**Mastering Advanced OOP Concepts in Python: Advanced Class Features**

[[Raman Bazhanau](https://medium.com/@ramanbazhanau?source=post_page-----a8790fa2b5a2--------------------------------)](https://medium.com/@ramanbazhanau?source=post_page-----a8790fa2b5a2--------------------------------)

[Raman Bazhanau](https://medium.com/@ramanbazhanau?source=post_page-----a8790fa2b5a2--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Fa082bddc70ad&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40ramanbazhanau%2Fmastering-advanced-oop-concepts-in-python-advanced-class-features-a8790fa2b5a2&user=Raman+Bazhanau&userId=a082bddc70ad&source=post_page-a082bddc70ad----a8790fa2b5a2---------------------post_header-----------)

9 min read

·

Dec 15

171

2



Welcome to this deep dive into the world of advanced Object-Oriented Programming (OOP) in Python. Whether you’re an intermediate Python developer looking to sharpen your skills or an experienced coder aiming to solidify your understanding of OOP principles, this article is crafted to guide you through some of the more sophisticated aspects of Python’s OOP capabilities.

**Magic Methods**

Magic methods in Python are special methods that begin and end with double underscores (\_\_method\_\_). They enable operator overloading and allow custom behavior for built-in Python functionality.

**Object Creation and Initialization**

* \_\_new\_\_(cls, ...) : Called to create a new instance of a class. This method is unique as it’s called before \_\_init\_\_ and is responsible for returning a new instance of your class. It's particularly useful in creating immutable types and in metaprogramming.
* \_\_init\_\_(self, ...) : : Called after the instance has been created by \_\_new\_\_, this method initializes the object.

class Example:  
 def \_\_new\_\_(cls):  
 print("Creating Instance")  
 return super(Example, cls).\_\_new\_\_(cls)  
   
 def \_\_init\_\_(self):  
 print("Initializing Instance")  
  
  
# Usage  
ex = Example()  
  
# Output: Creating Instance  
# Initializing Instance

**Object Representation**

* \_\_repr\_\_(self): Should return an unambiguous string representation of the object, ideally one that could be used to recreate the object.
* \_\_str\_\_(self): Returns a user-friendly string representation of the object.

class Person:  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
 def \_\_str\_\_(self):  
 return f"{self.name}, aged {self.age}"  
  
 def \_\_repr\_\_(self):  
 return f"Person('{self.name}', {self.age})"  
  
  
# Usage  
p = Person("Alice", 30)  
print(str(p)) # Output: Alice, aged 30  
print(repr(p)) # Output: Person('Alice', 30)

**Comparison Magic Methods**

* \_\_eq\_\_(self, other): Equality (==).
* \_\_ne\_\_(self, other): Inequality (!=).
* \_\_lt\_\_(self, other): Less than (<).
* \_\_le\_\_(self, other): Less than or equal to (<=).
* \_\_gt\_\_(self, other): Greater than (>).
* \_\_ge\_\_(self, other): Greater than or equal to (>=).

class Book:  
 def \_\_init\_\_(self, title, author):  
 self.title = title  
 self.author = author  
  
 def \_\_eq\_\_(self, other):  
 return self.title == other.title and self.author == other.author  
  
  
# Usage  
book1 = Book("1984", "George Orwell")  
book2 = Book("1984", "George Orwell")  
print(book1 == book2) # Output: True

**Arithmetic and Bitwise Magic Methods**

* \_\_add\_\_(self, other): Addition (+).
* \_\_sub\_\_(self, other): Subtraction (-).
* \_\_mul\_\_(self, other): Multiplication (\*).
* \_\_truediv\_\_(self, other): Division (/).
* \_\_floordiv\_\_(self, other): Floor Division (//).
* \_\_mod\_\_(self, other): Modulus (%).
* \_\_pow\_\_(self, other[, modulo]): Power (\*\*).
* \_\_and\_\_(self, other) : Bitwise AND operator (&)
* \_\_invert\_\_(self): Bitwise inversion.
* \_\_or\_\_(self, other) : OR operator(or)
* \_\_xor\_\_(self, other) : Bitwise OR (XOR) operator(^)

class Vector:  
 def \_\_init\_\_(self, x, y):  
 self.x = x  
 self.y = y  
  
 def \_\_add\_\_(self, other):  
 return Vector(self.x + other.x, self.y + other.y)  
  
  
# Usage  
v1 = Vector(2, 4)  
v2 = Vector(1, 3)  
v3 = v1 + v2  
print(v3.x, v3.y) # Output: 3 7

**Numeric Type Conversion**

* \_\_int\_\_(self) , \_\_float\_\_(self) , \_\_complex\_\_(self) , etc., for type conversion.

