1. **Where can we use Python?**

We can use Python in:

1. Back end web development

2. Data Analysis

3. AI and Scientific Computing

4. Building Games, Desktop Apps, Graphics

5. Creating GUIs

2. **What are features of Python?**

1. Python is easy to understand and Syntax and Programs are very clear in Python.

2. Python is portable language that means the code written in python can be run at numerous platform like windows, linux , unix etc.

3. Python contains less lines of code when compare to other programming languages so that it will reduce space and time.

4. Python supports Object Oriented Programming concepts.

3. **Why we need to use variables in Python?**

1. If we use Variables then it is very easy to understand the program.

2. When we want to use a large value or a name again and again in a program, then varaibles are very helpful to avoid wrong outputs.

3. Variables are also called as name assigned to the memory location. So Using variables we can easily find out the location of values in memory.

4. **What is format concept in Python?**

Format is one of the fundamental concept in python which is helpful to alter the format of Numbers and Strings.

**Syntax**: format(Value,Specifiers)

**EX**: >>> 8/9

0.888888888888

>>> format(8/9,".3f") # To get only 3 digits after decimal point in the output

'0.889'

>>> format("hello","<20") # **<** indicates left justification

'hello '

>>> format("hello",">20") # **>** indicates right justification

' hello'

>>> format("hello","^20") # **^** indicates centre justification

' hello '

5. **What is difference between Division and Truncating Division?**

1. **Division(/)** gives the output in floating point numbers

Ex: >>> 5/4

1.25

2. **Truncating Division(//)** truncates the values after decimal point and gives output as integer.

EX: >>> 5//4

1

6. **Mention the Precedence order of operators from highest to lowest?**

1. () --- Parenthesis 2. \*\* ---- Exponential

3. Unary Operator 4. \*, / , // , % (Same precedence)

5. + , - 6. <<,>> (Bitwise Shift operators)

7. & (Bitwise AND) 8. ^ (Bitwise XOR)

9. | (Bitwise OR) 10. Relational and membership Operators

11. not 12. and 13. Or

7. **What are Control Structures in Python?**

There are 3 types of controls in Python:

1. **Sequential Control**:

Instructions in program will follow the sequential order that means it will follow the order in which we wrote the program i.e., first line of the code will be executed first and followed by second and then third and then fourth and so on.

1. **Selection Control**: ( if statements, if – else statements, Nested if, elif header)

The statement will be executed based on the Condition.

**Ex** : If Condition is true then a set of statements will be executed and if Condition is false then another set of statements will be executed.

1. **Iterative Control**: ( for statement, while statement)

The statement(s) will be executed again and again in a loop based on the condition until condition fails.

**Ex**: While Condition is true then statement(s) will be executed until the condition fails and once the condition is failed then the statement(s) outside the loop will be executed.

8. **What is List ? What are Characteristics of List in Python?**

**List**: List is a group of different types of elements.

**Characteristics of List** :

1. **Mutable** – We can change or alter the list.
2. **Linear Data Structure** – Elements are arranged in linear order.
3. **Mixed Type Elements** – List contains integer values, floating point values and string values.
4. **Variable Length** – We can add or remove elements in the list.
5. **Zero Based Indexing** – First element in the list will start from base Zero.

9. **What are the Operations that we can perform with Lists in Python?**

1. **Replace**:

**Ex**:

>>> list1=[10,20,30,45.6,67.97]

>>> list1

[10, 20, 30, 45.6, 67.97]

>>> list1[2]="hello" # Replacing 2nd element with "hello"

>>> list1

[10, 20, 'hello', 45.6, 67.97]

2. **Insert**:

**Syntax** : <listname>.insert(index, Value)

**Ex**:

>>> list1.insert(4,'Welcome')

>>> list1

[10, 20, 'hello', 45.6, 'Welcome', 67.97]

1. **Sort**: Sort operation is performed only when list contains similar type of elements

**Ex 1**:

>>> animals = ['monkey','dog','cat','lion']

>>> animals.sort() # Sort String values in the list based on Alphabetical order

>>> animals

['cat', 'dog', 'lion', 'monkey']

**Ex 2**:

>>> num=[5,3,1.5,9,7.4]

>>> num.sort() # Sort numbers in ascending order

>>> num

[1.5, 3, 5, 7.4, 9]

1. **Delete**:

**Ex 1**:

>>> **del** num[3] # Delete 3rd element in the list 'num'

>>> num

[1.5, 3, 5, 9]

**Ex 2**:

>>> **del** num # Delete entire list 'num'

>>> num

Traceback (most recent call last):

File "<pyshell#15>", line 1, in <module>

num

NameError: name 'num' is not defined

1. **Append**: Adds element at the end of the list

**Ex**:

>>> animals

['cat', 'dog', 'lion', 'monkey']

>>> animals.append('donkey')

>>> animals

['cat', 'dog', 'lion', 'monkey', 'donkey']

1. **Reverse**: This operation is used to reverse the position of the elements in the list.

**Ex**:

>>> animals

['cat', 'dog', 'lion', 'monkey', 'donkey']

>>> animals.reverse()

>>> animals

['donkey', 'monkey', 'lion', 'dog', 'cat']

10. **What are Sequence Operations in Python?**

1. **Length**:

**len()** is used to find length of Sequences.

>>> name = "Python"

>>> list = [2,4,5.7,"hello"]

>>> tuple = (3,9,12,26.9)

>>> len(name)

6

>>> len(list)

4

>>> len(tuple)

4

2. **Select**:

To select Individual element from Sequences we can use select operation.

>>> name[2]

't'

>>> list[1]

4

>>> tuple[3]

26.9

3. **Slice**:

To get part of list or string or tuple we can use slice operation.

