findall(pattern,string)

Matching all instances of pattern in a String and returns list of matches:

```
#Extracting all instances of a pattern from a string; in form of list
import re

inputString = 'one 1 two 2 three 3 four thousand 4000'
#search decimal digits
sarchPattern = '\d+'    #try this>>> sarchPattern = '\d'
#or
#sarchPattern = '[0-9]+'

matches = re.findall(sarchPattern, inputString)
print(matches)
```

re.split(pattern,string)

Splits the string where there is a match and returns a list of strings where the splits have occurred.

```
import re
inputString = 'one 1 two 2 three 3 four thousand 4000'
#search decimal digits
sarchPattern = '\d+'    #try this>>> sarchPattern = '\d'
#or
#sarchPattern = '[0-9]+'

matches = re.split(sarchPattern, inputString)
print(matches)
```

re.split(pattern,string,maximum_no_of_splits)

Splits the string where there is a match and returns a list of strings where the splits have occurred.

```
import re
inputString = 'one 1 two 2 three 3 four thousand 4000'
#search decimal digits
sarchPattern = '\d+'    #try this>>> sarchPattern = '\d'
#or
```

```
#sarchPattern = '[0-9]+'

matches = re.split(sarchPattern, inputString,2)
print(matches)
```

re.sub(pattern, replace_with_sbstitute, string)
Substitute/Replace the pattern matches with some string

```
import re
inputString = 'one 1 two 2 three 3 four thousand 4000'
#search decimal digits
sarchPattern = '\d+'
substituteString='Number'
matches = re.sub(sarchPattern,substituteString, inputString)
print(matches)
```

Controlling the Substitute
re.sub(pattern, replace_with_sbstitute, string, maximum_match_and_Replace)

```
import re
inputString = 'one 1 two 2 three 3 four thousand 4000'
#search decimal digits
sarchPattern = '\d+'
substituteString='Number'
matches = re.sub(sarchPattern,substituteString, inputString,2)
print(matches)
```

re.subn(pattern, replace_with_sbstitute, string, maximum_match_and_Replace)
In addition to the results of re.sub() it also returns number of substitution made

```
import re
inputString = 'one 1 two 2 three 3 four thousand 4000'
#search decimal digits
sarchPattern = '\d+'
substituteString='X'
matches = re.subn(sarchPattern,substituteString, inputString)
#matches = re.subn(sarchPattern,substituteString, inputString,2)
print(matches)
```

re.search(pattern, str)

```
import re
inputString = 'one two three four four'
sarchPattern = 'four'  #search first instance of 'four'
#sarchPattern = '^four' #search 'four' at the begnining of the line
#sarchPattern = '\Afour
#sarchPattern = 'four$'  #search 'four' at the end of the line
#sarchPattern = 'four\Z'
match = re.search(sarchPattern,inputString)
print(match)
```

match.group()
returns the matched pattern as group
match.start(), match.end() and match.span()
returns the index of the match, starting position, end position, (start, end)
match.re, match.string
returns the regular expression and input string

```
import re
inputString = 'once there was a crow'
#starting with c and followed by one or more char and end by e
searchPattern = 'c.+e'
match = re.search(searchPattern, inputString)
#now printing the match object only
if match:
    print(match)
    print(match.group())
    print(match.start())
    print(match.end())
    print(match.span())
    print(match.re)
    print(match.string)
    print(re.compile(searchPattern))
else:
    print("No Match")
```

import myScript
How to Access:
myScript.fun1()

from myScript import fun1,fun2, fun3
from myScript import myClass,fun1,fun2, fun3
from myScript import *
How to Access:
fun1()

When two modules have identical names of their methods/calss
form myScript1 import fun1,fun2
form myScript2 import fun1,fun2

How to Access:
form myScript1 import fun1,fun2
fun1()
fun2()

form myScript2 import fun1,fun2
fun1()
fun2()

OR
import myScript1
import myScript2
myScript1.fun1()
myScript2.fun1()