Q1. What is the purpose of Python's OOP?

Ans: OOP is a programming model that uses objects and classes. It aims to implement real world entities like inheritance, polymorphism, encapsulation in the programming. The main concepts of the OOP are as follows:

I. **Class:** A class is collection of objects. It is a logical entity that has attributes and methods. A class contains the blueprints or prototype from which the objects being created. There are two types of attributes.

Attributes are always public and can be accessed using the dot (.) operator. `E.g.: Myclass.Myattribute`

Class Attributes: Defined outside method. Accessed by class and instance objects. They belong to the class.

```
Class attr:
      x="class attribute"
      print("Output:")
obj=attr()
      print(obj.x)
```

Here, variable x is a class attribute. Created an inject obj of class attr and by the instance of class print the value of x.

Instance Attributes: Defined inside **__init__ method.** Can be accessed by an instance of the class.

```
class attr:
y="class attribute"
def init (self,y):
     self.y=y
print("Output")
obj=attr()
print(obj.y)
```

Here, created a variable y inside **__init__** method. y is an instance attribute. Create an obj object f class and print the value of y.

Consider an example, let's say you wanted to track the number of dogs that may have different attributes like breed, age. If a list is used, the first element could be the dog's breed while the second element could represent its age. Let's suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it's the exact need for classes.

Class Definition Syntax:

```
class ClassName:
    # Statement-1
    .
    # Statement-N
```

Python Syntax:

```
class A:
    pass
```

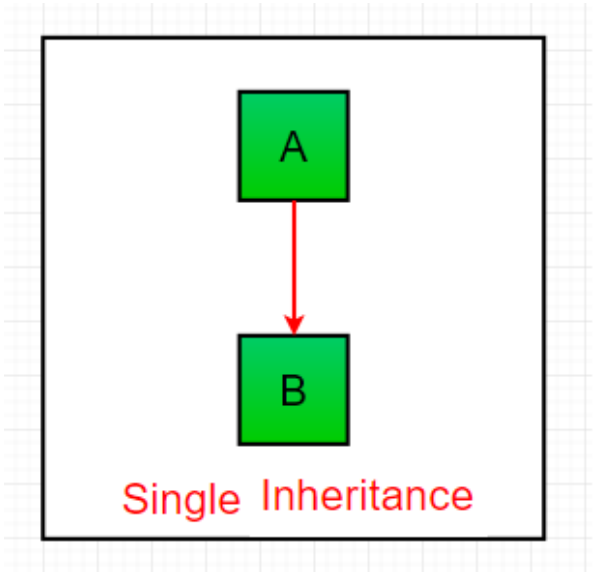
II. **Objects:** Object is an entity that has a state and behavior associated with it. It may be a real-world entity like table, mouse, chair etc. Strings, Integers, floating point numbers even arrays and dictionaries all are objects.

To understand the state, behavior, and identity let us take the example of the class dog (explained above).

- The identity can be considered as the name of the dog.
- State or Attributes can be considered as the breed, age, or color of the dog.
- The behavior can be considered as to whether the dog is eating or sleeping.

III. **Inheritance:** The inheritance is a capability of a class to inherit the properties of another class. The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class. Different types of inheritance are as follows:

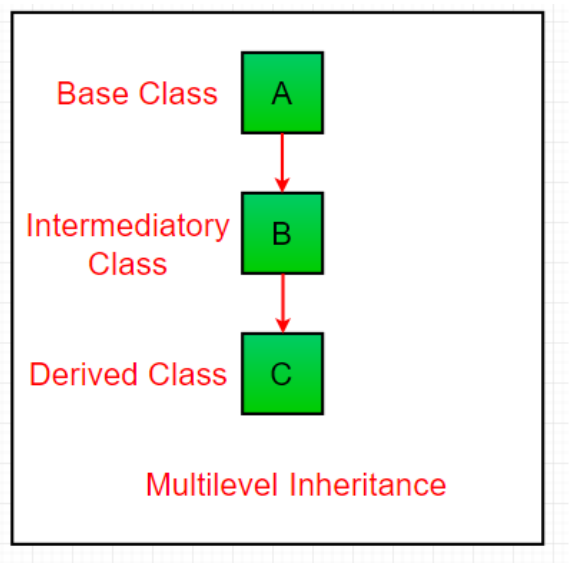
Single Inheritance: Single inheritance allows a derived class to derive the properties from single parent class. The structure of the Single Inheritance is as follows:




```
# The base class
class Animal:
    def bird(self):#Function or Method
        print("Birds are chirping!")

# Now, we"ll derived the Animal class which is base class
class dog(Animal): # Child class
    def mydog(self): #Function
        print("Dog is barking!")

a=dog()
a.bird()
a.mydog()
```

```
Birds are chirping!
Dog is barking!
```

