

# Object Oriented Basics

Yes, all the object-oriented programming (OOP) terms and concepts mentioned apply to Python. Python is a multi-paradigm programming language that fully supports object-oriented programming. In fact, OOP is one of the primary programming paradigms used in Python, and its syntax and features are designed to facilitate object-oriented design and development.

Here's how these OOP terms apply to Python specifically:

1. **Class:** In Python, classes are defined using the `class` keyword, and they encapsulate data (attributes) and behavior (methods). Objects are created by instantiating classes using the class name followed by parentheses, optionally passing arguments to the constructor (`__init__` method).
2. **Object:** Objects in Python are instances of classes. They have attributes (instance variables) and methods (functions defined within the class). Objects are created dynamically at runtime and can be assigned to variables, passed as arguments, and returned from functions.
3. **Attribute:** Attributes in Python are data items associated with objects. They can be accessed using dot notation (`object.attribute`) or through methods defined within the class. Attributes can be public, private (using name mangling), or protected (using single underscore convention).
4. **Method:** Methods in Python are functions defined within a class that operate on its instances. They are defined using the `def` keyword within the class definition. Methods can access and modify the object's state (attributes) and behavior.
5. **Inheritance:** Python supports single and multiple inheritance, allowing classes to inherit attributes and methods from one or more parent classes. Inheritance relationships are defined using parentheses after the class name in the class definition.
6. **Encapsulation:** Python supports encapsulation through the use of classes, which bundle data and methods together. Python does not have built-in support for access modifiers like private or protected, but encapsulation can be achieved through conventions and name mangling.
7. **Polymorphism:** Python supports polymorphism through method overriding and duck typing. Method overriding allows subclasses to provide their own implementation of methods inherited from parent classes, while duck typing allows objects to be treated uniformly based on their behavior rather than their type.
8. **Abstraction:** Python supports abstraction through classes and interfaces, allowing programmers to model real-world entities and concepts while hiding implementation details. Python encourages writing code that operates on interfaces rather than concrete implementations.
9. **Constructor and Destructor:** In Python, the constructor is defined using the `__init__` method, which is automatically called when an object is instantiated. Python does not have explicit destructors, but the `__del__` method can be used to define cleanup tasks.

when an object is garbage-collected.

10. **Instance, Class, and Instance Variables:** Python distinguishes between instance variables (attributes specific to individual objects) and class variables (attributes shared among all instances of a class). Instance variables are defined within methods using the `self` parameter, while class variables are defined outside methods within the class.
11. **Method Overriding and Overloading:** Python supports method overriding by allowing subclasses to provide their own implementations of methods inherited from parent classes. However, Python does not support method overloading in the traditional sense (having multiple methods with the same name but different signatures), but you can achieve similar functionality using default parameter values or variable-length argument lists.

Overall, Python's support for object-oriented programming makes it a versatile and powerful language for designing and implementing software systems using OOP principles.

In Python, the `self` keyword is used within methods of a class to refer to the instance of the class itself. It is passed implicitly as the first argument to instance methods, allowing those methods to access and modify attributes of the instance. Here's a guideline on when to use `self` and when not to:

## Use `self`:

1. **Inside Instance Methods:** Within instance methods, use `self` to access instance attributes and methods.

```
class MyClass:
    def __init__(self, value):
        self.value = value

    def print_value(self):
        print(self.value)

obj = MyClass(10)
obj.print_value() # Output: 10
```

**When Assigning Instance Attributes:** Use `self` to assign instance attributes within the `__init__` method or other instance methods.

```
class MyClass:
    def __init__(self, value):
```

```
self.value = value
```

```
obj = MyClass(10)
```

## Do Not Use `self`:

**Outside Class Definition:** When accessing class attributes or methods outside the class definition, you do not need to use `self`.

```
class MyClass:
    class_attribute = 100

    def __init__(self, value):
        self.instance_attribute = value

obj = MyClass(10)
print(obj.instance_attribute) # Output: 10
print(MyClass.class_attribute) # Output: 100
```

**Static Methods and Class Methods:** In static methods and class methods, `self` is not used because these methods are not bound to a specific instance.

```
class MyClass:
    @staticmethod
    def static_method():
        print("Static method")

    @classmethod
    def class_method(cls):
        print("Class method")

MyClass.static_method() # Output: Static method
MyClass.class_method() # Output: Class method
```

Remember that `self` is a convention in Python, and you could technically name the first parameter of an instance method anything you like, but using `self` is highly recommended for readability and consistency with Python conventions.

Revision #5

Created 26 February 2024 01:22:40 by victor

Updated 28 February 2024 00:46:53 by victor