Python OOPs Interview Questions Overview

Object-oriented programming (OOP) is an important paradigm in software development, and Python is a popular programming language that supports OOP concepts. If you are preparing for a Python programming job interview, it's important to have a good understanding of OOP and be prepared to answer Python OOPs interview questions.

In this article, we will cover some common OOP interview questions that you might encounter in a Python programming interview. We will discuss the basics of OOP in Python, including classes, objects, inheritance, and polymorphism, and provide sample interview questions and answers to help you prepare. By the end of this article, you should have a better understanding of OOP in Python and be better prepared for your next interview.

Top 50 Python OOPs Interview Questions and Answers

Python is a popular programming language that supports Object-Oriented Programming (OOP) principles. OOP is a programming paradigm that allows for the creation of reusable code by defining objects that encapsulate data and behaviour. Python's OOP capabilities make it a popular choice for building large-scale applications, making it an essential topic for interviewers. The Top 50 Python OOPs Interview Questions provide a comprehensive list of questions that interviewers might ask candidates to assess their understanding of OOP principles, their proficiency in using Python's OOP features, and their ability to apply OOP concepts to solve real-world problems. These questions cover a range of topics, including classes and objects, inheritance and polymorphism, abstract classes and interfaces, encapsulation, and more. A good understanding of these concepts and their application in Python programming can help candidates stand out during an interview and showcase their expertise in OOP programming.

Q1: What is OOPS?

OOPS stands for Object-Oriented Programming System, which is a programming paradigm based on the concept of "objects". It is a way of organizing and designing computer programs using objects, which are instances of classes. The primary goal of OOPS is to make programming more modular, reusable, and maintainable.

In OOPS, objects represent real-world entities or concepts and have properties (attributes) and behaviours (methods) associated with them. These objects interact with each other by sending messages and exchanging data. OOPS provides several concepts such as encapsulation, inheritance, and polymorphism, which help in designing robust and flexible programs.

Q2: What is a class in Python?

A class in Python is a blueprint for creating objects. It is a user-defined data type that defines a set of attributes (variables) and methods (functions) that the objects of the class will have. A class is defined using the class keyword, followed by the class name, a colon, and the class body.

Here is an example of a simple class definition in Python:

```
class Person:
def __init__(self, name, age):
self.name = name
self.age = age
def greet(self):
print("Hello, my name is", self.name)
person1 = Person("John", 30)
person2 = Person("Jane", 25)
```

In this example, the Person class has two attributes (name and age) and one method (greet). The __init__ method is a special method that is called when an object of the class is created, and it initializes the attributes with the values passed as arguments. The greet method is a regular method that takes no arguments and prints a greeting message.

Q3: What is the difference between a class and an object?

A: In Python, a class is a blueprint or a template for creating objects. It defines the attributes and methods that an object of that class will have. On the other hand, an object is an instance of a class that has its own unique identity and state. An object is created from a class using the constructor method, and it can access the attributes and methods defined in the class.

Class	**Object**
A blueprint or template for creating objects	An instance of a class
Defines the attributes and behaviour that an object will have	Contains the data and functions defined by the class
Can have multiple instances	Refers to a single instance of a class
Can have class-level variables and methods	Has its own unique set of instance variables
Can be inherited by another class	Can be passed as a parameter to a function
Created using the 'class' keyword	Created using the constructor of a class
Example: 'class Car:'	Example: 'my_car = Car()'

Q4: What is encapsulation in Python?

A: Encapsulation is one of the four fundamental principles of object-oriented programming (OOP) that allows data to be hidden and protected from outside interference. In Python, encapsulation is implemented using access modifiers, such as private, public, and protected. By using these access modifiers, we can restrict the access of attributes and methods of a class from outside the class.

Q5: What is inheritance in Python?

A: Inheritance is another fundamental principle of OOP that allows a class to inherit the attributes and methods of another class. The class that inherits the properties of another class is called the child class or subclass, and the class whose properties are inherited is called the parent class or superclass. In Python, inheritance is implemented using the keyword "class" followed by the name of the child class and the name of the parent class in parentheses.

Q6: What is polymorphism in Python?

