**Object Oriented Programming**

lunes, 13 de noviembre de 2023

12:35 p. m.

OOP

Real Python ref: <https://realpython.com/python3-object-oriented-programming/>

geeks for geeks ref: <https://www.geeksforgeeks.org/python-oops-concepts/>

Patrick Loeber Ref: [Object Oriented Programming (OOP) In Python - Beginner Crash Course](https://www.youtube.com/watch?v=-pEs-Bss8Wc)

freeCodeCamp Ref: [Object Oriented Programming with Python - Full Course for Beginners](https://www.youtube.com/watch?v=Ej_02ICOIgs)

Corey Schafer Ref: [Python OOP Tutorials - Working with Classes](https://www.youtube.com/playlist?list=PL-osiE80TeTsqhIuOqKhwlXsIBIdSeYtc)

Introduction

Real Python ref: <https://realpython.com/python3-object-oriented-programming/>

Object-oriented programming is a programming paradigm that provides a means of structuring programs so that properties and behaviors are bundled into individual **objects**.

For example, an object could represent a person with **properties** like a name, age, and address and **behaviors** such as walking, talking, breathing, and running. Or it could represent an email with properties like a recipient list, subject, and body and behaviors like adding attachments and sending.

Put another way, object-oriented programming is an approach for modeling concrete, real-world things, like cars, as well as relations between things, like companies and employees or students and teachers. OOP models real-world entities as software objects that have some data associated with them and can perform certain operations.

Class methods vs Instance methods

pending…

There are some main concepts from OOP: Objects, Encapsulation, Polymorphism, Inheritance, Abstraction and Classes

Diagrama

Descripción generada automáticamente

Classes

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

Python classes are pretty cool and powerful tools that you can use in multiple scenarios. Because of this, some people tend to overuse classes and solve all their coding problems using them. However, sometimes using a class isn’t the best solution. Sometimes a couple of functions are enough.

In practice, you’ll encounter a few situations in which you should avoid classes. For example, you shouldn’t use regular classes when you need to:

* Store only data. Use a data class or a named tuple instead.
* Provide a single method. Use a function instead.

Data classes, enumerations, and named tuples are specially designed to store data. So, they might be the best solution if your class doesn’t have any behavior attached.

If your class has a single method in its API, then you may not require a class. Instead, use a function unless you need to retain a certain state between calls. If more methods appear later, then you can always create a class.

Classmethods vs Staticmethods

The difference between @classmethod and @staticmethod lies in how they interact with the class and instance within a class.

1. @classmethod:

* Receives the class itself as the first argument (cls).
* Allows access to class variables and methods via the cls argument.
* Typically used when a method needs to work with class-level variables or perform operations that are related to the class itself.
* It's useful when you need a method to interact with the class but don't necessarily need to access or modify instance-specific data.

class MyClass:

    class\_variable = 10

    @classmethod

    def class\_method(cls, x, y):

        print(f"Class variable: {cls.class\_variable}")

        print(f"Arguments: {x}, {y}")

1. @staticmethod:

* Does not receive an implicit reference to the class or instance (no cls or self argument).
* Behaves like a regular function within the class scope.
* Cannot access class or instance variables directly.
* Often used when a method doesn't rely on class or instance-specific data and operates more as a utility function that's related to the class but doesn't need access to its internals.
* class MyClass:

    @staticmethod

    def static\_method(x, y):

        print(f"Arguments: {x}, {y}")

In summary, @classmethod is used when you want a method to operate on the class itself or class-specific variables, while @staticmethod is used when a method is relevant to the class but doesn't rely on class or instance-specific data. Both decorators have their uses depending on the context and requirements of the class design.

Objects

The object is an entity that has a state and behavior associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects. More specifically, any single integer or any single string is an object. The number 12 is an object, the string “Hello, world” is an object, a list is an object that can hold other objects, and so on. You’ve been using objects all along and may not even realize it.

