A Beginner's Guide to Class Construction in Python

Satya Komatineni, 2/10/24 (version 2)

Contents

[Simple Class Construction and Instance Variables 1](#_Toc158467613)

[Encapsulation with Private and Public Methods 1](#_Toc158467614)

[Importance of "self" 2](#_Toc158467615)

[Inheritance and Method Overriding 2](#_Toc158467616)

[Abstract Classes and Method Implementation 2](#_Toc158467617)

[Class Variables and Initialization 3](#_Toc158467618)

[Properties for Access Control 3](#_Toc158467619)

Python has Object-Oriented Programming (OOP). For those familiar with OOP in languages like Java or C++, Python's syntax and dynamic nature present a slightly different paradigm. This guide starts with foundational concepts and progresses to more slightly complex aspects of class design in Python, incorporating direct code examples for clarity.

# Simple Class Construction and Instance Variables

In Python, defining a class and initializing instance variables is straightforward. The \_\_init\_\_ method (Python's constructor) is used to initialize an object's state, with instance variables declared within:

1. class MyBaseClass:

2. name: str

3. age: int

4.

5. def \_\_init\_\_(self, name: str, age: int):

6. self.name = name

7. self.age = age

This snippet demonstrates defining instance variables (name and age) with type annotations, which are hints to the type of variable but are not enforced by Python at runtime. Type annotations help with readability and can be leveraged by tools for type checking.

# Encapsulation with Private and Public Methods

Python uses a naming convention to denote private members: a prefix underscore (\_). While not enforced by the Python runtime, this convention signals to developers that a method or variable is intended for internal use only:

1. # Private method (starts with \_)

2. def \_private\_method(self):

3. print("This is a private method")

4.

5. # Public method

6. def public\_method(self):

7. print("This is a public method")

# Importance of "self"

1. When calling another private method you have to use ***self.\_the\_other\_method()***
2. You don't have to pass the self as an argument.
3. Self is automatically passed as the first argument.
4. While defining a method inside a class **"self" MUST be the first argument**.
5. The type-checker or the editor usually tells you this while you are coding.

# Inheritance and Method Overriding

Python supports class inheritance, allowing a class to inherit attributes and methods from another class. Overriding methods in the subclass allows for extending or changing the behavior of the base class methods:

1. class MyDerivedClass(MyBaseClass):

2. gender: str

3.

4. def \_\_init\_\_(self, name: str, age: int, gender: str):

5. super().\_\_init\_\_(name, age) # Call to the super class \_\_init\_\_ method

6. self.gender = gender

7.

8. # Overriding the private method

9. def \_private\_method(self):

10. super().\_private\_method()

11. print("This is a private method in MyDerivedClass")

12.

13. # Overriding the public method

14. def public\_method(self):

15. super().public\_method()

16. print("This is a public method in MyDerivedClass")

17.

Same named methods in derived classes override the base class methods.

# Abstract Classes and Method Implementation

Moving towards more advanced concepts, abstract classes in Python are defined using the ABC module. Abstract classes cannot be instantiated and require subclasses to provide implementations for abstract methods. You get a type error if a derived base class does not implement an abstract method.

1. from abc import ABC, abstractmethod

2.

3. class MyBaseClass(ABC):

4. @abstractmethod

5. def my\_abstract\_method(self):

6. pass

7.

8. class MySubclass(MyBaseClass):

9. def my\_abstract\_method(self):

10. print("Implemented abstract method in MySubclass")

11.

# Class Variables and Initialization

Class variables are shared across all instances of a class. Python allows the initialization of class variables in various ways, including using decorators:

1. def initialize\_class\_variables(cls):

2. if not hasattr(cls, 'class\_variable'):

3. cls.class\_variable = 10

4. return cls

5.

6. @initialize\_class\_variables

7. class MyClass:

8. class\_variable = None

9.

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

11. pass

12.

13. # Example usage

14. obj = MyClass()

15. print(obj.class\_variable) # Output: 10

This code snippet demonstrates a pattern to ensure that class variables are initialized in a controlled manner, leveraging Python's dynamic capabilities.

Some essentials of class variables

1. class variables declared as class\_variable\_name: type.
2. They are initialized with a function using an annotation.
3. This is ensured by python imports where it is called only once during the import.
4. Python doesn't have java like static block that is guaranteed to be executed only once in a multi-threaded environment.

# Properties for Access Control

Python's property decorators provide a Pythonic way to implement getters, setters, and deleters, allowing for controlled access to an object's attributes:

1. class MyClass:

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

3. self.\_name = name

4.

5. @property

6. def name(self):

7. return self.\_name

8.

9. @name.setter

10. def name(self, value):

11. self.\_name = value

12.

13. @name.deleter

14. def name(self):

15. del self.\_name

Properties encapsulate the internal representation of an attribute, offering an interface for getting, setting, or deleting it, akin to getters and setters in other languages but with a more intuitive syntax.

Notice how you are invoking these methods.

1. obj.name #call the getter

2. obj.name = "blah" #call the setter

3. del obj.name #call the deleter

Notice how annotations differ for each of these methods above.