**PYTHON OOP ASSIGNMENT**

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

Python is a multi-paradigm programming language. It supports different programming approaches.

One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

attributes

behavior

Let's take an example:

A parrot is an object, as it has the following properties:

name, age, color as attributes

singing, dancing as behavior

1. The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).
2. It helps us to write clean code which resembles a real world problem's solution. Thus, O.O.P. helps us to solve real world problems in much easier way.
3. It boosts easy maintainence, reusability and updation of the existing code base.
4. It gives our code a better modular approach. Thus partial updates are easy to do. In other hand, modularity helps us to add or improvise existing functionalities and check their dependencies in much easier way.
5. Frameworks are built around the OOP structure which helps us to render any component into our codebase without writing the entire structure of the framework. We can just create objects and use their methods to accomplish our task.

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

An inheritance search looks for an attribute first in the instance object, then in the class the instance was created from, then in all higher superclasses, progressing from left to right (by default). The search stops at the first place the attribute is found.

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

**class object:**

when we create a class in python then a class object is created so whenever python finds a class statement in the whole program then it creates a class object and assigns a name to that object i.e. class name. As we know in python, everything is an object so the class itself is an object and is the instance of metaclasses. Look at the following example

class MyClass:

pass

above code will generate a class object and name it ‘MyClass’. From this class object, we will create instance objects.

**Instance object:**

when we call a class, it creates an instance object of that class from which the object has been created. For example when we call the above-created class then it will create an instance object like this.

Obj1=MyClass()

the above statement creates an object and names it to Obj1 which is an instance of MyClass.

Instance objects are real objects in your python code process. The instance object has access to attributes of the class from which it is created. For example, Obj1 is the instance of class MyClass so, Now Obj1 can access everything defined in the class, and in the class object, we define the default behavior and properties of objects.

Reference - <https://artificialintelligencestechnology.com/python/difference-between-class-object-and-instance-object-in-python/>

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

1. self represents the instance of the class. By using the “self” we can access the attributes and methods of the class in python. It binds the attributes with the given arguments.

The reason you need to use self. is because Python does not use the @ syntax to refer to instance attributes. Python decided to do methods in a way that makes the instance to which the method belongs be passed automatically, but not received automatically: the first parameter of methods is the instance the method is called on.

1. It always receives the instance object that is the implied subject of the method call. It’s usually called 'self' by convention.

**Q5. What is the purpose of the init method?**

The Default \_\_init\_\_ Constructor in C++ and Java. Constructors are used to initializing the object’s state. The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created. Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

**EXAMPLE:**

# A Sample class with init method

class Person:

# init method or constructor

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

self.name = name

# Sample Method

def say\_hi(self):

print('Hello, my name is', self.name)

p = Person('Nikhil')

p.say\_hi()

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

To create instances of a class, you call the class using class name and pass in whatever arguments its \_\_init\_\_ method accepts.

"This would create first object of Employee class"

emp1 = Employee("Zara", 2000)

"This would create second object of Employee class"

emp2 = Employee("Manni", 5000)

**Q7. What is the process for creating a class?**

The class statement creates a new class definition. The name of the class immediately follows the keyword class followed by a colon as follows −

class ClassName:

'Optional class documentation string'

class\_suite

**Q8. How would you define the superclasses of a class?**

They are classes which are used to inherit from.

class Son(Father, Mother): ...

In this case Father and Mother are superclasses for Son subclass.

**Q9. What is the relationship between classes and modules?**

The difference between a class and a module in python is that a class is used to define a blueprint for a given object, whereas a module is used to reuse a given piece of code inside another program.

A class can have its own instance, but a module cannot be instantiated. We use the ‘class’ keyword to define a class, whereas to use modules, we use the ‘import’ keyword. We can inherit a particular class and modify it using inheritance. But while using modules, it is simply a code containing variables, functions, and classes.

Modules are files present inside a package, whereas a class is used to encapsulate data and functions together inside the same unit.