**Attribute Access and Descriptor Methods**

* \_\_getattr\_\_(self, name), \_\_setattr\_\_(self, name, value), and \_\_delattr\_\_(self, name): Customize attribute access.
* \_\_getattribute\_\_(self, name): Called unconditionally to implement attribute access.
* \_\_get\_\_(self, instance, owner), \_\_set\_\_(self, instance, value), \_\_delete\_\_(self, instance): Implement descriptor protocol.

class ProtectedAttributes:  
 def \_\_init\_\_(self):  
 self.\_protected = "This is protected"  
  
 def \_\_getattr\_\_(self, name):  
 if name == "secret":  
 raise AttributeError("Access Denied")  
 return self.\_\_dict\_\_.get(name, f"{name} not found")  
  
 def \_\_setattr\_\_(self, name, value):  
 if name == "secret":  
 raise AttributeError("Cannot modify secret")  
 self.\_\_dict\_\_[name] = value  
  
 def \_\_delattr\_\_(self, name):  
 if name == "secret":  
 raise AttributeError("Cannot delete secret")  
 del self.\_\_dict\_\_[name]  
  
  
# Usage  
obj = ProtectedAttributes()  
print(obj.\_protected) # Access allowed  
print(obj.missing) # Outputs "missing not found"  
# obj.secret # Raises AttributeError: Access Denied  
# obj.secret = "New" # Raises AttributeError: Cannot modify secret

**Container Magic Methods**

* \_\_len\_\_(self): Returns the length of the container. Part of the protocol for both sequences and mappings.
* \_\_getitem\_\_(self, key): Defines behavior for accessing an item (obj[key]).
* \_\_setitem\_\_(self, key, value): Assigns a value to an item (obj[key] = value).
* \_\_delitem\_\_(self, key): Deletes an item (del obj[key]).
* \_\_iter\_\_(self): Should return an iterator for the container.
* \_\_contains\_\_(self, item): Checks if the container contains item.

class Library:  
 def \_\_init\_\_(self, books):  
 self.books = books  
  
 def \_\_len\_\_(self):  
 return len(self.books)  
  
 def \_\_getitem\_\_(self, index):  
 return self.books[index]  
  
  
# Usage  
library = Library(["Book1", "Book2", "Book3"])  
print(len(library)) # Output: 3  
print(library[1]) # Output: Book2

**Context Managers**

* \_\_enter\_\_(self) (\_\_aenter\_\_(self)): Called at the beginning of a with block. It sets up the context and optionally returns an object that is bound to the variable after the as keyword in the with statement
* \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb) (\_\_aexit\_\_(self, exc\_type, exc\_val, exc\_tv)): Called after the with block. It handles the teardown of the context, like closing a file or releasing a lock. This method receives three arguments (exc\_type, exc\_val, exc\_tb), which are used to manage exceptions raised within the with block.

class AsyncFileHandler:  
 def \_\_init\_\_(self, filename, mode):  
 self.filename = filename  
 self.mode = mode  
 self.file = None  
  
 async def \_\_aenter\_\_(self):  
 self.file = await aiofiles.open(self.filename, self.mode)  
 return self.file  
  
 async def \_\_aexit\_\_(self, exc\_type, exc\_val, exc\_tb):  
 await self.file.close()  
  
  
# Usage with asyncio and aiofiles  
import asyncio  
import aiofiles  
  
  
async def main():  
 async with AsyncFileHandler('example.txt', 'w') as f:  
 await f.write('Hello, async world!')  
  
  
asyncio.run(main())

**Callable Objects**

* \_\_call\_\_(self, [...]): Allows an instance of a class to be called as a function (def / async def).

import asyncio  
  
  
class AsyncAdder:  
 def \_\_init\_\_(self, value):  
 self.value = value  
  
 async def \_\_call\_\_(self, x):  
 await asyncio.sleep(1) # Simulate an async operation  
 return self.value + x  
  
  
# Usage in an async context  
async def main():  
 add\_ten = AsyncAdder(10)  
 result = await add\_ten(20) # Awaits the \_\_call\_\_ method  
 print(result) # Output: 30  
  
  
asyncio.run(main())

**Unary Operations and More**

* \_\_neg\_\_(self): Negative (-obj).
* \_\_pos\_\_(self): Unary plus (+obj).
* \_\_abs\_\_(self): Absolute value (abs(obj)).