>>> name[2:5]

'tho'

>>> list[:4:2]

[2, 5.7]

>>> tuple[-1:]

(26.9,)

4. **Count**:

To find number of times an element is present in a sequence we can use **count()** operation.

>>> name.count('t')

1

>>> list.count(4)

1

5. **Index**:

To find index of an element in a sequence we can use **index()** operation.

>>> name.index('h')

3

>>> tuple.index(12)

2

1. **Membership**:

To find whether a element is present in the sequence we can use **in** and **not in** operators.

>>> 'h' in name

True

>>> 9 not in tuple

False

1. **Concatenation**:

To join 2 or more strings or lists or tuples we can use concatenation operator **(+).**

>>> name2 = " welcome"

>>> name + name2

'Python welcome'

>>> list2 = [345,7986,525.67]

>>> list + list2

[2, 4, 5.7, 'hello', 345, 7986, 525.67]

>>> list + tuple

TypeError: can only concatenate list (not "tuple") to list

1. **Minimum**:

>>> min(name2)

'c'

>>> min(tuple)

3

1. **Maximum**:

>>> max(name)

'y'

>>> max(list2)

7986

10**. Sum**:

>>> sum(name,name2)

TypeError: sum() can't sum strings [use ''.join(seq) instead]

>>> sum(list2)

8856.67

11. **What is definition of Functions?**

**Definition of Functions**:

Functions are nothing but number of statements which are grouped together to do some particular task.

Functions will reduce the Space of Memory and Time of work.

There are 2 parts in a function:

1. Function Definition
2. Function Call

**Ex**:

def average(n1,n2,n3): # Function Definition

return (n1+n2+n3)/3.0

print("Welcome!!! Functions")

result1 = average(10,20,30) # Function call

result2 = average(3.5,2.8,6.9) # Function call

print(result1)

print(result2)

**Output**:

Welcome!!! Functions

20.0

4.3999999999999995

**Formal Parameters**:

The parameters which are used in Function Definition are called Formal Parameters.

**Actual Parameters**:

The parameters which are used in Function Call are called Actual Parameters.

12. **Write an example Program to update Global Variable inside a function?**

We can update Global Variable inside a function by using **global** Keyword.

**Example**:

var = 10

def func1():

global var

var = var + 1

print("Variable value in func1 is : ", var)

# main

Func1()

**Output**:

Variable value in func1 is : 11

13. **What is Anonymous function in python?**

**Anonymous Function**:

A function without any name is known as Anonymous function i.e., here we don’t use any function name or we don’t use any keyword called def.

In this anonymous function we use a keyword called **lambda**. Hence anonymous function is also called **lambda function**.

**Example**:

sum = lambda var1,var2 : var1 + var2

print("sum is : ", sum(10,40))

**Output**:

Sum is : 50

14. **Write an example program with variable length arguments in a function?**

**Variable Length Arguments**:

These arguments are very useful when we don’t know how many parameters need to be mentioned while defining a function.

**Ex**:

def func1(\*mylist):

for num in mylist:

print(num)

return

# main

func1(10,20,30)

func1(12,56)

func1(9999)

**Output**:

10

20

30

12

56

9999

15. **What are Modules in Python?**

**Modules**: Modules are nothing but group of instruction or functions or statements. To use any module in a program we need to import module using **import** keyword.

**Ex**: Math Module

>>> import math

>>> math.factorial(5)

120

To know how many functions are defined in a particular module we can use **help(<module name>)**

>>> help(math)

16. **What are the different ways to create Dictionaries in Python?**

**Dictionaries**:

Dictionaries are un-ordered collection of items with Key and Value Pairs whereas Key can be float, int, string, tuple and Key is of Immutable (Non-changeable) type and value can be any type like float, int, string, list, tuple.

We can’t use list and dictionary as Key in any dictionary because list and dictionary are mutable.

**Different ways of Creating Dictionaries:**

1. **Using Curly Bracket Syntax**

>>> my\_dict = {1:"Apple",2:"Mango",3:"Orange"}

>>> my\_dict

{1: 'Apple', 2: 'Mango', 3: 'Orange'}

1. **Creating empty dictionary, fill in the entries one by one**

>>> my\_dict = {}

>>> my\_dict[1] = {"Banana"}

>>> my\_dict[2] = {"Bat"}

>>> my\_dict[3] = {"Bark"}

>>> my\_dict

{1: {'Banana'}, 2: {'Bat'}, 3: {'Bark'}}

1. **Dictionary constructor and a list of tuples**

>>> d = dict([(1,"mahesh"),(2,"arjun"),(3,"kalyan")])

>>> d

{1: 'mahesh', 2: 'arjun', 3: 'kalyan'}

1. **From two Parallel List**

>>> a = [1,2,3,4]

>>> b = ["Animals","Birds","Mammals","Human"]

>>> my\_dict = {}

>>> for i in range(len(a)):

my\_dict[a[i]] = b[i]

>>> my\_dict

{1: 'Animals', 2: 'Birds', 3: 'Mammals', 4: 'Human'}

17. **What is enumerate() in Python?**

The enumerate() method adds counter to an iterable and returns it. The returned object is an enumerate object.

We can convert enumerate objects to list and tuple using list() and tuple() methods respectively.

**Syntax**:

enumerate(iterable, start=0)

**Parameters**:

1. iterable : any object that supports iteration
2. start (optional) : the index value from which the counter is to be started, by default it is 0.