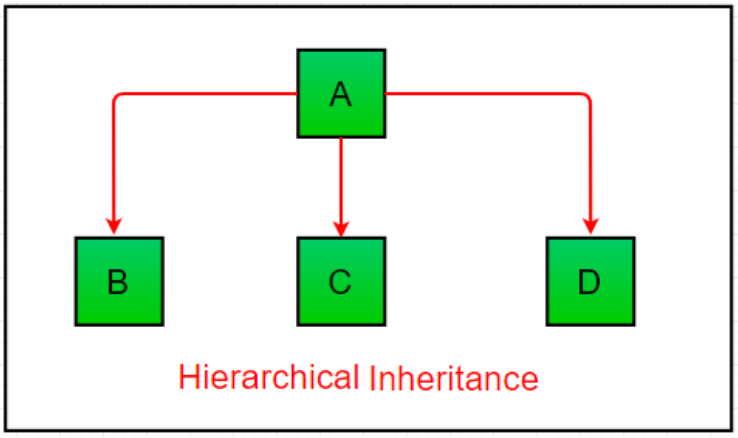
Multilevel: Multilevel inheritance allows to derive properties from a class which also a deriver class from another class. The structure of the Multilevel Inheritance is as follows:




```
# The base class
class Animal:
    def bird(self):#Function or Method
        print("Birds are chirping!")

# Now, we"ll derived the Animal class which is base class
class dog(Animal): # Child class of Animal class
    def mydog(self): #Function
        print("Dog is barking!")

# Now we'll derived the Dog class which is the child of Animal
class eat(dog): #Child class of Dog class
    def food(self):
        print("Birds eat seeds while dog eats dog food!")
a=eat()
a.bird()
a.mydog()
a.food()
```

```
Birds are chirping!
Dog is barking!
Birds eat seeds while dog eats dog food!
```

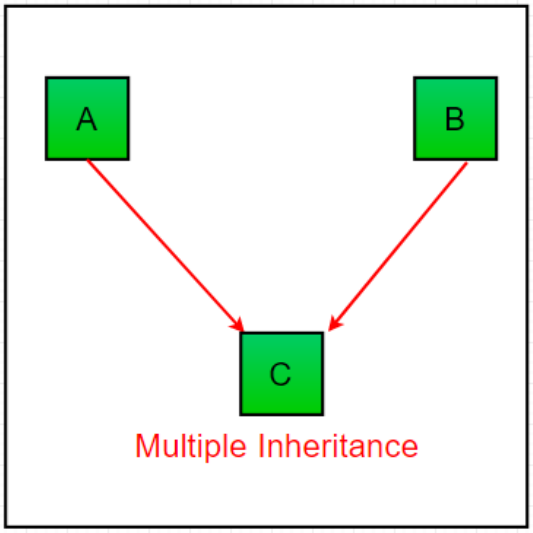
Hierarchical: Hierarchical inheritance allows more than one derived classis to derive the properties from one parent class. The structure of the Hierarchical Inheritance is as follows:




```
# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")

# Derived class1
class Child1(Parent):
    def func2(self):
        print("This function is in child 1.")

# Derivied class2
class Child2(Parent):
    def func3(self):
        print("This function is in child 2.")

object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