A: Polymorphism is the ability of an object to take on many forms. In Python, polymorphism is implemented using method overriding and method overloading. Method overriding is when a subclass defines a method that has the same name and parameters as a method in its parent class, but with different functionality. Method overloading is when a class defines multiple methods with the same name but with different parameters.

Q7: What is an abstraction in Python?

A: Abstraction is the process of hiding complex implementation details and providing a simpler interface for the user. In Python, abstraction is implemented using abstract classes and interfaces. An abstract class is a class that cannot be instantiated and has one or more abstract methods, which must be implemented by its subclasses. An interface is a blueprint for a class that defines the methods that must be implemented by the class.

Q8: What is the difference between abstraction and encapsulation?

A: Abstraction and encapsulation are two fundamental principles of OOP, but they serve different purposes. Encapsulation is used to hide the implementation details of a class from outside interference, while abstraction is used to provide a simpler interface for the user. Encapsulation is implemented using access modifiers, while abstraction is implemented using abstract classes and interfaces. In other words, encapsulation protects the data and methods of a class, while abstraction simplifies the usage of a class.

Abstraction	**Encapsulation**
A technique for hiding implementation details and exposing only necessary information to the user	A technique for hiding implementation details and protecting the data from unwanted access
It is achieved using abstract classes and interfaces	It is achieved by making use of access modifiers such as public, private, and protected
It focuses on what the object does and how it is used	It focuses on how the object is implemented and how the data is accessed
It allows the user to work with high-level objects without worrying about low-level details	It prevents the user from accessing data directly and enforces the use of methods for accessing and modifying data
It helps in reducing complexity and increasing reusability	It helps in maintaining the integrity of data and preventing unauthorized access
Example: A Shape class with subclasses Circle and Square	Example: A BankAccount class with private data members such as account number and balance

Abstraction is about the user interface or API of a class	Encapsulation is about the internal implementation of a class

Q9: What is a constructor in Python?

A: A constructor is a special method that is automatically called when an object is created from a class. It is used to initialize the attributes of the object to some default or user-defined values. In Python, the constructor method is named "init()" and it takes the "self" parameter, which refers to the object being created, and any other parameters that the user wants to pass.

Q10: What is the use of *init* method in Python?

A: The "init()" method in Python is used to initialize the attributes of an object when it is created from a class. It can be used to set default values for the attributes or to accept values from the user. The "init()" method is called automatically when an object is created, and it takes the "self" parameter, which refers to the object being created, and any other parameters that the user wants to pass.

Q11: What is a destructor in Python?

A: A destructor is a special method that is automatically called when an object is about to be destroyed or removed from memory. It is used to free up any resources that were allocated by the object during its lifetime, such as file handles or database connections. In Python, the destructor method is named "del()" and it takes only the "self" parameter.

Q12: How do you create an instance of a class in Python?

A: To create an instance of a class in Python, you need to use the constructor method "init()" of the class. The syntax for creating an instance of a class is as follows:

object_name = ClassName()

Here, "object_name" is the name of the object that you want to create, and "ClassName" is the name of the class that you want to instantiate. The empty parentheses after the class name indicate that no parameters are being passed to the constructor method.

Q13: What is the difference between instance and class variables in Python?

A: Instance variables are variables that are unique to each instance of a class. They are created and initialized inside the constructor method "init()" and can be accessed using the "self" keyword. Instance variables have different values for each object of the class.

On the other hand, class variables are variables that are shared among all instances of a class. They are defined outside the constructor method and can be accessed using the class name or the "cls" keyword. Class variables have the same value for all objects of the class.

Q14: How do you access class variables in Python?

A: Class variables in Python can be accessed using either the class name or the "cls" keyword. To access a class variable using the class name, you can use the following syntax:

ClassName.variable_name

Here, "ClassName" is the name of the class and "variable_name" is the name of the class variable.

Alternatively, you can also access a class variable using the "cls" keyword inside a class method. The syntax for accessing a class variable using the "cls" keyword is as follows:

class ClassName:

variable_name = value

@classmethod

def class_method(cls):

print(cls.variable_name)

Here, "cls" refers to the class itself, and "variable_name" is the name of the class variable that you want to access. The "class_method()" method is a class method that can be used to access the class variable.