**Ex 1:**

>>> color = ['Red','Blue','Green','Yellow']

>>> for i,j in enumerate(color):

print(i,j)

**Output:**

1. Red
2. Blue
3. Green
4. Yellow

**Ex 2:**

l1 = ["eat","sleep","repeat"]

s1 = "king"

# creating enumerate objects

obj1 = enumerate(l1)

obj2 = enumerate(s1)

print("Return type:", type(obj1))

print(list(enumerate(l1))) # converting enumerate to list

print(tuple(enumerate(l1))) # converting enumerate to tuple

# changing start index to 2 from 0

print(list(enumerate(s1,2))) # converting enumerate to list

print(tuple(enumerate(s1,2))) # converting enumerate to tuple

**Output**:

Return type: <class 'enumerate'>

[(0, 'eat'), (1, 'sleep'), (2, 'repeat')]

((0, 'eat'), (1, 'sleep'), (2, 'repeat'))

[(2, 'k'), (3, 'i'), (4, 'n'), (5, 'g')]

((2, 'g'), (3, 'e'), (4, 'e'), (5, 'k'))

18. **What is set data type in Python?**

**Set**:

Set is a mutable data type with non-duplicated un-ordered values. It is an associated data structure.

**Creating a set** :

>>> fruits={"apple","banana"}

>>> fruits

{'banana', 'apple'}

**Adding an element to set:**

>>> fruits.**add**("grapes")

>>> fruits

{'grapes', 'banana', 'apple'}

**Removing an element from the set:**

>>> fruits.**remove**("banana")

>>> fruits

{'grapes', 'apple'}

**Creating set with Duplicate elements:**

>>> num = {1,2,4,2,1,3,2,5,3,4,5}

>>> num

{1, 2, 3, 4, 5} # Set contains only non-duplicated values

**Creating an Empty set**:

>>> emptyset = set()

>>> type(emptyset)

<class 'set'>

**Copying a set:**

To Copy a set of elements in to another set, we can use **set()** method.

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

>>> numbers = **set**(num)

{1, 2, 3, 4, 5}

To Copy a set of elements in to another set, we can use **copy()** method.

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

>>> numbers = num.**copy**()

{1, 2, 3, 4, 5}

We can also create **Immutable set** data type by using **frozenset** keyword.

**Ex**:

>>> animal = frozenset(["tiger","lion"])

>>> animal

frozenset({'lion', 'tiger'})

>>> animal.add("cat")

Traceback (most recent call last):

AttributeError: 'frozenset' object has no attribute 'add'

19. **Write an example program for Exception Handling in Python?**

**Code**:

**import** math

num=int(input("Enter a number to find factorial: "))

**try**:

result = math.factorial(num)

print(result)

**except ValueError**: # if we didn’t specify any Exception here then it will be applicable for all type of Exceptions

print("Can not find factorial of Negative numbers")

**Output 1**: input with Positive Number

Enter a number to find factorial: 6

720

**Output 2**: input with Negative Number

Enter a number to find factorial: -5

Can not find factorial of Negative numbers

20. **What is Object and Class?**

**Object**: It is collection of data and its functionality.

**Class**: It is blueprint for objects. In Class, we can create many objects.

**Ex**:

# Creating a empty class

class person:

pass

# main

p=person() # Creating a object called **p**

print(p)

**Output**:

<\_\_main\_\_.person object at 0x03D311B0> # Represents object is created.

21. **What is difference between a Method in Class and a Normal function in Python?**

There is a difference between Method and function in the parameter list.

In Methods, we need to **add an extra parameter** to the beginning of the parameter list.

In Methods, the first parameter of parameter list refers to the object itself and we use word **self** for this parameter which is highly recommended by the Programmers.

In Methods, Parameters after self are Optional.

**Syntax**:

**class** <classname>:

**def** <nameofmethod>(self,[parameters]):

**Ex**:

# Creating a class with Methods

**class** person:

**def** \_\_init\_\_(self,name): # **\_\_init\_\_(self,[parameters])** is used to initialize the object

self.name = name

name = "John "

print(name)

**def** display(self):

print("Hello!!!! ",self.name)

p1 = person("Python") # Creating a object called **p1**

p1.display() # Calling Method with object

person("Cameroon").display() # Calling method with calss

**Output:**

John

Hello!!!! Python

Hello!!!! Cameroon

**Use of self parameter in every method**:

**self** parameter differentiate between local variables and instance variables.

**Instance Variable**:

The variable which is belongs to an object is called instance variable.

In the above example, **self.name** is a instance variable because we know that self refers to the object.

**Local Variable**:

The variable which does not belongs to an object is called local variable .

In the above example, **name** is local variable.

22. **Write an Example Program for Class and Instance Variables?**

**Code**:

**class** Student:

clg = 'svce' # Class Variable

**def** \_\_init\_\_(self,rollno,name):

self.rollno = rollno # self.rollno is Instance variable and rollno is Local Variable

self.name = name # self.name is Instance variable and name is Local Variable

**def** display(self):

print("Student Name: ",self.name)

print("Student Roll No: ",self.rollno)

print("College: ",student.clg)

student1 = student('svce001', "Ram Charan")

student1.display()

student2 = student('svce002', "Komaram Bheem")

student2.display()

**Output**:

Student Name: Ram Charan

Student Roll no: svce001

College: svce

Student Name: Komaram bheem

Student Roll no: svce002

College: svce

23. **What if we don’t use self parameter in methods in class?**

**Code**:

**class** person:

**def** display(): # creating display method without self parameter

print("Hello!!!!")

**p=person()** # creating a object

p.display()

**Output**:

Traceback (most recent call last):

p.display()

TypeError: display() takes 0 positional arguments but 1 was given

By using **@Staticmethod** we can avoid **self** parameter in methods in class

**Ex**:

**class** person:

**@Staticmethod**

**def** display(): # creating display method without self parameter

print("Hello!!!!")

**p=person()** # creating a object

p.display()

**Output**:

Hello!!!!