```
This function is in parent class.
This function is in child 1.
This function is in parent class.
This function is in child 2.
```

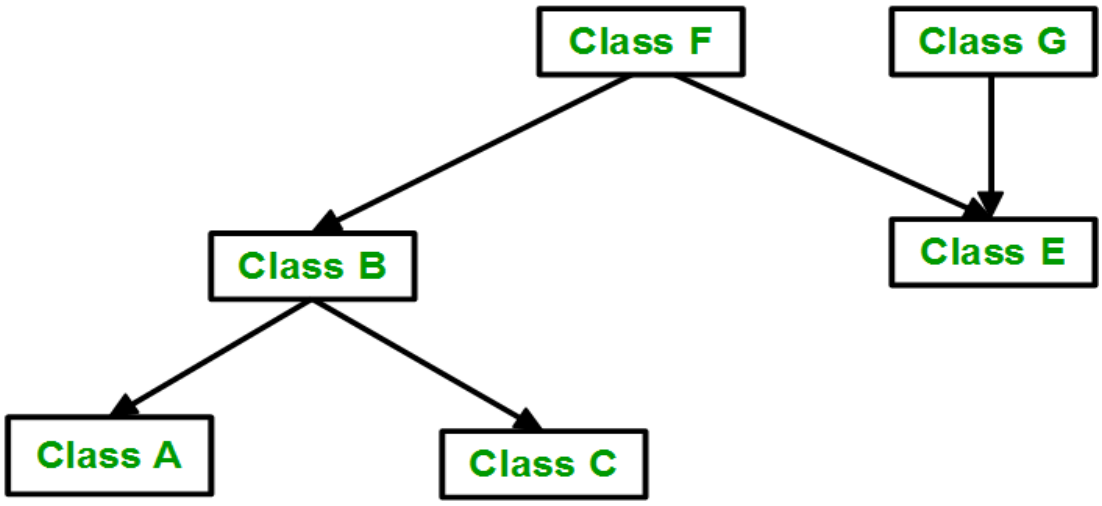
Multiple: Multiple inheritance allows the derived class to derive properties from more than one parent or base classes. The structure of the Multiple Inheritance is as follows:




```
# Base class 1
class Grand_Parent:
    def func1(self):
        print("This is the function of base class 1!")

# Base class 2
class Parent:
    def func2(self):
        print("This is the function of base class 2!")

# Derivied class or Child class
class Child(Grand_Parent, Parent):
    def func3(self):
        print("This is the function of child class")

object1 = Child()
object1.func1()
object1.func2()
object1.func3()
```

```
This is the function of base class 1!
This is the function of base class 2!
This is the function of child class
```

Hybrid Inheritance: The inheritance that have multiple types of inheritance is called Hybrid Inheritance. The structure of Hybrid Inheritance is as follows:




```
# Base class 1
class Grand_Parent:
    def func1(self):
        print("This is the function of base class 1!")

# Base class 2
class Parent:
    def func2(self):
        print("This is the function of base class 2!")

# Derivied class or Child class
class Child(Grand_Parent, Parent):
    def func3(self):
        print("This is the function of child class!")

# Derived class of base class 1
class kids (Grand_Parent):
    def func4(self):
        print("This is the function of kids class!")


object=Child()
object1=kids()
object.func1()
object1.func1()
object.func2()
object.func3()
object1.func4()
```

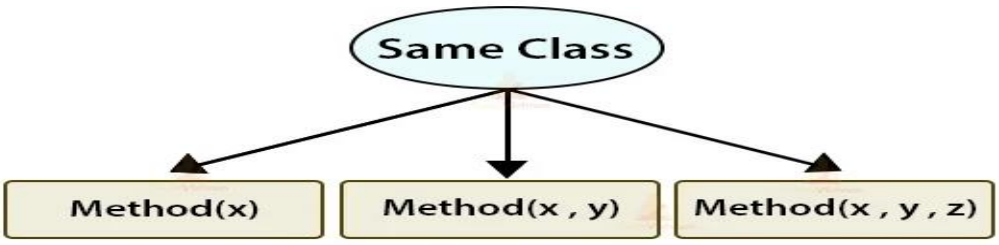
```
This is the function of base class 1!
This is the function of base class 1!
This is the function of base class 2!
This is the function of child class!
This is the function of kids class!
```

IV. **Polymorphism:** The polymorphism is nothing but just a concept which means ability to take various forms. For example we can use the len() function to know the length of different objects such as list, dictionary, tuple etc.