Ref - https://www.pythonpool.com/python-class-vs-module/

**Q10. How do you make instances and classes?**

To create instances of a class, you call the class using class name and pass in whatever arguments its \_\_init\_\_ method accepts.

"This would create first object of Employee class"

emp1 = Employee("Zara", 2000)

"This would create second object of Employee class"

emp2 = Employee("Manni", 5000)

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows −

emp1.displayEmployee()

emp2.displayEmployee()

print "Total Employee %d" % Employee.empCount

**Q11. Where and how should be class attributes created?**

**Class attributes** are the variables defined directly in the class that are shared by all objects of the class.

1)Defined directly inside a class.

2)Shared across all objects.

3)Accessed using class name as well as using object with dot notation, e.g. classname.class\_attribute or object.class\_attribute

4)Changing value by using classname.class\_attribute = value will be reflected to all the objects.

**Q12. Where and how are instance attributes created?**

**Instance attributes** are attributes or properties attached to an instance of a class. Instance attributes are defined in the constructor.

1)Defined inside a constructor using the self parameter.

2)Specific to object.

3)Accessed using object dot notation e.g. object.instance\_attribute

4)Changing value of instance attribute will not be reflected to other objects.

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

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 has to be the first parameter of any function in the class:

**Q14. How does a Python class handle operator overloading?**

Operator Overloading

Python allows us to change the default behavior of an operator depending on the operands that we use. This practice is referred to as "operator overloading".

The functionality of Python operators depends on built-in classes. However, the same operator will behave differently when applied to different types. A good example is the "+" operator. This operator will perform an arithmetic operation when applied on two numbers, will concatenate two strings, and will merge two lists.

Examples of Operator Overloading

>>> 4 + 4

8

>>> "Py" + "thon"

'Python'

In the first command, we have used the "+" operator to add two numbers. In the second command, we used the same operator to concatenate two strings.

In this case, the "+" operator has two interpretations. When used to add numbers, it is referred to as an "addition operator". When used to add strings, it is referred to as "concatenation operator". In short, we can say that the "+" operator has been overloaded for int and str classes.

To achieve operator overloading, we define a special method in a class definition. The name of the method should begin and end with a double underscore (\_\_). The + operator is overloaded using a special method named \_\_add\_\_(). This method is implemented by both the int and str classes.

Consider the following expression:

x + y

Python will interpret the expression as x.\_\_add\_\_(y). The version of \_\_add\_\_() that is called will depend on the types of x and y. For example:

>>> x, y = 5, 7

>>> x + y

12

>>> x.\_\_add\_\_(y)

12

>>>

The above example demonstrates how to use the + operator demonstrates how to use the + operator as well as its special method.With operator overloading, we are able to change the meaning of a Python operator within the scope of a class.

**Q15. When do you consider allowing operator overloading of your classes?**

Consider the following scenario: we have two objects that are physical representations of a class (user-defined data type). If we try to add two objects using the binary ‘+’ operator, the compiler throws an error since the compiler does not know how to add two objects. So we define a method for an operator, which is referred to as operator overloading. We can overload all existing operators but not build new ones. The method’s name should start and conclude with a double underscore (\_\_).

To overload the + operator, we must include the \_\_add\_\_() function in the class. Within this function, we can do whatever we want.

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

A very popular and convenient example is the Addition (+) operator.

Just think how the ‘+’ operator operates on two numbers and the same operator operates on two strings. It performs “Addition” on numbers whereas it performs “Concatenation” on strings.

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

Both inheritance and polymorphism are key ingredients for designing robust, flexible, and easy-to-maintain software.

**Inheritance**

It allows us to define a class that inherits all the methods and properties from another class.Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

**Example**

Create a class named Person, with firstname and lastname properties, and a printname method:

class Person:

def \_\_init\_\_(self, fname, lname):

self.firstname = fname

self.lastname = lname

def printname(self):

print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:

x = Person("John", "Doe")

x.printname()

**POLYMORHPHISM:**

The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function.