24. **Write Syntax and Example Program for Inheritance in Python?**

**Syntax**:

**class** **baseclass**:

base\_class\_body

**class** **derivedclass**(baseclass):

derived\_class\_body

**Example**:

**class** **animal**:

**def** \_\_**init**\_\_(self,name):

self.name = name

**def** **eat**(self):

print("Eating")

**class** **dog**(animal):

**def** **bark**(self):

print("Barking")

**def** **display**(self):

print(self.name)

d = dog("Baby Dog") # creating a object

d.display() # Inherited from Base Class

d.eat() # Inherited from Base Class

d.bark()

**Output:**

Baby Dog

Eating

Barking

25. **Write an Example Program for Multiple Inheritance in Python?**

In **Multiple Inheritance**, there will be more than one Base class and one Derived Class.

**Syntax**:

Baseclass1 Baseclass2

\ /

Derivedclass

**Example**:

**class** **land\_animal**:

**def** **printing**(self):

print("This animal lives on land")

**class** **water\_animal**:

**def** **display**(self):

print("This is water animal also")

**class** **frog**(land\_animal,water\_animal):

**pass**

**f1 = frog()** # creating a object

f1.printing()

f1.display()

**Output**:

This animal lives on land

This is water animal also

26. **Write an Example Program for Multi-level Inheritance in Python?**

In **Multi-level Inheritance**, there will be one Base Class and more than one Derived Classes.

**Syntax**:

Baseclass 🡪 Derivedclass1 🡪 Derivedclass2

**Example**:

**class** **Person**:

**def** **display**(self):

print("This is class Person")

**class** **Employee**(Person):

**def** **printing**(self):

print("This is Derived class Employee")

**class** **Programmer**(Employee):

**def** **show**(self):

print("This is another derived class Programmer")

**p = Programmer()** # creating a object

p.display()

p.printing()

p.show()

**Output**:

This is Base class Person

This is Derived class Employee

This is another derived class Programmer

27. **What is Method Overriding and Write an Example program for Method Overriding in Python?**

**Method Overriding**:

We can change the implementation of a method in derived class which is provided by its base class.

To override a method, We have to define that method in derived class with same name.

**Example**:

**class** **A**:

**def** **display**(self):

print("Method belongs to base class A")

**class** **B**(A):

**def** **display**(self): # Overridden Method

print("Method belongs to derived class B")

**Output**:

Method belongs to derived class B

28. **What is Encapsulation and Write an Example program for Encapsulation in Python?**

**Encapsulation**:

In Object Oriented Programming, We can restrict the access to the variables and methods. This is called as Encapsulation.

We use Encapsulation to prevent the data from being modified accidently.

**Example with Private Method**:

# **Double Underscore (\_\_)** before a method indicates that method is Private.

# We can not access/call private method outside the class

**class** **car**:

**def** **\_\_init\_\_**(self):

self.\_\_updatesoftware() # calling Private Method

**def** **drive**(self):

print("Driving")

**def** **\_\_updatesoftware**(self): # Defining private method

print("Updating")

blackcar = car()

blackcar.drive()

**# blackcar.\_\_updatesoftware()** # Calling private method outside the class **throws Error**

**Output**:

Updating

Driving

**Example with Private Variable**:

# Private Variables can be modified only inside the class methods.

# We can not modify private variables outside the class.

**class** **car**:

\_\_maxspeed = 0 # private variable

\_\_name = " " # private variable

**def** **\_\_init\_\_**(self):

self.\_\_maxspeed = 200 # Modifying Private Variable inside the class

self.\_\_name = "Super Car" # Modifying Private Variable inside the class

**def** **drive**(self):

print("Driving")

print(self.\_\_maxspeed)

**def** **speed**(self,speed): # Defining private method

self.\_\_maxspeed = speed # Modifying Private Variable inside the class

print(self.\_\_maxspeed)

redcar = car()

redcar.drive()

redcar.speed(100)

# **redcar.\_\_maxspeed = 100** # Modifying Private Variable outside the class **throws Error**

**Output**:

Driving

200

100

29. **What is Polymorphism and Write an Example program for Polymorphism in Python?**

**Polymorphism**: Poly means "many" and morph means "forms"

Polymorphism is the ability of an object to adopt the code to the type of data processing.

Polymorphism helps us to define an action regardless of type of object.

Polymorphism is nothing but **a method which behaves differently for different objects.**

Polymorphism is a built-in feature in Python.

**Example program for Polymorphism**:

**class** **dog**:

**def** **sound**(self):

print("Bow Bow")

**class** **cat**:

**def** **sound**(self):

print("Meow")

**class** **makesound**(animaltype):

animaltype.sound()

catobj = cat()

dogobj = dog()

makesound(catobj)

makesound(dogobj)

**Output**:

Meow

Bow Bow

30. **What is time module? What are the methods in time module?**

**Time Module**:

Using the methods which are belongs to Time module, we can get current date, time, day, year, month.

To use those methods, we need to **import** **time** module.

**Methods to get current time**:

>>> import time

>>> time.time() # To get Current time in seconds we use **time()** method

1575561356.1108036

>>> time.localtime(time.time()) # To get Current time in appropriate format we use **localtime(time.time())** method

time.struct\_time(tm\_year=2019, tm\_mon=12, tm\_mday=5, tm\_hour=21, tm\_min=28, tm\_sec=40, tm\_wday=3, tm\_yday=339, tm\_isdst=0)

>>> time.asctime() # To get time in another format

'Thu Dec 5 21:32:54 2019'

**Methods to get information about Upcoming days or Previous days**: For this use tuples.

>>> tuple1 = (1997,12,25,7,30,59,0,0,0)

>>> time.mktime(tuple1)

883015259.0

>>> time.localtime(time.mktime(tuple1))

time.struct\_time(tm\_year=1997, tm\_mon=12, tm\_mday=25, tm\_hour=7, tm\_min=30, tm\_sec=59, tm\_wday=3, tm\_yday=359, tm\_isdst=0)

**Sleep() Method in time module**:

Using **sleep()** method, we can **delay the execution** of the instruction.