**Using \_\_slots\_\_**

**What are \_\_slots\_\_?**

\_\_slots\_\_ is a special class attribute in Python. By defining \_\_slots\_\_ in a class, you explicitly declare that instances of that class will have a fixed set of attributes. This restricts the dynamic creation of new attributes and can lead to significant memory savings, especially for programs creating many instances of a class.

**How to Use \_\_slots\_\_**

You define \_\_slots\_\_ as an iterable (like a list or tuple) of strings that represent the names of the attributes.

**Example: Using \_\_slots\_\_ in a Class**

class Player:  
 \_\_slots\_\_ = ['name', 'score']  
  
 def \_\_init\_\_(self, name, score):  
 self.name = name  
 self.score = score  
  
  
# Usage  
player1 = Player("Alice", 100)  
player1.name # Accessible  
player1.score # Accessible  
# player1.age = 25 # AttributeError: 'Player' object has no attribute 'age'

**Benefits of Using \_\_slots\_\_**

1. *Memory Efficiency*: By preventing the creation of \_\_dict\_\_ and \_\_weakref\_\_ for each instance, \_\_slots\_\_ can lead to significant memory savings, especially with a large number of instances.
2. *Faster Attribute Access*: Access to slot attributes is faster than accessing attributes stored in a \_\_dict\_\_.

**Considerations and Limitations**

* *Inheritance*: If a class inherits from another class without \_\_slots\_\_, it will still have a \_\_dict\_\_ attribute.
* *Immutability*: \_\_slots\_\_ only defines a fixed set of attributes. You cannot add new attributes to instances of the class (though the existing attributes can be modified if not made read-only).
* *No Default* \_\_weakref\_\_: If you need weak references to your objects, you'll need to include '\_\_weakref\_\_' in \_\_slots\_\_.

**Best Practices**

* Use \_\_slots\_\_ when you are sure about the fixed structure of your objects, and you have a large number of instances where memory savings will be significant.
* Avoid using \_\_slots\_\_ if you need dynamic assignment of new attributes or if the class is meant to be subclassed by unknown users' classes.

By incorporating \_\_slots\_\_ into your classes, you can create more memory-efficient and performant Python applications, especially in scenarios where you're dealing with millions of object instances with a fixed attribute structure.

**Classmethod and Staticmethod Decorator**

Another advanced concept in Python OOP is the use of classmethod and staticmethod decorators. These decorators modify methods in a class to change how they can be called and what data they can access.

**Understanding @classmethod**

The @classmethod decorator is used to create a method that is bound to the class and not the instance of the class. This means it can access the class attributes but not the instance attributes. The first parameter of a class method is usually named cls, which refers to the class itself.

**Benefits:**

* Class methods can be called on the class itself, not just on instances.
* They are commonly used for factory methods, which return an instance of the class.

**Example: Using @classmethod as a Factory**

class Person:  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
 @classmethod  
 def from\_full\_name(cls, full\_name):  
 name = full\_name.split()[0]  
 return cls(name)  
  
  
# Usage  
john = Person.from\_full\_name("John Doe")  
print(john.name) # Output: John

In this example, from\_full\_name is a class method that creates an instance of Person using a full name.

**Understanding @staticmethod**

The @staticmethod decorator is used to create a method that doesn't access instance or class data. Static methods do not receive an implicit first argument (neither self nor cls).

**Benefits:**

* Static methods can be called on the class itself, similar to class methods.
* Useful for utility functions that perform a task in isolation.