An object consists of:

* State: It is represented by the attributes of an object. It also reflects the properties of an object.
* Behavior: It is represented by the methods of an object. It also reflects the response of an object to other objects.
* Identity: It gives a unique name to an object and enables one object to interact with other objects.

Polymorphism

*"changing base classes behavior on derived ones"*

Polymorphism simply means having many forms, and for this context, this happens when a child class overrides a method of the parent.

class Animal:

    def intro(self):

        return print('Hi, I'm an animal')

    def speak(self):

        return print('Generic animal sound')

class Dog(Animal):

    def speak(self):

        return print('Woof!')

class Cat(Animal):

    def speak(self):

        return print('Meow!')

doggo = Dog()

catto = Cat()

doggo.intro() # Hi, I'm an animal

doggo.speak() # Woof!

print()

catto.intro() # Hi, I'm an animal

catto.speak() # Meow!

Is easy to be mistaken as inheritance, since those are pretty similar. The difference that the concept of inheritance is that the child class preserves the attributes and method of the parent (not necessarily overriding them).

class Animal:

    def speak(self):

        return print('Generic animal sound')

class Dog(Animal):

    def bark(self):

        return print('Woof!')

class Cat(Animal):

    def meow(self):

        return print('Meow!')

doggo = Dog()

catto = Cat()

doggo.speak() # Generic animal sound

catto.speak() # Generic animal sound

print()

doggo.bark() # Woof!

catto.meow() # Meow!

Inheritance

*"Allowing derived classes to access the Base class methods and attributes"*

Inheritance allow **is-a** type of relationship between the base and derived classes.

The Super() function

This function is used in the context of heritance and mostly used in the initialization of the instance. It allows to have a temporary object of the superclass (or the parent) to be able to call its methods.

The primary use case for super() is within the \_\_init\_\_ method of a subclass to ensure that the initialization of the parent class is executed before the subclass's initialization. This is important for maintaining the correct order of initialization in the inheritance hierarchy.

Using super() ensures that the parent class is properly initialized before the child class, and it allows you to access and use methods from the parent class within the child class.

class Parent:

    speaks = ["English"]

class Child(Parent):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.speaks.append("German")

jake = Child()

print(jake.speaks)  # ['English', 'German']

Inheritance Pros: It allows for code reuse, promotes a hierarchical structure, and supports polymorphism, enabling objects to be treated as instances of their parent class. This can reduce redundant code and provide a clear, organized structure.

Inheritance Cons: Overuse of inheritance can lead to a complex hierarchy that's hard to maintain, known as the "diamond problem." Changes in the base class can impact all derived classes, potentially causing unforeseen issues. It also tightly couples classes, making them more dependent on each other.

Alternatives to Inheritance

Inheritance, and especially multiple inheritance, can be a complex and hard-to-grasp topic. Fortunately, inheritance isn’t the only technique that allows you to reuse functionality in object-oriented programming. You also have composition, which represents a has-a relationship between classes.

Composition allows you to build an object from its components. The composite object doesn’t have direct access to each component’s interface. However, it can leverage each component’s implementation.

Delegation is another technique that you can use to promote code reuse in your OOP programs. With delegation, you can represent can-do relationships, where an object relies on another object to perform a given task.

Composition

As you already know, you can use composition to model a **has-a** relationship between objects. In other words, through composition, you can create complex objects by combining objects that will work as components. Note that these components may not make sense as stand-alone classes.

Favoring composition over inheritance leads to more flexible class designs. Unlike inheritance, composition is defined at runtime, which means that you can dynamically replace a current component with another component of the same type. This characteristic makes it possible to change the composite’s behavior at runtime.

In the python file will be an example of Composition: Creating an IndustrialRobot class from Arm and Body classes.

An idea to make this robot even cooler is to implement several types of arms with different welding technologies. Then you can change the arm by running robot.arm = NewArm(). You can even add a .change\_arm() method to your robot class. How does that sound as a learning exercise?