**Ex**:

import time

time.sleep(10) # Output will be displayed after 10 seconds

print("Hello")

**Output:**

Hello # Output will be displayed after 10 seconds because of sleep(10) method before print statement

31. **What is calendar module? What are the methods in calendar module?**

**Calendar Module**:

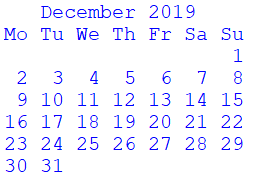
By using Calendar Module Methods, we can print the calendar of a month or a whole year.

To use those methods, we need to **import** **calendar** module.

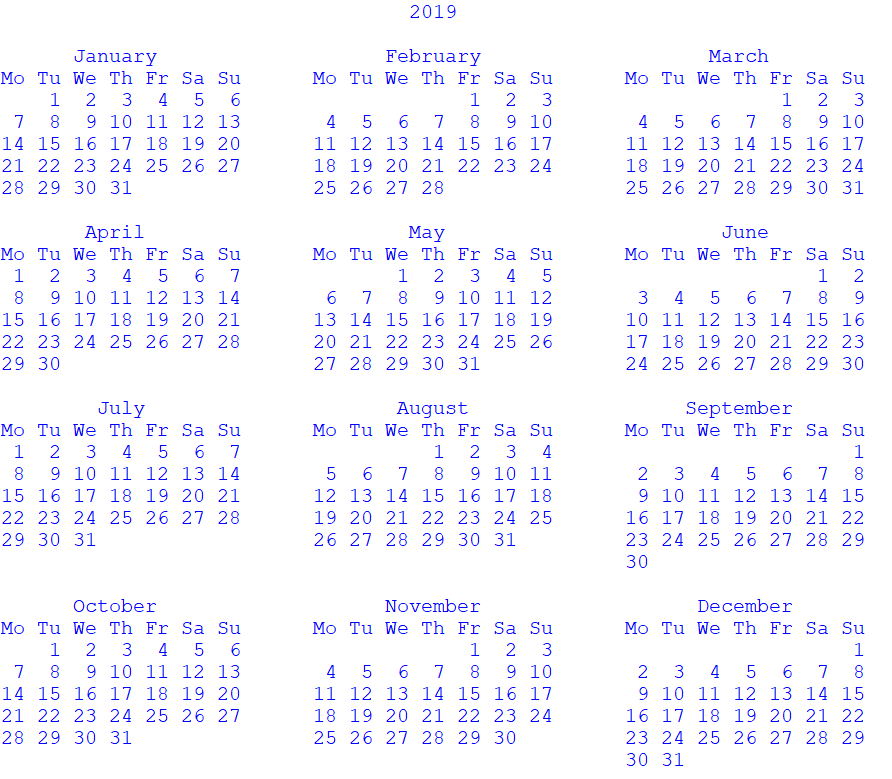
**Method to print the calendar of a month**:

>>> import calendar

>>> print(calendar.month(2019,12)) # To print calendar of a month we use **month(year,month,[width,length])** method of calendar module.



>>> print(calendar.calendar(2019)) # To print calendar of a whole year we use **calendar(year,[width,length,distance b/w two months,no.of months per column])** method of calendar module.



**calendar.weekday(year, month, date)** method:

Index of Monday = 0, Tuesday = 1, Wednesday = 2, Thursday = 3, Friday = 4, Saturday = 5, Sunday = 6

>>> import calendar

>>> calendar.weekday(2019,12,5)

3

**calendar.isleap(year)** Method:

By using **isleap(year)**, we can find out whether the year is Leap Year or Not.

>>> calendar.isleap(2000)

True

>>> calendar.isleap(2019)

False

**calendar.leapdays(year1,year2)** Method:

By using **leapdays(year1,year2)**, We can find out number of leap years are present between year1 and year2.

>>> calendar.leapdays(2000,2019)

5

**To get more information about calendar module, use help(calendar) method.**

32. **What is type function? What is dir function?**

**type** function:

**type** is a built-in function and it is used to find the data type of the object.

Ex:

>>> type(1)

<class 'int'>

>>> type('python')

<class 'str'>

>>> list1 = []

>>> type(list1)

<class 'list'>

**dir** function:

**dir**  is a built-in function which gives the name present in the current scope.

Ex:

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_']

>>> list1 = []

>>> A = 10

>>> dir()

[**'A'**, '\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', **'list1'**]

We can find out the directory of specific object also by using **dir(object)**

>>> import calendar

>>> dir(calendar)

33. **Python Iterators and Iterables:**

An **Iterator** is an Object that contains a Countable Number of Values and iterator can be iterated upon all the Values.

In Python, an iterator is an object which implements the iterator Protocol, which consist of the methods **\_\_iter()\_\_** and **\_\_next()\_\_** .

**Strings, Lists, Tuples, Dictionaries and sets** are all **Iterable** **Objects**. They are **Iterable Containers** which we can get an iterator from.

Example:

mystr = "HEAD"

myit = iter(mystr)

print(next(myit))

print(next(myit))

print(next(myit))

print(next(myit))

**OUTPUT:**

H

E

A

D

**Looping through an Iterator**:

The **for** loop actually creates an iterator object and executes the next method for each loop.

Example:

mytuple = ("rat","dog","kitten")

for x in mytuple:

print(x)

OUTPUT:

rat

dog

kitten

**How for loop actually works?**

**for element in iterable:**

is actually implemented as:

# create an iterator object from that iterable

**iter\_obj = iter(iterable)**

# infinite loop

**while true:**

**try**:

# get next item

**element = next(iter\_obj)**

**except stopIteration**:

# if stopIteration is raised, break from loop

**Break**

So internally, **for** loop creates an iterator object, **iter\_obj** by calling **iter()** on the iterable.