**Example: Using @staticmethod**

class TemperatureConverter:  
 @staticmethod  
 def celsius\_to\_fahrenheit(celsius):  
 return (celsius \* 9/5) + 32  
  
  
# Usage  
fahrenheit = TemperatureConverter.celsius\_to\_fahrenheit(0)  
print(fahrenheit) # Output: 32.0

In this example, celsius\_to\_fahrenheit is a static method because it simply performs a calculation without needing any data from a TemperatureConverter instance or class.

**Best Practices and Considerations:**

* Use @classmethod when you need access to class attributes or when you need to instantiate the class in the method (like factory methods).
* Use @staticmethod when the method does not interact with the class or its instances, and it serves as a utility function.

Both decorators enhance the readability and organization of your code by clearly indicating the purpose of the method to other developers. They are a vital part of writing clean, maintainable, and well-structured object-oriented code in Python.

**Property Decorators**

Property decorators play a crucial role. They allow for the management of class attributes by providing a way to implement getter, setter, and deleter methods, encapsulating the internal representation of attributes and offering a Pythonic way for attribute access and mutation.

**Understanding Property Decorators**

The @property decorator turns a method into a "getter" for a read-only attribute with the same name. To add corresponding "setter" and "deleter" functionality, you can decorate additional methods with @<attribute>.setter and @<attribute>.deleter.

**Getter Method**

A getter method is used to access the value of an attribute without directly exposing the attribute itself.

**Example: Getter for a Temperature Class**

class Temperature:  
 def \_\_init\_\_(self, celsius):  
 self.\_celsius = celsius  
  
 @property  
 def celsius(self):  
 return self.\_celsius  
  
  
# Usage  
temp = Temperature(100)  
print(temp.celsius) # 100

In this example, celsius is a property, and its value is accessed via the getter method.

**Setter Method**

A setter method is used to set the value of an attribute. It provides a controlled way of setting attributes, allowing for validation or computation.

**Example: Setter for Temperature Class**

class Temperature:  
 # ... (previous code)  
  
 @celsius.setter  
 def celsius(self, value):  
 if value < -273.15:  
 raise ValueError("Temperature below absolute zero!")  
 self.\_celsius = value  
  
  
# Usage  
temp = Temperature(0)  
temp.celsius = -300 # Raises ValueError

In this example, setting the celsius property is managed by the setter method, which includes validation.

**Deleter Method**

A deleter method is used to delete an attribute. It’s useful when deleting an attribute requires more than just removing it from the object’s \_\_dict\_\_.

**Example: Deleter for Temperature Class**

class Temperature:  
 # ... (previous code)  
  
 @celsius.deleter  
 def celsius(self):  
 print("Deleting celsius")  
 del self.\_celsius  
  
  
# Usage  
temp = Temperature(50)  
del temp.celsius # "Deleting celsius"

In this example, the deleter for celsius provides a controlled way to delete the \_celsius attribute.

**Benefits and Best Practices**

* *Encapsulation*: Property decorators help in encapsulating data and validation logic inside a class, adhering to the principles of encapsulation.
* *Readability and Maintenance*: They enhance readability and make maintenance easier, as changes to the attribute handling can be made in one place.
* *Backward Compatibility*: They allow you to introduce getter and setter methods without changing the class interface.

Property decorators are a powerful feature in Python for controlling attribute access and manipulation in a class, ensuring that the principles of encapsulation and abstraction are not violated. They provide a clean and Pythonic way to manage class attributes, making your code more robust, readable, and maintainable.

**Conclusion**

In this exploration of advanced Object-Oriented Programming in Python, we’ve uncovered the power and versatility of Python’s OOP features. From the nuanced world of magic methods to the efficient use of \_\_slots\_\_, and from the utility of classmethod and staticmethod decorators to the elegance of property decorators, these advanced concepts are vital tools in the repertoire of any proficient Python developer.

As you integrate these techniques into your coding practice, you’ll find your Python code becoming more robust, efficient, and maintainable. Remember, the journey to mastering Python is ongoing and ever-evolving. Keep experimenting, keep learning, and let these advanced OOP features inspire you to new heights in your Python programming journey

[*Mastering Advanced OOP Concepts in Python: Theory Behind OOP*](https://medium.com/@ramanbazhanau/mastering-advanced-oop-concepts-in-python-theory-behind-oop-c9e87fb1697b)