**Example:**

class India():

def capital(self):

print("New Delhi is the capital of India.")

def language(self):

print("Hindi is the most widely spoken language of India.")

def type(self):

print("India is a developing country.")

class USA():

def capital(self):

print("Washington, D.C. is the capital of USA.")

def language(self):

print("English is the primary language of USA.")

def type(self):

print("USA is a developed country.")

obj\_ind = India()

obj\_usa = USA()

for country in (obj\_ind, obj\_usa):

country.capital()

country.language()

country.type()

**Q18. Describe three applications for exception processing.**

Exception handling is used when the frequency of occurance of an exception cannot be predicted.

real world examples:]

1)you provide a web form for users to fill in and submit.but incase there are a lot of conditions to be handled and the conditions keeps changing periodically,you use exception handling to simplify the process

2)database connectivity uses exception handling(why???) this is because the reason for database connectivity failure cannot be predicted and handled as it can be caused by many variables such as power failure, unreachable server,failure at client front/back end and so on.

3)internet communication

4)arithmetic exceptions such as division by zero and so on.

5)operating systems use exception handling to resolve deadlocks,recover from crash and so forth

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

If we don’t treat an exception the execution will stop and an error will occur. And, the remaining code will not be executed

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

we can use try and except blocks for handling an exception

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

In Python, exceptions can be handled using a try statement.

The critical operation which can raise an exception is placed inside the try clause. The code that handles the exceptions is written in the except clause.

We can thus choose what operations to perform once we have caught the exception - **example**.

import sys

randomList = ['a', 0, 2]

for entry in randomList:

try:

print("The entry is", entry)

r = 1/int(entry)

break

except:

print("Oops!", sys.exc\_info()[0], "occurred.")

print("Next entry.")

print()

print("The reciprocal of", entry, "is", r)

**O/P:**

The entry is a

Oops! <class 'ValueError'> occurred.

Next entry.

The entry is 0

Oops! <class 'ZeroDivisionError'> occured.

Next entry.

The entry is 2

The reciprocal of 2 is 0.5

In this program, we loop through the values of the randomList list. As previously mentioned, the portion that can cause an exception is placed inside the try block.

If no exception occurs, the except block is skipped and normal flow continues(for last value). But if any exception occurs, it is caught by the except block (first and second values).

Here, we print the name of the exception using the exc\_info() function inside sys module. We can see that a causes ValueError and 0 causes ZeroDivisionError.

**Q22. Identify two methods for specifying actions to be executed at termination time, regardless of whether or not an exception exists.**

Finally block always executes irrespective of an exception being thrown or not. The final keyword allows you to create a block of code that follows a try-catch block.

Finally, clause is optional. It is intended to define clean-up actions which should be that executed in all conditions.

try:

raise KeyboardInterrupt

finally:

print 'welcome, world!'

**Output**

Welcome, world!

KeyboardInterrupt

**Q23. What is the purpose of the try statement?**

The try statement allows you to define a block of code to be tested for errors while it is being executed.

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

There are two other optional segments to a try block: else and finally. Both of these optional blocks will come after the try and the except. Also, there’s nothing stopping you from using both else and finally in a single statement — but keep them in that order if you do.

**Ex:**

When attaching an else statement to the end of a try/except, this code will be executed after the try has been completed, but only if no exceptions occur.

while True:

try:

num = int(input("Enter an int: "))

except Exception as e:

print(e)

else:

print("Thank you for the integer!")

break

# Enter an int: a

# invalid literal for int() with base 10: 'a'

# Enter an int: 3

# Thank you for the integer

**Try/Except/Finally**

When attaching a finally statement to the end of a try/except, this code will be executed after the try has been completed, regardless of exceptions.

**EX:**

count = 0

while True:

try:

num = int(input("Enter an int: "))

break

except Exception as e:

print(e)

finally:

count += 1

print("Attempt #:",count)

# Enter an int: a

# invalid literal for int() with base 10: 'a'

# Attempt #: 1

# Enter an int: 3

# Attempt #: 2

This might look a bit odd because the break is still inside the try. It’s reasonable to think that the finally would be cut short upon proper input, however, that’s not the case. The finally section will still execute, regardless of how the try is exited.