This **for** loop is actually an **infinite while loop**.

Inside the loop, it calls **next()** to get the next element and executes the body of **for** loop with this value. After all the items exhaust, **stopIteration** is raised which is internally caught and the loop ends. Note that any other kind of exception will pass through.

**Create an Iterator**:

To Create an object/class as an iterator we have to implement the methods **\_\_iter()\_\_** and **\_\_next()\_\_** to the objects.

The **\_\_iter()\_\_** method allows to do operations but must always **return the iterator object** itself.

The **\_\_next()\_\_** method allows to do operations but must always **return the next item** in the sequence.

Example 1:

Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

class MyNumbers:  
  def \_\_iter\_\_(self):  
    self.a = 1  
    return self  
  
  def \_\_next\_\_(self):  
    x = self.a  
    self.a += 1  
    return x  
  
myclass = MyNumbers()  
myiter = iter(myclass)  
  
print(next(myiter))  
print(next(myiter))  
print(next(myiter))  
print(next(myiter))  
print(next(myiter))

OUTPUT:

1

2

3

4

5

Example 2:

Create an Iterator that returns Top Ten numbers starting form 1

class TopTen:

def \_\_init\_\_(self):  
    self.num = 1

  def \_\_iter\_\_(self):  
       return self  
  
  def \_\_next\_\_(self):

if  self.num <= 10:

val = self.num

self.num += 1  
        return val  
 else:

raise stopIteration

values = TopTen()

print(next(values)) # **Prints 1 and doesn’t print 1 again in next for loop i.e., for loop will print values from 2**

for i in values:  
 print(i)

OUTPUT:

1

2

3

4

5

6

7

8

9

10

34. **Python Generators:**

**Generators are used to fetch one value at a time from more number of values defined in a function using yield keyword.**

1. **Generator Function**:

A Generator Function is defined like a Normal Function, but whenever it needs to generate a value, it does so with **yield** Keyword rather than **return**.

If the body of a **def function contains yield**, the function automatically becomes a **generator function**.

Generator function generates Iterator.

**Example**:

# A generator function that yields 1 for first time, 2 second time and 3 third time

**def** simpleGeneratorFun():

yield 1

yield 2

yield 3

# Driver code to check above generator function

for value in simpleGeneratorFun():

print(value)

OUTPUT:

1

2

3

2. **Generator Object**:

Generator functions return a generator object. Generator objects are used either by calling the next method on the generator object or using the generator object in **for** loop.

# A Python program to demonstrate use of generator object with next()

# A generator function

**def** simpleGeneratorFun():

yield 1

yield 2

yield 3

# x is a generator object

x = simpleGeneratorFun()

# Iterating over the generator object using next

print(x.\_\_next\_\_()); # In Python 2, next()

print(x.\_\_next\_\_());

print(x.\_\_next\_\_());

OUTPUT:

1

2

3

35. **Python Recursive Function**:

If a function calls itself, it is called as **Recursive Function**.

Example:

A Program using Recursive Function to find the Factorial of a Number

def factorial(num):

if num == 1:

return 1

else:

return num \* **factorial(num -1)**  # Here function is calling itself which is called as **Recursive function**

num = 4

print( factorial(num))

OUTPUT:

24

Advantages of Recursive Function:

1. Recursive Functions make the code look clean and simple.

2. A complex task can be broken down into simpler sub-problems using Recursion.

Dis-Advantages of Recursive Function:

1. Some times logic behind recursion is hard to follow through.

2. Recursive calls are more expensive as they take up lot of memory and time.

3. Recursive Functions are hard to debug.

**36. Dunder or Magic Methods in Python:**

1. The Variables, Methods or Functions that have Double Underscores on each side which represent that Variables, Methods or Functions are defined by Python.

2. Dunder means Double Under (Underscores). These are commonly used for Operator Overloading.

Example Methods: \_\_init\_\_, \_\_add\_\_, \_\_len\_\_, \_\_repr\_\_ etc.

The **\_\_init\_\_** method for initialization is invoked without any call, when an instance of a class is created.

3. These methods are the reason we can add two strings with ‘+’ operator without any explicit typecasting.

Example:

# declare our own string class

class String:

# magic method to initiate object

def **\_\_init\_\_**(self, string):

self.string = string

# print our string object

def **\_\_repr\_\_**(self):

return 'Object: **{}**'.format(self.string)

# Driver Code

if **\_\_name\_\_** == '**\_\_main\_\_**':

# object creation

string1 = String('Hello')

# print object location

print(string1)

OUTPUT:

Object: Hello

**37. List Unpacking:**

If we write variables on the left side separated by commas(,) , elements of a tuple and a list on the right side will be assigned to each variable.

Example:

a, b, c, \*other, d = [1,2,3,4,5,6,7,8,9]  
print(a)  
print(b)  
print(c)  
print(other)

print(d)

OUTPUT**:**

1

2

3

[4, 5, 6, 7, 8]

9

**38. PEP:**

**Python Enhancement Proposals** (**PEP**) are suggestions for improvements to the language, made by experienced Python Developers.

**PEP 8** is a style guide to write readable python code. It contains a number of guidelines in reference to variable names, which are summarized here:

* Modules should have short, all lowercase names.
* Class Names should be in the CapWords style.
* Most variables and function names should be lowercase\_with\_underscores.
* Constants should be CAPS\_WITH\_UNDERSCORES.
* Names that would clash with Python keywords (such as ‘class’ or ‘if’) should have a trailing underscore.
* Use spaces around operators and after commas to increase readability.
* Whitespaces should not be over used.
* For instance, avoid having any space directly inside any type of brackets.
* Lines should not be longer than 80 characters.
* ‘from module import \*’ should be avoided
* There should only be one statement per line.