```
a="Greeetings"
print(len(a))

b=[1,2,3,4,5,6]
print(len(b))

c={1:'QAMAR', 2:'AMIN'}
print(len(c))

# SIMILARLY, We can use print() in mupltiple ways
print(5+5) # Here we are taking 5 as integers
print("5" + "5") # Here we are taking the 5 as strings
```

```
10
6
2
10
55
```

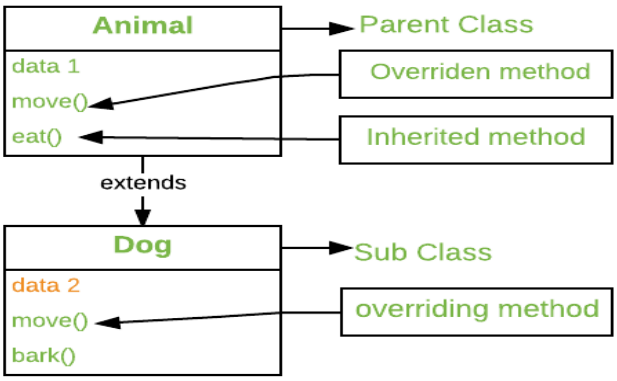
Method Overloading: It is an example of compile time polymorphism. In this, more than one method of the same class shares the same method name having different signatures (Forms). Python does not support the Method Overloading. We may overload the method but can only use the latest defined method. The structure of method overloading is as follows:




```
class Compute: # class
    def area(self, x = None, y = None): # area method
        if x != None and y != None:

            return x * y

        elif x != None:
            return x * x
        else:
            return 0

obj = Compute() # obj is Class object
print("Area Value:", obj.area()) # no argument
print("Area Value:", obj.area(4)) # one argument
print("Area Value:", obj.area(3, 5)) # two argument
```

```
Area Value: 0
Area Value: 16
Area Value: 15
```

Method Overriding: Method overriding is an example of the runtime polymorphism. When we use the method with the same name same parameters or signature and same return type, in child class which we already used in the parent class then the method in the child class is said to **override** the method in the super-class. The structure of the method overriding is as follows:



The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.

```
class parent1:
    def show(self):
        pritn("This is the parent1 class!")

class parent2:
    def disp(self):
        print("This is the parent2 class!")

class kid(parent1, parent2):
    def show(self):
        print("This the the child class!")

a=kid()
a.disp()
a.show()
```

```
This is the parent2 class!
This the the child class!
```

V. **Encapsulation:** Encapsulation in Python describes the concept of **binding data and methods within a single unit.** So, for example, when we create a class, it means we are implementing encapsulation. A class is an example of encapsulation as it binds all the data members (instance variables) and methods into a single unit.

VI. **Data Abstraction:**

Q2. Where does an inheritance search look for an attribute?

Ans: An inheritance search looks for an attribute first in object, then in all classes above it, from bottom to top and left to right. The search stops at the first place the attribute is found.

Q3. How do you distinguish between a class object and an instance object?

VII. **Ans:** Object is an entity that has a state and behavior associated with it. It may be a real-world entity like table, mouse, chair etc. Strings, Integers, floating point numbers even arrays and dictionaries all are objects.

To understand the state, behavior, and identity let us take the example of the class dog (explained above).

- The identity can be considered as the name of the dog.
- State or Attributes can be considered as the breed, age, or color of the dog.
- The behavior can be considered as to whether the dog is eating or sleeping.

On the other hand, In the python, an instance of a class is also called an object. The call will comprise both data members and methods and will be accessed by an object of that class.

Q4. What makes the first argument in a class's method function special?

Ans: Self: With this keyword, we can access the attributes and methods of the class in python. It binds the attributes with the given arguments.

```
class food():

# init method or constructor
    def __init__(self, fruit, color):
        self.fruit = fruit # Binding attribute with argument "fruite"
        self.color = color # Binding attribute with argument "color"

    def show(self):
        print("fruit is", self.fruit)
        print("color is", self.color )

apple = food("apple", "red")
grapes = food("grapes", "green")

apple.show()
grapes.show()
```

```
fruit is apple
color is red
fruit is grapes
color is green
```

Q5. What is the purpose of the init method?

Ans: The init function is just like a constructure as in C++. It automatically called when we create the object of the class in which it is created. The purpose of constructors is to initialize (assign values) to the data members of the class when an object of the class is created.

Q6. What is the process for creating a class instance?

Ans: To create instances of a class, we call the class using class name and pass in whatever arguments its __*init*__ method accepts.

