A Beginner's Guide to Class Construction in Python

Satya Komatineni, 2/17/24 (version 4) [started 2/1/24]

Location: <https://github.com/SatyaKomatineni/articles-repo/blob/master/classes-in-python-a-beginers-guide.docx>

Audience: Fairly experienced Object-oriented programmer in any other language.

Absorption time: 10 pages and about 2 leisurely hours.

Contents

[Abstract 2](#_Toc159052178)

[Simple Class Construction and Instance Variables 2](#_Toc159052179)

[Encapsulation with Private and Public Methods 2](#_Toc159052180)

[Importance of "self" 3](#_Toc159052181)

[Inheritance and Method Overriding 3](#_Toc159052182)

[Abstract Classes and Method Implementation 3](#_Toc159052183)

[Multiple Inheritance with a Surprise 4](#_Toc159052184)

[Less important for most 4](#_Toc159052185)

[Class Variables and Initialization 5](#_Toc159052186)

[Properties for Access Control 5](#_Toc159052187)

[Static methods Example 6](#_Toc159052188)

[\_\_Call\_\_ method: Objects as functions 7](#_Toc159052189)

[The lowdown on underscores 7](#_Toc159052190)

[A single underscore \_ : 7](#_Toc159052191)

[Double underscores \_\_ : 7](#_Toc159052192)

[Dunder Methods 8](#_Toc159052193)

[Creating singletons and \_\_new\_\_ 8](#_Toc159052194)

[2 Mode Dunder methods: repr(), and str() 9](#_Toc159052195)

[The nature of repr() 9](#_Toc159052196)

[The nature of str() 9](#_Toc159052197)

[Context Managers 9](#_Toc159052198)

# Abstract

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 slightly advanced aspects of class design in Python, incorporating direct code examples to 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) # super 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")

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.

# Multiple Inheritance with a Surprise

Python allows duplicate methods, although I suspect you want to avoid. See example below:

1. class Base1:

2. def method(self):

3. print("Method of Base1")

4.

5. class Base2:

6. def method(self):

7. print("Method of Base2")

8.

9. class Derived(Base1, Base2):

10. def another\_method(self):

11. print("Another method of Derived")

12.

13. # Creating an instance of Derived

14. d = Derived()

15.

16. # Calling methods: surprise

17. d.method()

18. # This will call the method from Base1,

19. due to the order in the inheritance list

20.

21. d.another\_method()

22. # Calls method defined in Derived

### Less important for most

Python uses a specific rule, known as the C3 linearization, to determine the method resolution order when dealing with multiple inheritance.

This can be inspected using the .\_\_mro\_\_ attribute or the mro() method on the class.

It's a concept to understand when working with multiple inheritance to predict which method will be called when multiple base classes define methods with the same name.

Author’s note: For sanity, I can’t imagine I would want to go there. 😊

# 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.

# Static methods Example

1. These methods annotated as such are at a class level.
2. @staticmethod annotation
3. They don’t have “self” as an argument.

Example

1. class MathOperations:

2. @staticmethod

3. def add(x, y):

4. return x + y

5.

6. # Using the static method

7. result = MathOperations.add(5, 3)

8. print(result) # Output: 8

# \_\_Call\_\_ method: Objects as functions

1. An object of class with a \_\_call\_\_ method will have like a function.
2. It can have arguments.

Example

1. class Multiplier:

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

3. self.factor = factor

4.

5. def \_\_call\_\_(self, x):

6. return self.factor \* x

7.

8. # Create an instance

9. doubler = Multiplier(2)

10.

11. # Use the instance as if it were a function

12. print(doubler(5)) # Output: 10

13. print(doubler(10)) # Output: 20

# The lowdown on underscores

You will see “\_” and “\_\_” often used in python.

Here is a quick summary of some of the known places:

### A single underscore \_ :

1. Is used for ignoring values, as in assigning from a tuple (look up docs)
2. Indicating private or internal variables, or by convention to signify that a name is meant for internal use within modules and classes.

### Double underscores \_\_ :

1. At the beginning of a name (but not at the end) enable name mangling, which helps to make an attribute or method private to its class.
2. **Dunder methods:** Double underscores at both the beginning and end of a name (dunder methods) are reserved by Python for special methods, which allow your objects to implement, override, or interact with Python's built-in behavior.
3. **Example of dunder methods are:**
   1. **\_\_Init\_\_**
   2. \_\_call\_\_
   3. \_\_str\_\_

# Dunder Methods

There is a lot of magic wrapped around these built-in over-ridable methods in python.

These methods address the following categories of needs:

1. Initialization and Construction
2. Representation Methods
3. Comparison Methods
4. Arithmetic and Bitwise Operators
5. Container Methods
6. Attribute Access
7. Callable Objects
8. Context Managers
9. Iterable Objects
10. Numeric Conversion
11. Hashing and Boolean Test
12. Descriptor Protocol
13. Copying

Note: Each of these are interesting concepts from the idea of classes in any language. So by the time you are done with Python you will run into them at some point. Knowing about these early you might have better constructed your code.

# Creating singletons and \_\_new\_\_

Here is an example with the \_\_new\_\_ dunder method. Note, I haven’t looked into the threading aspects of this. Do that research as you might need it.

1. class Singleton:

2. \_instance = None # Keep instance reference

3.

4. def \_\_new\_\_(cls, \*args, \*\*kwargs):

5. if cls.\_instance is None:

6. cls.\_instance =

7. super(Singleton, cls).

8. \_\_new\_\_(cls, \*args, \*\*kwargs)

9. return cls.\_instance

10.

11. # Test the Singleton class

12. obj1 = Singleton()

13. obj2 = Singleton()

14.

15. print(obj1 is obj2) # Output: True

# 2 Mode Dunder methods: repr(), and str()

Here is an example:

1. class Product:

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

3. self.name = name

4. self.price = price

5.

6. def \_\_repr\_\_(self):

7. return f"Product({self.name!r}, {self.price!r})"

8.

9. def \_\_str\_\_(self):

10. return f"{self.name} - ${self.price}"

11.

12. # Creating an instance of Product

13. product = Product("Coffee Mug", 12.99)

14.

15. # \_\_repr\_\_ is used when echoing in a console or using repr()

16. print(repr(product)) # Output: Product('Coffee Mug', 12.99)

17.

18. # \_\_str\_\_ is used when printing or using str()

19. print(product) # Output: Coffee Mug - $12.99

## The nature of repr()

The \_\_repr\_\_ method returns a string that could be used to recreate the object. It's a convention to make it look like a valid Python expression that could be used to recreate the object with the same state.

## The nature of str()

Method returns a string that provides a user-friendly description of the object.

# Context Managers

Used for managing resources with the “with” key word.

Here is an example:

1. class ManagedFile:

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

3. self.filename = filename

4.

5. def \_\_enter\_\_(self):

6. self.file = open(self.filename, 'r')

7. return self.file

8.

9. def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

10. if self.file:

11. self.file.close()

12.

13. # Usage

14. with ManagedFile('example.txt') as f:

15. content = f.read()

16. print(content)

Here is accomplishing the same with the interesting “yield” concept of python.

Here this is done with a “function” as opposed to a class.

1. from contextlib import contextmanager

2.

3. @contextmanager

4. def managed\_file(filename):

5. try:

6. f = open(filename, 'r')

7. yield f

8. finally:

9. f.close()

10.

11. # Usage

12. with managed\_file('example.txt') as f:

13. content = f.read()

14. print(content)

15.