**39. What is MRO in Python?**

**Method Resolution Order (MRO)** is the order in which Python looks for a method in a hierarchy of classes.

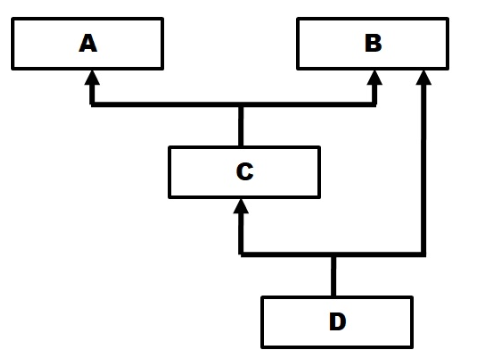
MRO plays vital role in the context of multiple inheritance as single method may be found in multiple super classes.

MRO follows **DEPTH FIRST SERACH** hierarchy and **searches from left to Right** **when two classes are at same level** while looking for a method in hierarchy of classes.

It is possible to see MRO of a class using **mro()** method of the class.

Example:

we create D from C and B. Classes C and B have process() method and as expected MRO chooses method from C. **Remember it goes from left to right. So it searches C first and all its super classes of C and then B and all its super classes**. We can observe that in MRO of the output given below.



**class** A:  
 **def** process(self):  
 print(**'A process()'**)  
   
**class** B:  
 **def** process(self):  
 print(**'B process()'**)  
  
**class** C(A, B):  
 **def** process(self):  
 print(**'C process()'**)  
  
**class** D(C,B):  
 **pass**obj = D()  
obj.process()  
  
print(D.mro())

OUTPUT:

C process()

[<class '\_\_main\_\_.**D**'>, <class '\_\_main\_\_.**C**'>, <class '\_\_main\_\_.**A**'>, <class '\_\_main\_\_.**B**'>, <class 'object'>]

Note: However, **if we remove process() method from class C** then process() method in **class A will be called** as it is the next class to be searched according to MRO.

The ambiguity that arises from multiple inheritance is handled by Python using MRO.

**40. What is Zen of Python?**

* It is important to write clean code that is easily understood, even weeks after we have written it.
* One way of doing this is to follow Zen of Python, a set of principles that serves as a guide to programming the clean code.
* **Zen of Python**: Listing of Python design principles and philosophies that are helpful in understanding and using the language effectively.
* Use the following code to access Zen of Python

**import** this

OUTPUT:

The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than \*right\* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

**Note**: There are 20 principles in the Zen of Python, but only 19 lines of text.The 20th principle is a matter of opinion, but our interpretation is that the blank line means “Use whitespace”.

**41. What is Functional Programming?**

**Functional Programming** is all about separation of concerns ie., **separation of data and functions** which object-oriented programming does as well. It’s all about packaging our code into separate chunks so that everything’s well organized in each part of our code and each part is organized in a way that make sense based on functionality.

Functions operate on well-defined data structures like lists and dictionaries. But the goal of a Functional Programming paradigm is same as object oriented paradigm.

Idea of Functional Programming:

1. To make our code clean and Understandable

2. Easy to extend our code

3. Easy to maintain

4. Its keep our code DRY so that we are not repeating ourself

5. Keep our code memory efficient because we are not storing information all over the place.

**Note**: The idea of Functional Programming is that there is a separation between data of a program and behaviour of a program.

**42. What is Pure Function?**

A function is called **Pure Function** if it **always returns the same result** for same argument values and it **has no side effects** like modifying an argument (or global variables) or outputting something.

The only result of pure function is the return value to keep our code clean and avoid bugs in our code.

Examples of **Pure Functions**: strlen(), pow(), sqrt() etc.,

Examples of **Impure Functions**: printf(), rand(),time() etc.,

**Note**: It’s impossible to have Pure Functions everywhere.

Example of Pure Function:

**def** multiple\_by2(li):  
 new\_list = []  
 **for** item **in** li:  
 new\_list.append(item\*2)  
 **return** new\_list  
  
print(multiple\_by2([1,2,3]))

OUTPUT:

[2, 4, 6]

**Note 1**: if **new\_list = [] is outside the function** then above Example is **not Pure function** because then above Example program will **have a side effect** that our function interacts with variable which is outside world of function in the program.

**Note 2**: if **print is inside a function in return statement** then also above Example is **not Pure function** because then above Example program will **have a side effect** that Outputting something on the output screen which is outside of function in the program.

**43. What are Map, Filter, Zip and Reduce Methods in Python?**

**Map Method:**

The **map()** function is a built-in function that calls the specified function for each item of an iterable (such as string, list, tuple or dictionary) and returns a list of results.

**Syntax**: map(function\_to\_apply, list\_of\_inputs)

Example 1:

my\_list = [1,2,3]  
# Pure function   
**def** multiple\_by2(item):  
 **return** item\*2  
  
print(list(**map**(multiple\_by2,my\_list)))  
print(my\_list)

OUTPUT:

[2, 4, 6]

[1, 2, 3]

Example 2:

print(list(map(**lambda** x: x\*\*2,(1,2,3,4,5))))

OUTPUT:

[1, 4, 9, 16, 25]

**Filter Method:**

The **filter()** method filters the given iterable with the help of a function that tests each element in the iterable to be true or not.

**Syntax**: filter(function, iterable)

Example 1:

**def** only\_odd(item):  
 **return** item % 2 != 0  
  
print(list(filter(only\_odd,[1,2,3,4,5])))

OUTPUT:

[1, 3, 5]

Example 2:

number\_list = range(-5, 5)  
less\_than\_zero = list(**filter**(lambdax: x < 0, number\_list))  
print(less\_than\_zero)

OUTPUT:

[-5, -4, -3, -2, -1]

**Zip Function:**

The zip() function takes iterables (can be zero or more), aggregates them in a tuple and return it.