```
class food():

# init method or constructor
    def __init__(self, fruit, color):
        self.fruit = fruit # Binding attribute with argument "fruite"
        self.color = color # Binding attribute with argument "color"

    def show(self):
        print("fruit is", self.fruit)
        print("color is", self.color )

apple = food("apple", "red") # Creating class instance & passing arguments to init
grapes = food("grapes", "green") # Creating class instance & passing arguments to init

apple.show()
grapes.show()
```

```
fruit is apple
color is red
fruit is grapes
color is green
```

Q7. What is the process for creating a class?

Ans: We can create a class by using keyword class:

```
class A:
    pass
```

Q8. How would you define the super classes of a class?

Ans: Super class are the parent of base class from which a child class can be derived.

```
# The base class
class Animal:
    def bird(self):#Function or Method
        print("Birds are chirping!")

# Now, we"ll derived the Animal class which is base class
class dog(Animal): # Child class of Animal class
    def mydog(self): #Function
        print("Dog is barking!")

# Now we'll derived the Dog class which is the child of Animal
class eat(dog): #Child class of Dog class
    def food(self):
        print("Birds eat seeds while dog eats dog food!")
a=eat()
a.bird()
a.mydog()
a.food()
```

```
Birds are chirping!
Dog is barking!
Birds eat seeds while dog eats dog food!
```

Q9. What is the relationship between classes and modules?

Ans: A class is used to define a blueprint for a given object, whereas a module is a python file with .py extension and used to reuse a given piece of code inside another program.

Q10. How do you make instances and classes?

Ans: To create instances of a class, you call the class using class name and pass in whatever arguments its *__init__* method accepts.

```
class Book():
   fontsize = 9
   page_width = 15
   def __init__(self, name, writer, length):
      self.name = name
      self.writer = writer
      self.length = length
book1 = Book("Intro to Python", "John Doe", 50000)
print(book1.writer)
print(book1.page_width)
print(book1.length)
```

```
John Doe
15
50000
```

Q11. Where and how should be class attributes created?

Ans: A class attribute should be created outside method. Accessed by class and instance objects. They belong to the class.

```
Class attr:

      x="class attribute"
      print("Output:")
obj=attr()
      print(obj.x)
```

Q12. Where and how are instance attributes created?

Ans: An instance attribute should be created inside **__init__ method.** Can be accessed by an instance of the class.

```
class attr:

y="class attribute"
def init (self,y):
      self.y=y
obj=attr()
print(obj.y)
```

Q13. What does the term "self" in a Python class mean?

Ans: The self-parameter is a reference to the current instance of the class and is used to access variables that belongs to the class. It does not have to be named self, you can call it whatever you like, but it must be the first parameter of any function in the class.

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

```
Hello my name is John
```

Q14. How does a Python class handle operator overloading?

Ans: The operator overloading in Python means provide extended meaning beyond their predefined operational meaning. Such as, we use the "+" operator for adding two integers as well as joining two strings or merging two lists. We can achieve this as the "+" operator is overloaded by the "int" class and "str" class. The user can notice that the same inbuilt operator or function is showing different behavior for objects of different classes. This process is known as operator overloading.

```
print (14 + 32)

# Now, we will concatenate the two strings
print ("Java" + "Tpoint")

# We will check the product of two numbers
print (23 * 14)

# Here, we will try to repeat the String
print ("X Y Z " * 4)
```

```
46
JavaTpoint
322
X Y Z X Y Z X Y Z X Y Z
```

Q16. What is the most popular form of operator overloading?

Ans: The most popular form of the operator overloading is addition (+) operator. We can use the "+" operator for adding two integers as well as joining two strings or merging two lists.

Q17. What are the two most important concepts to grasp in order to comprehend Python OOP code?

Ans: Both inheritance and polymorphism are fundamental concepts of object-oriented programming. These concepts help us to create code that can be extended, and easily maintainable Inheritance mean parent child relation. subclass and super class. Property of one class used in another chilled class. The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types. child class that has the same name as the methods in the parent class.

Q18. Describe three applications for exception processing.

Ans: In python, we catch exceptions and handle them using try and except code blocks. The try clause contains the code that can raise an exception, while the except clause contains the code lines that handle the exception. Let's see if we can access the index from the array, which is more than the array's length, and handle the resulting exception.

Q19. What happens if you don't do something extra to treat an exception?

Ans: If we don’t handle it, the program terminates abruptly and the code past the line that caused the exception will not get executed.

Q20. What are your options for recovering from an exception in your script?

Ans:

Q21. Describe two methods for triggering exceptions in your script.

Ans:

Q22. Identify two methods for specifying actions to be executed at termination time, regardless of whether an exception exists.

Ans:

Q23. What is the purpose of the try statement?

Ans: The try statement used to test a block of code for errors. The code with the exception(s) to catch. If an exception is raised, it jumps straight into the except block.