Q15: What is the difference between public, private, and protected access specifiers in Python?

A: In Python, there are no real access specifiers like in some other programming languages. However, there are conventions that can be used to indicate the access level of class attributes.

Public attributes are accessible from anywhere, both inside and outside the class. By convention, public attributes are named with a single leading underscore, for example "_name".

Protected attributes are intended to be accessed only within the class and its subclasses. By convention, protected attributes are named with a single leading underscore, followed by the attribute name, for example "_name".

Private attributes are intended to be accessed only within the class. By convention, private attributes are named with a double leading underscore, followed by the attribute name, for example "__name".

Q16: How do you implement private and protected access specifiers in Python?

A: In Python, the private and protected access specifiers are implemented using naming conventions, as described above. By convention, attributes with a single leading underscore are considered protected, and attributes with a double leading underscore are considered private. However, this is just a convention, and it is still possible to access these attributes from outside the class if necessary.

Q17: What is method overriding in Python?

A: Method overriding is a feature of object-oriented programming in which a subclass provides its own implementation of a method that is already defined in its parent class. When a method is called on an object of the subclass, the subclass's implementation of the method is executed instead of the parent class's implementation.

To override a method in Python, you simply define a method with the same name in the subclass, and it will automatically override the method in the parent class.

Q18: What is method overloading in Python?

A: Method overloading is a feature of some programming languages in which a method can have multiple definitions with different parameters. In Python, method overloading is not supported, because it is a dynamically typed language. Instead, you can achieve similar functionality using default arguments or variable-length argument lists.

Q19: What is a static method in Python?

A: A static method is a method that belongs to a class rather than an instance of the class. It is defined using the "@staticmethod" decorator and does not take the "self" parameter that is normally used to refer to the object that the method is being called on. Static methods can be called on the class itself, without needing to create an instance of the class.

Q20: What is a class method in Python?

A: A class method is a method that belongs to a class rather than an instance of the class. It is defined using the "@classmethod" decorator and takes the "cls" parameter that refers to the class itself. Class methods are typically used to modify or access class-level variables or to create new instances of the class. They can be called on the class itself, without needing to create an instance of the class.

Q21: What is a decorator in Python?

A: A decorator in Python is a design pattern that allows you to modify the behaviour of a function or class without changing its source code. Decorators are implemented using the '@' symbol followed by the name of the decorator function, which is then placed above the function or class definition. The decorator function takes in the original function or class as an argument and returns a new function or class with modified behaviour.

Q22: What is the difference between @staticmethod and @classmethod in Python?

A: Both @staticmethod and @classmethod are used to define methods in Python that can be called on the class itself, rather than on an instance of the class. The main difference between the two is that @staticmethod does not receive any special first argument, while @classmethod receives the class itself as its first argument. This means that @classmethod can access class-level data and call other class methods, while @staticmethod cannot.

Q23: What is a module in Python?

A: In Python, a module is a file containing Python definitions and statements. A module can define functions, classes, and variables, and can also include runnable code. Modules are used to organize code into reusable and sharable units, and can be imported into other modules or scripts using the 'import' statement.

Q24: What is the difference between a module and a package in Python?

A: A module is a single file containing Python code, while a package is a directory containing one or more modules and an optional 'init.py' file. The 'init.py' file is used to mark the directory as a Python package and can contain initialization code for the package. Packages are used to organize modules into a hierarchical structure, making it easier to manage large codebases and to distribute reusable code.

Q25: How do you import a module in Python?

A: To import a module in Python, use the 'import' statement followed by the name of the module. For example, to import the 'math' module, you can use the following statement:

import math

You can then access functions and variables defined in the module using the dot notation, such as:

result = math.sqrt(25)

Q26: How do you create a package in Python?

A: To create a package in Python, you need to create a directory with a unique name and an 'init.py' file. The 'init.py' file can be empty, or it can contain initialization code for the package. You can then add one or more modules to the package by creating Python files within the package directory. For example, to create a package named 'my_package' with a module named 'my_module', you can create the following directory structure:

```
my_package/

├── __init__.py

└── my_module.py
```

You can then import the module in your Python code using the dot notation, such as:

```
from my_package import my_module
```

Q27: What is a diamond problem in multiple inheritances?