Unlike inheritance, composition doesn’t expose the entire interface of components, so it preserves encapsulation. Instead, the composite objects access and use only the required functionality from their components. This characteristic makes your class design more robust and reliable because it won’t expose unneeded members.

Following the robot example, say you have several different robots in a factory. Each robot can have different capabilities like welding, cutting, shaping, polishing, and so on. You also have several independent arms. Some of them can perform all those actions. Some of them can perform just a subset of the actions.

Now say that a given robot can only weld. However, this robot can use different arms with different welding technologies. If you use inheritance, then the robot will have access to other operations like cutting and shaping, which can cause an accident or breakdown.

If you use composition, then the welder robot will only have access to the arm’s welding feature. That said, composition can help you protect your classes from unintended use.

Comp. Pros: Offers more flexibility by allowing objects to be composed of other objects, promoting a "has-a" relationship rather than an "is-a" relationship. This results in looser coupling, making the code more modular and easier to maintain. It enables greater flexibility in swapping out components.

Comp. Cons: Can lead to more boilerplate code depending on the complexity of the relationships between objects. Sometimes, it might be less intuitive than inheritance, especially in simpler cases.

Delegation

Delegation is another technique that you can use as an alternative to inheritance. With delegation, you can model **can-do** relationships, where an object hands a task over to another object, which takes care of executing the task. Note that the delegated object can exist independently from the delegator.

You can use delegation to achieve code reuse, separation of concerns, and modularity. For example, say that you want to create a stack data structure. You think of taking advantage of Python’s list as a quick way to store and manipulate the underlying data.

In the python file will be an example of Delegation: A Stack class delegating method to list built-in.

Note how these operations conveniently delegate their responsibilities on .\_items.append() and .\_items.pop(), respectively. Your Stack class has handed its operations over to the list object, which already knows how to perform them.

It’s important to notice that this class is pretty flexible. You can replace the list object in .\_items with any other object as long as it implements the .pop() and .append() methods. For example, you can use a deque object from the collections module.

Because you’ve used delegation to write your class, the internal implementation of list isn’t visible or directly accessible in Stack, which preserves encapsulation:

class Stack:

    def \_\_init\_\_(self, items=None ):

        if items is None:

            self.\_items = []

        else:

            self.\_items = list(items)

    def push(self, item):

        self.\_items.append(item)

    def pop(self, item):

        return self.\_items.pop()

    def \_\_repr\_\_(self):

        return f"{type(self).\_\_name\_\_}({self.\_items})"

stack = Stack([1,2,3,4])

print(stack)    # Stack([1, 2, 3, 4])

print(dir(stack))

# [

# …

# '\_items',

#   'pop',

#   'push'

# ]

The public interface of your Stack class only contains the stack-related methods .pop() and .push(), as you can see in the dir() function’s output.

This prevents the users of your class from using list-specific methods that aren’t compatible with the classic stack data structure.

If you use inheritance, then your child class, Stack, will inherit all the functionality from its parent class, list:

class Stack(list):

    def push(self,item):

        self.append(item)

    def pop(self):

        return super().pop

    def \_\_repr\_\_(self):

        return f"{type(self).\_\_name\_\_}({super().\_\_repr\_\_()})"

stack = Stack([1,2,3,4])

print(stack)    # Stack([1, 2, 3, 4])

print(dir(stack))

# [

#     …

#     'append',

#     'clear',

#     'copy',

#     'count',

#     'extend',

#     'index',

#     'insert',

#     'pop',

#     'push',

#     'remove',

#     'reverse',

#     'sort'

# ]

Delegation Pros: Composition offers greater flexibility as it allows objects to be composed of other objects, forming a "has-a" relationship. This enables more dynamic and adaptable code structures. It promotes modularity and code reuse by separating concerns into distinct objects, making the codebase easier to maintain and extend. Objects created through composition are loosely coupled, reducing dependencies between classes and making the code more maintainable and testable. Modifications or updates in the behavior of individual components can be made without affecting the entire structure, enhancing the code's adaptability.

