Q1. What is the relationship between classes and modules?

**A module can have zero or one or multiple classes.** **A class can be implemented in one or more .** **py files (modules)**. But often, we can organize a set of variables and functions into a class definition or just simply put them in a

Q2. 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**.

An instance is a specific object created from a particular class. To create instances of a class, **call the class using the class name and pass in whatever arguments its \_\_init\_\_ method accepts**—in this example, the \_\_init\_\_ method takes name , age , and rating .

Q3. Where and how should be class attributes created?

Class attributes belong to the class itself they will be shared by all the instances. Such attributes are **defined in the class body parts usually at the top**, for legibility. Unlike class attributes, instance attributes are not shared by objects.

Q4. Where and how are instance attributes created?

# Write Python code here

class sampleclass:

    count = 0     # class attribute

    def increase(self):

        sampleclass.count += 1

# Calling increase() on an object

s1 = sampleclass()

s1.increase()

print(s1.count)

# Calling increase on one more

# object

s2 = sampleclass()

s2.increase()

print(s2.count)

print(sampleclass.count)

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

class Student:

count = 0

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

Student.count += 1

In the above example, count is an attribute in the Student class. Whenever a new object is created, the value of count is incremented by 1. You can now access the count attribute after creating the objects, as shown below

>>> std1=Student()

>>> Student.count

1

>>> std2 = Student()

>>> Student.count

2

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

The following demonstrates the instance attributes.

Example: Setting Attribute Values Copy

class Student:

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

self.name = name

self.age = age

Now, you can specify the values while creating an instance, as shown below.

Example: Passing Instance Attribute Values in Constructor Copy

>>> std = Student('Bill',25)

>>> std.name

'Bill'

>>> std.age

25

>>> std.name = 'Steve'

>>> std.age = 45

>>> std.name

'Steve'

>>> std.age

45

Q5. What does the term “self” in a Python class mean?

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.

#it is clearly seen that self and obj is referring to the same object

class check:

    def \_\_init\_\_(self):

        print("Address of self = ",id(self))

obj = check()

print("Address of class object = ",id(obj))

# this code is Contributed by Dipans Verma

**Output**

Address of self = 140124194801032

Address of class object = 140124194801032

Q6. How does a Python class handle operator overloading?

Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because ‘+’ operator is overloaded by int class and str class. You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called *Operator Overloading*. 

* Python3

|  |
| --- |
| # Python program to show use of  # + operator for different purposes.    print(1 + 2)    # concatenate two strings  print("Geeks"+"For")    # Product two numbers  print(3 \* 4)    # Repeat the String  print("Geeks"\*4) |

**Output**

3

GeeksFor

12

GeeksGeeksGeeksGeeks

#### Python magic methods or special functions for operator overloading

#### [Binary Operators](https://www.geeksforgeeks.org/basic-operators-python/):

|  |  |
| --- | --- |
| Operator | Magic Method |
| **+** | \_\_add\_\_(self, other) |
| **–** | \_\_sub\_\_(self, other) |
| **\*** | \_\_mul\_\_(self, other) |
| **/** | \_\_truediv\_\_(self, other) |
| **//** | \_\_floordiv\_\_(self, other) |
| **%** | \_\_mod\_\_(self, other) |
| **\*\*** | \_\_pow\_\_(self, other) |
| >> | \_\_rshift\_\_(self, other) |
| << | \_\_lshift\_\_(self, other) |
| & | \_\_and\_\_(self, other) |
| | | \_\_or\_\_(self, other) |
| ^ | \_\_xor\_\_(self, other) |

#### Comparison Operators :

|  |  |
| --- | --- |
| Operator | Magic Method |
| **<** | \_\_lt\_\_(self, other) |
| **>** | \_\_gt\_\_(self, other) |
| **<=** | \_\_le\_\_(self, other) |
| **>=** | \_\_ge\_\_(self, other) |
| **==** | \_\_eq\_\_(self, other) |
| **!=** | \_\_ne\_\_(self, other) |

#### Assignment Operators :

|  |  |
| --- | --- |
| Operator | Magic Method |
| **-=** | \_\_isub\_\_(self, other) |
| **+=** | \_\_iadd\_\_(self, other) |
| **\*=** | \_\_imul\_\_(self, other) |
| **/=** | \_\_idiv\_\_(self, other) |
| **//=** | \_\_ifloordiv\_\_(self, other) |
| **%=** | \_\_imod\_\_(self, other) |
| **\*\*=** | \_\_ipow\_\_(self, other) |
| **>>=** | \_\_irshift\_\_(self, other) |
| **<<=** | \_\_ilshift\_\_(self, other) |
| **&=** | \_\_iand\_\_(self, other) |
| **|=** | \_\_ior\_\_(self, other) |
| **^=** | \_\_ixor\_\_(self, other) |

#### Unary Operators :

|  |  |
| --- | --- |
| Operator | Magic Method |
| **–** | \_\_neg\_\_(self) |
| **+** | \_\_pos\_\_(self) |
| **~** | \_\_invert\_\_(self) |

**Note:**It is not possible to change the number of operands of an operator. For ex. you cannot overload a unary operator as a binary operator. The following code will throw a syntax error.