A: The diamond problem is a common issue in programming languages that support multiple inheritances, including Python. It occurs when a class inherits from two or more classes that have a common ancestor. In this scenario, if a method is defined in the common ancestor and is not overridden by the derived classes, it will be called multiple times during method resolution, leading to unexpected behaviour.

Q28: How do you resolve the diamond problem in Python?

A: In Python, the diamond problem can be resolved using the method resolution order (MRO) algorithm. The MRO determines the order in which Python looks for a method in a hierarchy of classes. To avoid the diamond problem, Python follows the C3 linearization algorithm, which creates a linear order of the classes that preserves the order of method calls.

Q29: What is a mixin in Python?

A: A mixin is a special kind of multiple inheritance in Python that is used to add a specific behaviour to a class without changing its hierarchy. Mixins are typically small, reusable classes that are designed to be mixed into other classes. They can define methods, properties, and other attributes that can be used by the target class.

Q30: How do you implement a mixin in Python?

A: To implement a mixin in Python, create a new class that defines the behaviour you want to add to your target class. Then, use the mixin class as a base class for your target class. For example, to add a 'debug' method to a class, you can define a mixin class like this:

```
class DebugMixin:

def debug(self):

print(self.__dict__)
```

Then, you can use this mixin class in your target class like this:

```
class MyClass(DebugMixin):

def __init__(self, value):

self.value = value
```

Now, instances of MyClass have access to the 'debug' method defined in the DebugMixin class.

Q31: What is a super() function in Python?

A: The super() function is a built-in function in Python that allows you to call a method in a superclass from a subclass. It provides a way to access the methods and attributes of a superclass that have been overridden or extended in a subclass. The super() function returns a temporary object of the superclass, which allows you to call its methods and access its attributes.

Q32: How do you use super() function in Python?

A: To use the super() function in Python, call it with two arguments: the name of the subclass and an instance of the subclass. For example, to call a method in a superclass from a subclass, you can use the following code:

```
class MySubclass(MySuperclass):

def my_method(self):

super(MySubclass, self).my_method()

# additional code for the subclass
```

In this example, the super() function is used to call the 'my_method' method in the superclass, and the subclass can then add additional code to extend or override the behaviour of the method.

Q33: What is the use of init.py in Python?

A: __init__.py is a special Python file that is used to mark a directory as a Python package. When a directory contains an __init__.py file, it tells Python that the directory should be treated as a package and allows you to import modules from that package. You can also use __init__.py to execute code when the package is imported, define package-level attributes or functions, or import modules that should be included in the package's namespace.

Q34: What is a namespace in Python?

A: A namespace in Python is a mapping between names (i.e., variable, function, or class names) and objects. It allows you to organize your code into logical groups and avoid naming conflicts. There are several types of namespaces in Python, including built-in namespaces (containing Python's built-in functions and types), global namespaces (containing names defined at the top-level of a module), local namespaces (containing names defined within a function), and object namespaces (containing attributes and methods of an object).

Q35: What is name mangling in Python?

A: Name mangling is a feature in Python that allows you to prefix attribute names with double underscores (__) to make them "private" and prevent them from being accessed or overwritten outside of the class. When you use name mangling, Python will automatically prepend the class name to the attribute name to create a unique name that cannot be easily accessed or modified from outside the class. For example, an attribute named __foo in a class named Bar would be renamed to _Bar__foo to prevent access from outside of the class.

Q36: What is a garbage collector in Python?

A: A garbage collector in Python is a program that automatically frees memory that is no longer being used by a program. In Python, memory management is handled automatically by the interpreter, which uses a reference counting system to keep track of when objects are no longer being used. When an object's reference count drops to zero, the garbage collector is responsible for freeing the memory associated with that object. The garbage collector in Python is designed to be transparent and does not require any special action on the part of the programmer.

Q37: How do you implement multiple inheritance in Python?