**Syntax**: zip(\*iterables)

Example:

number\_list = [1, 2, 3]  
str\_list = [**'one'**, **'two'**, **'three'**]  
*# No iterables are passed*print(list(**zip**())) # returns an empty iterator  
*# Two iterables are passed*print(set(**zip**(number\_list, str\_list)))

OUTPUT:

[]

{(1, 'one'), (2, 'two'), (3, 'three')}

**Reduce Method:**

The **reduce()** function accepts a function and a sequence and returns a single value calculated as follows:

1. Initially, the function is called with the first two items from the sequence and the result is returned.
2. The function is then called again with result obtained in step 1 and next value in the sequence. This process keeps repeating until there are items in the sequence.

**Syntax**: reduce (function, sequence [,initial])

When the **initial** value is provided, the function is called with the initial value and the first item from the sequence and initial is 0 by default if it is not provided.

In Python 2, reduce() was a built-in function whereas in Python 3, it is moved to **functools** module.

Example:

**from** functools **import** reduce  
  
**def** accumulator(acc,item):  
 print(acc, item)  
 **return** acc + item  
  
print(**reduce**(accumulator,[1,2,3,4],10))

OUTPUT:

10 1

11 2

13 3

16 4

20

**44. How do you debug a Python Program?**

By Using below command we can a debug a Python Program

**$ Python -m pdb python-script.py**

**45. What is <yield> keyword in python?**

The <**yield**> keyword can turn any function into a generator. Yield works like a standard return keyword.

But it’ll always return a generator object. Also, a function can have multiple calls to the <**yield**> keyword.

**Example Program**:

def testgen(index):

weekdays = ['sun','mon','tue','wed','thu','fri','sat']

yield weekdays[index]

yield weekdays[index+1]

day = testgen(0)

print(next(day), next(day))

**Output**:

sun mon

**46. How to convert a list into a string?**

To convert a list into a string, we can use <**' '.join()**> method which joins all elements into one and returns as a string.

**Example Program**:

weekdays = ['sun','mon','tue','wed','thu','fri','sat']

listAsString = ' '.join(weekdays)

print(listAsString)

**Output**:

sun mon tue wed thu fri sat

**47. How to convert a list into a tuple?**

To convert a list into a tuple, we can use python **tuple()** function. But, **we can’t change** the list after turning it into **tuple**, because it becomes **immutable**.

**Example Program**:

weekdays = ['sun','mon','tue','wed','thu','fri','sat']

listAsTuple = **tuple**(weekdays)

print(listAsTuple)

**Output**:

('sun', 'mon', 'tue', 'wed', 'thu', 'fri', 'sat')

**48. How to convert a list into a set?**

To convert a list into a tuple, we can use python **set()** function.

**Example Program**:

weekdays = ['sun','mon','tue','wed','thu','fri','sat']

listAsSet = **set**(weekdays)

print(listAsSet)

**Output:**

{'sun', 'sat', 'fri', 'tue', 'mon', 'thu', 'wed'}

**49. How to count the occurrences of a particular element in the list?**

In Python List, we can find the occurrences of an individual element using <**count()**> function.

**Example Program-1**:

weekdays = ['sun','mon','tue','wed','thu','fri','sun','mon','mon']

print(weekdays.**count**('mon'))

**Output**:

3

**Example Program-2**:

weekdays = ['sun','mon','tue','wed','thu','fri','sun','mon','mon']

print([[x,weekdays.**count**(x)] for x in set(weekdays)])

**Output**:

[['sun', 2], ['thu', 1], ['wed', 1], ['mon', 3], ['fri', 1], ['tue', 1]]

**50. What is NumPy Array?**

NumPy Arrays are more flexible then lists in Python. By using NumPy Arrays reading and writing items is faster and more efficient.

We can create Empty NumPy Array in 2 ways in Python.

1. import numpy

numpy.array([])

1. numpy.empty(shape=(0,0))

**51. What is Negative Index in Python?**

Python has a special feature like a negative index in Arrays and Lists. Positive index reads the elements from the starting of an array or list but in the negative index, Python reads elements from the end of an array or list.

**Example Program-1**:

import array

a = [1, 2, 3]

print(a[-3])

print(a[-2])

print(a[-1])

**Output**:

1

2

3

**Example Program-2**:

names = ['Chris', 'Jack', 'John', 'Daman']

print(names[-1][-1])

**Output**:

n

**52. What is enumerate() function in Python?**

The Python **enumerate() function adds counter to an iterable object**. Enumerate() function accepts sequential indexes starting from zero.

**Example Program**:

subjects = ('Python', 'Interview', 'Questions')

for i, subject in **enumerate**(subjects):

print(i, subject)

**Output**:

0 Python

1 Interview

2 Questions

**53. How to generate random numbers in Python?**

We can generate random numbers using different functions in Python. They are

1. **random()** – This command returns a floating point number, between 0 and 1.
2. **uniform(X, Y)** – It returns a floating point number between the values given as X and Y
3. **randint(X, Y)** – This command returns a random integer between the values given as X and Y

**54. How to print sum of numbers starting from 1 to 100?**

We can print sum of the numbers starting from 1 to 100 using below code:

**print(sum(range(1,101)))**

# In Python the range function does not include the end given, here it will exclude 101.

# Sum function print sum of the elements of range function, i.e 1 to 100.

**55. How do you set a global variable inside a function?**

We can use a global variable in other functions by declaring it as **global** in each function that assigns to it.

**Example Program:**

var = 0

def set\_var\_to\_one():

global var # Needed to modify global copy of var

var = 1

def print\_var():

print(var) # No need for global declaration to read value of var

set\_var\_to\_one()

print\_var() # Prints 1

**56. Write a Program to find the average of numbers in a list in Python?**