* Python3

|  |
| --- |
| # Python program which attempts to  # overload ~ operator as binary operator    class A:      def \_\_init\_\_(self, a):          self.a = a        # Overloading ~ operator, but with two operands      def \_\_invert\_\_(self, other):          return "This is the ~ operator, overloaded as binary operator."      ob1 = A(2)  ob2 = A(3)    print(ob1~ob2) |

Q7. When do you consider allowing operator overloading of your classes?

 The + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings. This feature in Python that **allows the same operator to have different meaning according to the context** is called operator overloading.

Operator Overloading means **giving extended meaning beyond their predefined operational meaning**. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because '+' operator is overloaded by int class and str class.

# Python program to show use of

# + operator for different purposes.

print(1 + 2)

# concatenate two strings

print("Geeks"+"For")

# Product two numbers

print(3 \* 4)

# Repeat the String

print("Geeks"\*4)

**Output**

3

GeeksFor

12

GeeksGeeksGeeksGeeks

Q8. 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.

In object-oriented programming, there exists this concept called **“Polymorphism”**. Polymorphism means **“one action, many forms”**. OOP allows objects to perform a **single** action in **different** ways. One way of implementing Polymorphism is through **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.

Operators in Python work for **built-in** classes, like **int**, **str**, **list**, etc. But you can extend their **operability** such that they work on objects of user-defined classes too.

Let’s try it!

**class** bubble:

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

self.volume = volume

We have defined a class bubble which has an **attribute volume**. Let’s see what happens when we combine two bubbles. We should get the volume of the combined bubble.

>>> b1 = bubble(20)

>>> b2 = bubble(30)

And now let’s add b1 and b2 to merge the bubbles.

>>> b1 + b2

**Output:**

TypeError: unsupported operand type(s) for +: ‘bubble’ and ‘bubble’

Clearly this doesn’t work right now. That’s because we haven’t extended its operability yet and it only works on **built-in** classes.

So how do we make these operators work on our user-defined class bubble as well? This is where **“magic methods”** come into the picture.

Magic methods in Python are special methods that begin and end with a **double underscore( \_\_ )**.

The**\_\_init\_\_()** is one such method.

Another magic method at our disposal is the**\_\_str\_\_()** method.

The**\_\_str\_\_()** method lets you control how an object of your class gets **printed**.

So if we add this method to our bubble class and print an object of the class, it should work as follows:

**class** bubble:

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

self.volume = volume

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

**return** "volume is " + str(self.volume)

Now do this in the terminal:

>>> b1 = bubble(20)

>>> print(b1)

**Output:**

volume is 20  
>>>

Now we can perform an **addition** on objects of our class.

Let’s see what happens behind the scenes.

* When we add b1 + b2, the interpreter calls b1.\_\_add\_\_(b2).
* And b1.\_\_add\_\_(b2) is actually executed as bubble.\_\_add\_\_(b1, b2).
* This will then return bubble(50).
* So, b3 = b1 + b2 is actually equivalent to b3 = bubble(50).

In this way, we can overload other operators as well.

Note that in the case of comparison operators, the magic method will return a **boolean expression** as a result of the **comparison** and not an **object**.

You’ll find various python operators and their magic methods in the table below.

### Magic methods in Python

|  |  |  |
| --- | --- | --- |
| **OPERATOR** | **EXPRESSION** | **MAGIC METHOD** |
| Addition | b1 + b2 | **\_\_add\_\_()** |
| Subtraction | b1 – b2 | **\_\_sub\_\_()** |
| Multiplication | b1 \* b2 | **\_\_mul\_\_()** |
| Division | b1 / b2 | **\_\_truediv\_\_()** |
| Power | b1 \*\* b2 | **\_\_pow\_\_()** |
| Floor division | b1 // b2 | **\_\_floordiv\_\_()** |
| Modulo operator | b1 % b2 | **\_\_mod\_\_()** |
| Bitwise left shift | b1 << b2 | **\_\_lshift\_\_()** |
| Bitwise right shift | b1 >> b2 | **\_\_rshift\_\_()** |
| Bitwise NOT | ~b1 | **\_\_invert\_\_()** |
| Bitwise AND | b1 & b2 | **\_\_and\_\_()** |
| Bitwise OR | b1 | b2 | **\_\_or\_\_()** |
| Bitwise XOR | b1 ^ b2 | **\_\_xor\_\_()** |
| Less than | b1 < b2 |  |

|  |  |  |
| --- | --- | --- |
| Less than | b1 < b2 | **\_\_lt\_\_()** |
| Less than equal to | b1 <= b2 | **\_\_le\_\_()** |
| Greater than | b1 > b2 | **\_\_gt\_\_()** |
| Greater than equal to | b1 >= b2 | **\_\_ge\_\_()** |
| Equal to | b1 == b2 | **\_\_eq\_\_()** |
| Not equal to | b1 !=  b2 | **\_\_ne\_\_()** |

**Q9. What are the two most important concepts to grasp in order to comprehend Python OOP code?**

Object oriented programming (OOP) paradigm is built around the idea of having objects that belong to a particular type. In a sense, the type is what explains us the object.