A: Multiple inheritance in Python can be implemented by creating a new class that inherits from two or more parent classes. Here's an example:

```
class Parent1:

def method1(self):

print("This is Parent1's method1")

class Parent2:

def method2(self):

print("This is Parent2's method2")

class Child(Parent1, Parent2):

pass

c = Child()

c.method1() # Output: This is Parent1's method1

c.method2() # Output: This is Parent2's method2
```

In this example, the Child class inherits from both Parent1 and Parent2 using the syntax class Child(Parent1, Parent2):. The pass keyword is used to indicate that the Child class does not have any additional methods or attributes beyond what is inherited from the parent classes. Once the Child class is defined, you can create an instance of it and call methods from both parent classes as shown in the example.

Q38: What is a metaclass in Python?

A: In Python, a metaclass is a class that defines the behavior of other classes. It is responsible for creating and initializing new class objects. Metaclasses are often used to add custom behavior to classes, such as adding class-level attributes or methods, modifying class inheritance, or enforcing certain constraints on the class. In Python, the type function is itself a metaclass that is used to create new class objects.

Q39: How do you create a metaclass in Python?

A: To create a metaclass in Python, you define a new class that inherits from the type class. This new class can then be used as the metaclass for other classes. Here's an example:

python

Copy code

```
class MyMeta(type):

def __new__(cls, name, bases, attrs):

# Modify the attributes of the new class here

attrs['custom_attr'] = 'Hello, world!'

return super().__new__(cls, name, bases, attrs)

class MyClass(metaclass=MyMeta):

pass

print(MyClass.custom_attr) # Output: Hello, world!
```

In this example, the MyMeta class inherits from type and overrides the __new__ method to modify the attributes of the new class being created. The MyClass class is defined with the metaclass argument set to MyMeta, which means that the MyMeta class will be used as the metaclass for MyClass. When MyClass is created, the __new__ method of MyMeta is called, which adds a new attribute to MyClass called custom_attr.

Q40: What is the use of slots in Python?

A: In Python, the __slots__ attribute is used to define a fixed set of attributes for a class. This can improve performance and reduce memory usage, particularly when creating many instances of a class. By default, Python uses a dictionary to store attributes for each object, which can be slow and memory-intensive. Using __slots__ instead allows Python to use a more efficient data structure to store the attributes. However, it also means that you cannot dynamically add new attributes to instances of the class after they are created.

Q41: What is a data descriptor in Python?

A: In Python, a data descriptor is a descriptor that defines both __get__ and __set__ methods. It is used to control access to a class attribute by intercepting calls to the attribute's getter and setter methods. This allows you to define custom behavior when getting or setting the attribute, such as validating inputs or triggering additional actions.

Q42: What is a non-data descriptor in Python?

A: In Python, a non-data descriptor is a descriptor that defines only the __get__ method. It is used to control access to a class attribute by intercepting calls to the attribute's getter method. Unlike data descriptors, non-data descriptors cannot be used to set the attribute value.

Q43: What is the difference between a data descriptor and a non-data descriptor in Python?

A: The main difference between data descriptors and non-data descriptors in Python is that data descriptors define both __get__ and __set__ methods, while non-data descriptors define only the __get__ method. This means that data descriptors can be used to control both getting and setting of a class attribute, while non-data descriptors can only control getting of the attribute. Additionally, when an attribute is both a data descriptor and a non-data descriptor, the data descriptor takes precedence and its __get__ and __set__ methods are used.

Aspect	**Data descriptor**	**Non-data descriptor**
Required methods	__get__, __set__	__get__
Access control	Can control both getting and setting of an attribute	Can only control getting of an attribute
Attribute priority	Takes precedence over non-data descriptors	Non-data descriptor is used if data descriptor not found
Examples	Properties, methods with decorators, descriptor classes	Methods with decorators, class-level constants

Q44: How do you use property() function in Python?

A: The property() function in Python is used to define getter, setter, and deleter methods for an attribute. It allows you to access an attribute like a normal variable, but behind the scenes, the attribute is managed by the getter, setter, and deleter methods. To use the property() function, you need to define the getter, setter, and deleter methods, and then create a property object that links these methods to the attribute. Here's an example:

```
class MyClass:

def __init__(self, value):

self._value = value

def get_value(self):

return self._value

def set_value(self, value):
```

```
self._value = value

def del_value(self):

del self._value

value = property(get_value, set_value, del_value)
```

In this example, the get_value(), set_value(), and del_value() methods define the behavior of the 'value' attribute. The property() function is used to create a property object that links these methods to the 'value' attribute.