**Q25. What is the purpose of the raise statement?**

Python raise Keyword is used to raise exceptions or errors. The raise keyword raises an error and stops the control flow of the program. It is used to bring up the current exception in an exception handler so that it can be handled further up the call stack.

**Example:**In the below code, we check if an integer is even or odd. if the integer is odd an exception is raised. a is a variable to which we assigned a number 5, as a is odd, then if loop checks if it’s an odd integer, if it’s an odd integer then an error is raised.

a = 5

if a % 2 != 0:

raise Exception("The number shouldn't be an odd integer")

While raising an error we can also what kind of error we need to raise, and if necessary print out a text.

**Syntax:**

raise TypeError

**Example:**In the below code, we tried changing the string ‘apple’ assigned to s to integer and wrote a try-except clause to raise the ValueError. the raise keyword raises a value error with the message “String can’t be changed into an integer”.

s = 'apple'

try:

num = int(s)

except ValueError:

raise ValueError("String can't be changed into integer")

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

**Assertions** in any programming language are the debugging tools that help in the smooth flow of code. Assertions are mainly assumptions that a programmer knows or always wants to be true and hence puts them in code so that failure of these doesn’t allow the code to execute further.

**assert in Python**

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 along with the optional message provided.

**Example 1: Python assert keyword without error message**

a = 4

b = 0

# using assert to check for 0

print("The value of a / b is : ")

assert b != 0

print(a / b)

**Example 2: Python assert keyword with error message**

# initializing number

a = 4

b = 0

# using assert to check for 0

print("The value of a / b is : ")

assert b != 0, "Zero Division Error"

print(a / b)

**O/P:**

AssertionError: Zero Division Error

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

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. Observe the following code example on how the use of with statement makes code cleaner.

# file handling

**# 1) without using with statement**

file = open('file\_path', 'w')

file.write('hello world !')

file.close()

**# 2) without using with statement**

file = open('file\_path', 'w')

try:

file.write('hello world')

finally:

file.close()

**# using with statement**

with open('file\_path', 'w') as file:

file.write('hello world !')

Notice that unlike the first two implementations, there is no need to call file.close() when using with statement. The with statement itself ensures proper acquisition and release of resources. An exception during the file.write() call in the first implementation can prevent the file from closing properly which may introduce several bugs in the code, i.e. many changes in files do not go into effect until the file is properly closed. The second approach in the above example takes care of all the exceptions but using the with statement makes the code compact and much more readable. Thus, with statement helps avoiding bugs and leaks by ensuring that a resource is properly released when the code using the resource is completely executed. The with statement is popularly used with file streams, as shown above and with Locks, sockets, subprocesses and telnets etc.

**Q28. What are \*args, \*\*kwargs?**

The special syntax \*args in function definitions in python is used to pass a variable number of arguments to a function. It is used to pass a non-key worded, variable-length argument list.

The syntax is to use the symbol \* to take in a variable number of arguments; by convention, it is often used with the word args.

What \*args allows you to do is take in more arguments than the number of formal arguments that you previously defined. With \*args, any number of extra arguments can be tacked on to your current formal parameters (including zero extra arguments).

For example, we want to make a multiply function that takes any number of arguments and is able to multiply them all together. It can be done using \*args.

Using the \*, the variable that we associate with the \* becomes an iterable meaning you can do things like iterate over it, run some higher-order functions such as map and filter, etc.

**Example 1:**

Python program to illustrate \*args for a variable number of arguments

def myFun(\*argv):

for arg in argv:

print(arg)

myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')

**Output:**

Hello

Welcome

to

GeeksforGeeks

**Python \*\*kwargs**

The special syntax \*\*kwargs in function definitions in python is used to pass a keyworded, variable-length argument list. We use the name kwargs with the double star. The reason is that the double star allows us to pass through keyword arguments (and any number of them).

A keyword argument is where you provide a name to the variable as you pass it into the function.