```
try:
# Some Code
except:
# Executed if error in the
# try block
```

Q24. What are the two most popular try statement variations?

Ans: Except: this code is only executed *if an exception occurred* in the try block. The except block is required with a try block, even if it contains only the pass statement.

It may be combined with the **else** and **finally** keywords.

Else: Code in the else block is only executed if no exceptions were raised in the try block.

Finally: The code in the final block is always executed, regardless of if an exception was raised or not.

```
try:
  print(x)
except:
  print("Something went wrong")
else:
    # else block will not executed because exception will occur in try
    print("Exception occured in the try statement")
finally:
  print("The 'try except' is finished")
```

```
Something went wrong
The 'try except' is finished
```

Q25. What is the purpose of the raise statement?

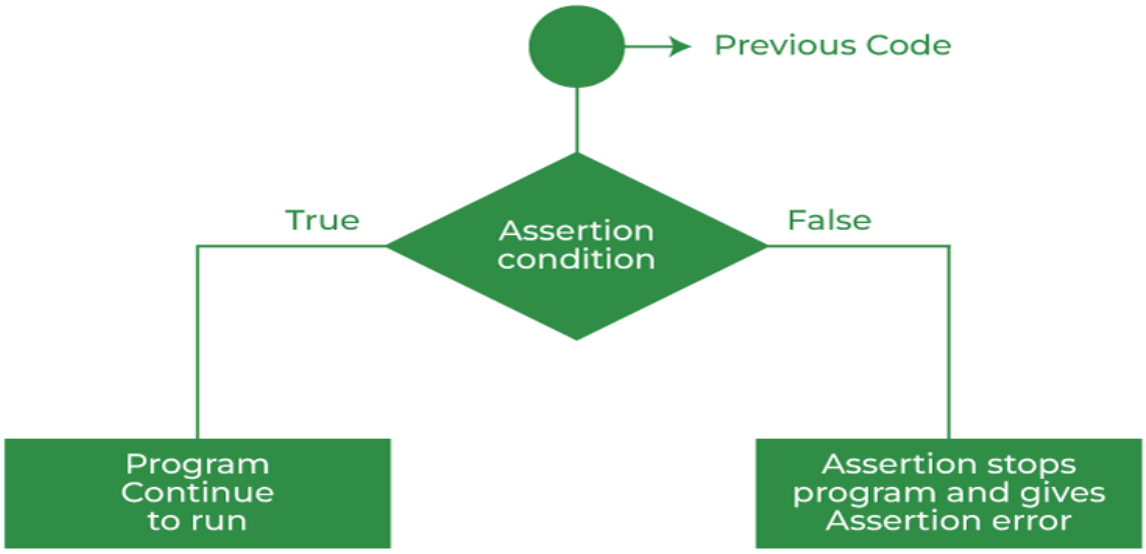
Ans: The raise keyword is used to raise an exception. We can define what kind of error to raise, and the text to print to the user.

```
x="Hello Python"
if not type(x) is int:
    raise TypeError("Only integers are allowed")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [11], in <cell line: 2>()
      1 x="Hello Python"
      2 if not type(x) is int:
----> 3     raise TypeError("Only integers are allowed")

TypeError: Only integers are allowed
```

Q26. What does the assert statement do, and what other statement is it like?

Ans: In simpler terms, we can say that assertion is the Boolean expression that checks if the statement is True or False. If the statement is true then it does nothing and continues the execution, but if the statement is False then it stops the execution of the program and throws an error. It is somehow like raise statement, but the difference is that raise statement is typically used when we have detected an error condition, or some condition does not satisfy. `assert` is similar but the exception is only raised if a condition is met (Like in example assert condition is met and it raised the message). The flow chart of the python assert statement is as follows:



The statement takes as input a Boolean condition, which when returns true doesn't do anything and continues the normal flow of execution, but if it is computed to be false, then it raises an AssertionError along with the optional message provided.