Q45: What is the use of getattr(), setattr(), and delattr() functions in Python?

A: The getattr(), setattr(), and delattr() functions in Python are used to get, set, and delete attributes on an object, respectively. These functions are often used to work with dynamic attributes, where the name of the attribute is not known until runtime. Here's an example of how to use these functions:

```
class MyClass:

pass

obj = MyClass()

setattr(obj, 'attr1', 123) # set the 'attr1' attribute to 123

value = getattr(obj, 'attr1') # get the value of the 'attr1' attribute

delattr(obj, 'attr1') # delete the 'attr1' attribute
```

In this example, the setattr() function sets the value of the 'attr1' attribute on the MyClass object, the getattr() function gets the value of the 'attr1' attribute, and the delattr() function deletes the 'attr1' attribute.

Q46: What is a property decorator in Python?

A: A property decorator in Python is a special type of decorator that allows you to define getter, setter, and deleter methods for a property. It provides a simpler way to define properties compared to using the property() function. To define a property using a decorator, you use the '@property' decorator for the getter method, and the '@property_name.setter' decorator for the setter method. Here's an example:

```
class MyClass:

def __init__(self, value):

self._value = value
```

```
@property
def value(self):
return self._value
@value.setter
def value(self, new_value):
self._value = new_value
```

In this example, the '@property' decorator is used to define the getter method for the 'value' property, and the '@value.setter' decorator is used to define the setter method. This allows you to access and modify the 'value' property using normal variable syntax, while still allowing you to add custom behaviour to the getter and setter methods.

Q47: What is a class decorator in Python?

A: A class decorator in Python is a special type of decorator that is used to modify the behaviour of a class. It is defined using the '@decorator_name' syntax and is applied to the class definition. Class decorators are often used to add functionality to a class or to modify its behaviour. Here's an example:

```
def my_decorator(cls):
cls.new_attribute = 123
return cls
@my_decorator
class MyClass:
```

Q48: What is a class decorator in Python?

A class decorator in Python is a function that takes a class and modifies or enhances its behaviour. It is used to add functionality to a class without modifying the class's source code directly. A class decorator is called using the '@' symbol followed by the name of the decorator function, and it is placed immediately before the class definition.

Here is an example of a class decorator that adds a method to a class:

```
def add_method(cls):
def new_method(self):
print("Hello World!")
```

```
cls.my_method = new_method

return cls

@add_method

class MyClass:

pass

obj = MyClass()

obj.my_method() # prints "Hello World!"
```

In this example, the add_method decorator takes the MyClass definition, creates a new method called my_method that prints "Hello World!", and then adds it to the class. When the my_method is called on an instance of MyClass, it will print "Hello World!".

Q49: What is a type hint in Python?

Type hints in Python are a way to specify the expected type of function arguments, variables, and return values. They are optional, but they can help improve code readability, and maintainability, and catch errors before the code is executed. Type hints are annotations that are added to the code using a special syntax that indicates the expected type of the annotated object.

Here is an example of type hinting in Python:

```
def add_numbers(a: int, b: int) -> int:

return a + b
```

In this example, the add_numbers function takes two integer arguments (a and b) and returns an integer value. The types of the arguments and the return value are specified using type hints. The colon followed by the type name is used to specify the type of the variable or argument, and the "->" symbol is used to specify the return type.

Q50: How do you use type hints in Python?

To use type hints in Python, you need to add the type annotations to the relevant objects. There are several ways to do this:

Function arguments: Add the type hint after the argument name, separated by a colon. For example: def add_numbers(a: int, b: int) -> int:

Function return values: Add the type hint after the closing parenthesis of the function arguments, separated by an arrow (->). For example: def add_numbers(a: int, b: int) -> int:

Variables: Add the type hint after the variable name, separated by a colon. For example: x: int = 42

Class attributes: Add the type hint as a class-level variable with the expected type. For example: class MyClass: my_attribute: int = 42

Type hints are optional, and Python will not enforce them at runtime. They are intended to be used as a form of documentation for developers and as a way to catch errors during development. Type hints can be checked using external tools or the mypy library, which is a static type checker for Python.