One can think of the kwargs as being a dictionary that maps each keyword to the value that we pass alongside it. That is why when we iterate over the kwargs there doesn’t seem to be any order in which they were printed out.

**Example 1:**

Python program to illustrate \*kwargs for a variable number of keyword arguments. Here \*\*kwargs accept keyworded variable-length argument passed by the function call. for first=’Geeks’ first is key and ‘Geeks’ is a value. in simple words, what we assign is value, and to whom we assign is key.

def myFun(\*\*kwargs):

for key, value in kwargs.items():

print("%s == %s" % (key, value))

# Driver code

myFun(first='Geeks', mid='for', last='Geeks')

**Output:**

first == Geeks

mid == for

last == Geeks

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

Gather the arguments using the \* and \*\* specifiers in the function's parameter list. This gives us positional arguments as a tuple and the keyword arguments as a dictionary. Then we can pass these arguments while calling another function by using \* and \*\*:

def fun1(a, \*tup, \*\*keywordArg):

...

keywordArg['width']='23.3c'

...

Fun2(a, \*tup, \*\*keywordArg)

**Q30. What are Lambda Functions?**

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

**Syntax**

lambda arguments : expression

The expression is executed and the result is returned:

**Example**

Add 10 to argument a, and return the result:

x = lambda a : a + 10

print(x(5))

**Q31. Explain Inheritance in Python with an example?**

**Inheritance**

It allows us to define a class that inherits all the methods and properties from another class.Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

**Example**

Create a class named Person, with firstname and lastname properties, and a printname method:

class Person:

def \_\_init\_\_(self, fname, lname):

self.firstname = fname

self.lastname = lname

def printname(self):

print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:

x = Person("John", "Doe")

x.printname()

**o/p:** John Doe

**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?**

class A:

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

self.name = name

def func(self):

print(self.name, "is not good")

class B:

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

self.name = name

def func(self):

print(self.name, "is good")

class C(A,B):

def whichColor(self):

print(self.name, "is color")

c1 = C("Anusha")

c1.func()

**O/P:**

Anusha is not good

If we call func() from an object of class C, first parent class gets invoked

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

**isinstance() and issubclass()**

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).

**isinstance(object, classinfo)**

Return true if the object argument is an instance of the classinfo argument, or of a (direct, indirect or virtual) subclass thereof.

**issubclass(class, classinfo)**

Return true if class is a subclass (direct, indirect or virtual) of classinfo. A class is considered a subclass of itself.

**Example**

class MyClass(object):

pass

class MySubClass(MyClass):

pass

print(isinstance(MySubClass, object))

print(issubclass(MySubClass, MyClass))

print(isinstance(MySubClass, MyClass))

**O/P:**

True

True

False

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

**Usage**

If a variable is declared in an enclosing (outer) function, it is nonlocal to nested (inner) function.

The nonlocal keyword is used to update these variables inside a nested function. The usage of nonlocal is very similar to that of global, except that the former is primarily used in nested functions.

**Basic Example**

Here’s a basic example that tries to reassign enclosing function’s local variable inside a nested function.

# enclosing function

def f1():

x = 42

# nested function

def f2():

x = 0

print(x) # x is 0

f2()

print(x) # x is still 42

f1()

Here, the value of existing variable x didn’t change. Because, Python created a new local variable named x that shadows the variable in the outer scope.

**Preventing that behavior is where the nonlocal keyword comes in.**

# enclosing function

def f1():

x = 42

# nested function

def f2():

nonlocal x

x = 0

print(x) # x is now 0

f2()

print(x) # x remains 0

f1()

The x inside the nested function now refers to the x outside the function, so changing x inside the function changes the x outside it.

**Q35. What is the global keyword?**

The global keyword is used to create global variables from a no-global scope, e.g. inside a function.

Declare a global variable inside a function, and use it outside the function:

#create a function:

def myfunction():

global x

x = "hello"

#execute the function:

myfunction()

#x should now be global, and accessible in the global scope.

print(x)

**O/P:**

hello