```
a=10
b=0
print("The value of a/b is:")
assert b!=0, "Zero Division Error"
print(a/b)
```

```
The value of a/b is:

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Input In [14], in <cell line: 4>()
      2 b=0
      3 print("The value of a/b is:")
----> 4 assert b!=0, "Zero Division Error"
      5 print(a/b)

AssertionError: Zero Division Error
```

Q27. What is the purpose of the with/as argument, and what other statement is it like?

Ans: In Python, with statement is used in exception handling to make the code cleaner and much more readable. It simplifies the management of common resources like file streams.

Q28. What are *args, **kwargs?

Ans: In the programming we define a function to make a code reusable that perform similar operation. To perform that operation, we call a function with the specific value (argument passing), this value is called a function argument.

```
def adder(x,y,z):
    print("Sum =",x+y+z)
adder(5,5,5)
```

```
Sum = 15
```

In the python we can pass variable number (many) of arguments toa function. To do so, we have two special symbols. We use *args and **kwargs as an argument when we are not sure about the number of arguments to pass in the function. Like in the following example we are not sure about the number of arguments to pass in the function. So the program will generate an error.

```
def adder(x,y,z):
    print("Sum =",x+y+z)
adder(5,5,5,5,7)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [4], in <cell line: 3>()
      1 def adder(x,y,z):
      2     print("Sum =",x+y+z)
----> 3 adder(5,5,5,5,7)

TypeError: adder() takes 3 positional arguments but 5 were given
```

1: *args (Keyword Arguments)

In the function, we use asterisk (*) before parameter name to pass variable length arguments.

```
def adder(*num):
    sum = 0

    for n in num:
        sum = sum + n

    print("Sum:",sum)

adder(3,5,7)
adder(4,5,6,7)
adder(1,2,3,5,6)
```

```
Sum: 15
Sum: 22
Sum: 17
```

In the above program, we used *num as a parameter which allows us to pass variable length argument list to the adder() function. Inside the function, we have a loop which adds the passed argument and prints the result. We passed 3 different tuples with variable length as an argument to the function.

2: **kwargs (Non-Keywords Arguments)

Python passes variable length non keyword argument to function using *args but we cannot use this to pass keyword argument. For this problem Python has got a solution called **kwargs, it allows us to pass the variable length of keyword arguments to the function.

In the function, we use the double asterisk ** before the parameter name to denote this type of argument. The arguments are passed as a dictionary and these arguments make a dictionary inside function with name.

```
def intro(**data):
    print("\nData type of argument:",type(data))

    for key, value in data.items():
        print("{} is {}".format(key,value))

intro(Firstname="Sita", Lastname="Sharma", Age=22, Phone=1234567890)
intro(Firstname="John", Lastname="Wood", Email="johnwood@nomail.com", Country="Wakanda", Age=25, Phone=9876543210)
```

```
Data type of argument: <class 'dict'>
Firstname is Sita
Lastname is Sharma
Age is 22
Phone is 1234567890

Data type of argument: <class 'dict'>
Firstname is John
Lastname is Wood
Email is johnwood@nomail.com
Country is Wakanda
Age is 25
Phone is 9876543210
```

Q29. How can I pass optional or keyword parameters from one function to another?

Ans: To pass, collect the arguments using the * and ** in the function's parameter list. Through this, you will get the positional arguments as a tuple and the keyword arguments as a dictionary. Pass these arguments when calling another function by using * and ** −

Q30. What are Lambda Functions?

Ans: A lambda function is an anonymous function (i.e., defined without a name) that can take any number of arguments but, unlike normal functions, evaluates and returns only one expression.

Q31. Explain Inheritance in Python with an example?

Ans: Kindly see the answer of the 1st question. I have explained the Inheritance there.

Q32. Suppose class C inherits from classes A and B as class C(A,B).Classes A and B both have their own versions of method func(). If we call func() from an object of class C, which version gets invoked?

Q33. Which methods/functions do we use to determine the type of instance and inheritance?

Ans: The isinstance() method checks whether an object is an instance of a class whereas issubclass() method asks whether one class is a subclass of another class (or other classes).

Q34.Explain the use of the 'nonlocal' keyword in Python.

Ans: The nonlocal keyword is used to work with variables inside nested functions, where the variable should not belong to the inner function. Use the keyword nonlocal to declare that the variable is not local.

Q35. What is the global keyword?

Ans: In Python, the global keyword allows you to change a variable value outside of its current scope. It is used to make changes to a global variable from a local location. The global keyword is only required for altering the variable value and not for publishing or accessing it.