Delegation Cons: Depending on the complexity of relationships between objects, implementing composition can sometimes involve writing additional code to manage these relationships, leading to increased boilerplate code. In intricate systems, managing the interactions between different components might become complex, requiring careful design and maintenance. Understanding and implementing composition might be a bit more challenging for beginners compared to inheritance, as it involves a different way of structuring classes and their relationships.

Encapsulation

*"The ability to have public and private attributes only accessible thru getters and setters"*

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object’s variable can only be changed by an object’s method. Those types of variables are known as private variables.

Encapsulation is a fundamental concept in Object-Oriented Programming (OOP) that involves bundling the data (attributes) and the methods (functions or procedures that operate on the data) into a single unit known as a class.

It's like putting your data and the operations that manipulate that data into a protective capsule, allowing controlled access to the inner workings of an object while hiding the implementation details from the outside.

The key idea behind encapsulation is to create a boundary around an object's data to prevent direct access from outside the class and instead provide controlled access through methods (often referred to as getters and setters). This helps in maintaining the integrity of the data and allows for better control over how the data is modified or accessed.

By encapsulating data within methods, you can enforce certain rules or validations before allowing any changes to the object's state. This promotes a more organized, modular, and secure way of building software, as it reduces the chance of unintended interference with the object's internal state.

In summary, encapsulation in OOP:

1. Bundles data (attributes) and methods (functions) into a single unit (class).
2. Controls access to data by providing methods (getters and setters).
3. Hides the implementation details, focusing on the interface or public methods to interact with the object.
4. Enhances data integrity and security by enforcing rules and validations.

Abstraction

It hides unnecessary code details from the user. Also, when we do not want to give out sensitive parts of our code implementation and this is where data abstraction came.

Abstraction is another key concept in object-oriented programming (OOP) that involves simplifying complex systems by modeling classes based on the essential features or behavior, while hiding the unnecessary details.

At its core, abstraction focuses on emphasizing the essential characteristics while suppressing or hiding the irrelevant or non-essential details. It allows you to create a model or representation of an entity that includes only the most relevant attributes and methods needed for a particular purpose. This simplification and focusing on the important aspects make the system more manageable and understandable.

There are two primary aspects of abstraction:

Data Abstraction: This involves hiding the complex implementation details of data and only displaying the necessary and essential features. For instance, a class might represent a "Car" with attributes such as make, model, and year without exposing the intricate engineering details.

Procedural Abstraction: This focuses on hiding the complex implementation of methods or procedures and providing a simpler interface for users. Users interact with the methods using a defined interface without needing to know how those methods are implemented internally. For example, you might have a method start\_engine() in a car class. Users don't need to know the technicalities of how the engine starts; they just need to call the method.

Abstraction is achieved in OOP through the use of abstract classes and interfaces, which provide a blueprint for other classes to follow. Abstract classes can have abstract methods that subclasses must implement, ensuring that essential functionality is present in all derived classes.

The benefits of abstraction include:

1. Managing Complexity: By hiding unnecessary details, abstraction simplifies understanding and managing complex systems.

1. Enhancing Reusability: Abstract classes and interfaces promote code reuse by allowing common functionality to be defined once and shared among multiple subclasses.

1. Facilitating Maintenance: Changes to the underlying implementation can be made without affecting the code that uses the abstraction, as long as the interface remains consistent.

Overall, abstraction in OOP is about creating a clear and simplified model of entities or processes by focusing on what's essential and hiding unnecessary complexity, leading to more manageable and maintainable code.

To create abstract classes the built-in module abc provides tools to do it so. Notes from the use of this module:

* An Abstract Class is uninstanseable by definition, so if a class inherits ABC from abc, it won't allow it to be instantiated by anyone.

* If a method is an Abstract Method, it means that it needs to be implemented on the concrete class derived from the abstract otherwise it'll throw a TypeError.