Everything in Python is an object and every object has a type. These types are declared and defined using [classes](https://towardsdatascience.com/a-comprehensive-guide-for-classes-in-python-e6bb72a25a5e). Thus, classes can be considered as the heart of OOP.

In order to develop robust and well-designed software products with Python, it is essential to obtain a comprehensive understanding of OOP. In this article, we will elaborate on two key concepts of OOP which are inheritance and polymorphism.

Both inheritance and polymorphism are key ingredients for designing robust, flexible, and easy-to-maintain software. These concepts are best explained via examples. Let’s start with creating a simple class.

class Employee(): def \_\_init\_\_(self, emp\_id, salary):  
 self.emp\_id = emp\_id  
 self.salary = salary def give\_raise(self):  
 self.salary = self.salary \* 1.05

We have created a class called Employee. It has two data attributes which are employee id (emp\_id) and salary. We have also defined a method called give\_raise. It applies a 5-percent increase on the salary of an employee.

We can create an instance of the Employee class (i.e. an object with Employee type) and apply the give\_raise method as follows:

emp1 = Employee(1001, 56000)print(emp1.salary)  
56000emp1.give\_raise()print(emp1.salary)  
58800.0

OOP allows us to create a class based on another class. For instance, we can create the Manager class based on the Employee class.

class Manager(Employee):  
 pass

In this scenario, Manager is said to be a child class of the Employee class. The child class copies the attributes (both data and procedural) from the parent class. This concept is called **inheritance.**

It is important to note that inheritance does not mean copying a class. We can partially inherit from a parent (or base class). Python also allows for adding new attributes as well as modifying the existing ones. Thus, inheritance comes with a great deal of flexibility.

We can now create a manager object just like we create an employee object.

mgr1 = Manager(101, 75000)  
print(mgr1.salary)  
75000

A child class can have new attributes in addition to the ones inherited from the parent class. Furthermore, we have the option to modify or override the inherited attributes.

Let’s update the give\_raise method so that it applies a 10-percent increase for the managers.

class Manager(Employee): def give\_raise(self):  
 self.salary = self.salary \* 1.10mgr1 = Manager(101, 75000)  
print(mgr1.salary)  
75000mgr1.give\_raise()  
print(mgr1.salary)  
82500

We will create another child class of the Employee class. The Director class inherits the attributes from the Employee class and modifies the give\_raise method with a 20-percent increase.

class Director(Employee): def give\_raise(self):  
 self.salary = self.salary \* 1.20

We now have three different class and they all have a give\_raise method. Although the name of the method is the same, it behaves differently for different type of objects. This is an example of **polymorphism**.

Polymorphism allows for leveraging the same interface for different underlying operations. Regarding our example of manager and director objects, we can use them as they were an instance of the employee class.

Let’s see polymorphism in action. We will define a function that applies raise to a list of employees.

def bulk\_raise(list\_of\_emps):  
 for emp in list\_of\_emps:  
 emp.give\_raise()

The bulk\_raise function takes a list of employees and apply the give\_raise function to each object in the list. The next step is to create a list of employees of different types.

emp1 = Employee(101, 45000)  
emp2 = Manager(103, 60000)  
emp3 = Director(105, 70000)list\_of\_emps = [emp1, emp2, emp3]

Our list contains one employee, one manager, and one director objects. We can now call the bulk\_raise function.

bulk\_raise(list\_of\_emps)print(emp1.salary)  
47250.0print(emp2.salary)  
66000.0print(emp3.salary)  
84000.0

Although each object in the list has a different type, we do not have to explicitly state it in our function. Python knows which give\_raise method to apply.

As we see in the examples, polymorphism is accomplished using inheritance. Subclasses (or child classes) make use of the methods from the parent class to establish polymorphism.

## Conclusion

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 is a great way to eliminate unnecessary repetitive code. A child class can inherit from the parent class partially or entirely. Python is quite flexible with regards to inheritance. We can add new attributes and methods as well as modify the existing ones.

Polymorphism contributes to Python’s flexibility as well. An object with a particular type can be used as if it belonged to a different type. We have seen an example of it with the give\_raise method.