**Advanced OOP in Python**

Here’s an \*\*advanced tutorial on Object-Oriented Programming (OOP) in Python\*\*. This guide assumes you already understand the basics of OOP, such as classes, objects, inheritance, and polymorphism. We'll dive into advanced concepts like metaclasses, multiple inheritance, abstract base classes, and more.  
  
---  
  
## \*\*1. Advanced Class Features\*\*  
  
### \*\*a. Class and Static Methods\*\*  
- \*\*Class methods\*\* are methods that operate on the class itself rather than an instance.  
- \*\*Static methods\*\* are methods that don’t operate on the instance or the class but are logically related to the class.  
  
```python  
class MyClass:  
 class\_variable = "I am a class variable"  
  
 def \_\_init\_\_(self, instance\_variable):  
 self.instance\_variable = instance\_variable  
  
 @classmethod  
 def class\_method(cls):  
 return f"Accessing class variable: {cls.class\_variable}"  
  
 @staticmethod  
 def static\_method():  
 return "I am a static method and do not depend on class or instance"  
  
# Usage  
print(MyClass.class\_method()) # Access class variable  
print(MyClass.static\_method()) # Call static method  
```  
  
---  
  
### \*\*b. Properties and Property Decorators\*\*  
- Use `@property` to define getter, setter, and deleter methods for attributes.  
  
```python  
class Circle:  
 def \_\_init\_\_(self, radius):  
 self.\_radius = radius  
  
 @property  
 def radius(self):  
 return self.\_radius  
  
 @radius.setter  
 def radius(self, value):  
 if value < 0:  
 raise ValueError("Radius cannot be negative")  
 self.\_radius = value  
  
 @radius.deleter  
 def radius(self):  
 print("Deleting radius...")  
 del self.\_radius  
  
# Usage  
c = Circle(5)  
print(c.radius) # Access radius  
c.radius = 10 # Set radius  
del c.radius # Delete radius  
```  
  
---  
  
## \*\*2. Inheritance and Method Resolution Order (MRO)\*\*  
  
### \*\*a. Multiple Inheritance\*\*  
Python supports multiple inheritance, but it can lead to complexity. The \*\*Method Resolution Order (MRO)\*\* determines the order in which methods are resolved.  
  
```python  
class A:  
 def greet(self):  
 print("Hello from A")  
  
class B(A):  
 def greet(self):  
 print("Hello from B")  
  
class C(A):  
 def greet(self):  
 print("Hello from C")  
  
class D(B, C):  
 pass  
  
d = D()  
d.greet() # Output: Hello from B (MRO determines this)  
print(D.mro()) # Shows the MRO: [D, B, C, A, object]  
```  
  
---  
  
### \*\*b. Super() in Multiple Inheritance\*\*  
The `super()` function respects the MRO and allows you to call methods from parent classes.  
  
```python  
class A:  
 def greet(self):  
 print("Hello from A")  
  
class B(A):  
 def greet(self):  
 super().greet()  
 print("Hello from B")  
  
class C(A):  
 def greet(self):  
 super().greet()  
 print("Hello from C")  
  
class D(B, C):  
 def greet(self):  
 super().greet()  
 print("Hello from D")  
  
d = D()  
d.greet()  
# Output:  
# Hello from A  
# Hello from C  
# Hello from B  
# Hello from D  
```  
  
---  
  
## \*\*3. Abstract Base Classes (ABCs)\*\*  
  
Abstract Base Classes enforce that derived classes implement certain methods. Use the `abc` module.  
  
```python  
from abc import ABC, abstractmethod  
  
class Shape(ABC):  
 @abstractmethod  
 def area(self):  
 pass  
  
 @abstractmethod  
 def perimeter(self):  
 pass  
  
class Rectangle(Shape):  
 def \_\_init\_\_(self, width, height):  
 self.width = width  
 self.height = height  
  
 def area(self):  
 return self.width \* self.height  
  
 def perimeter(self):  
 return 2 \* (self.width + self.height)  
  
# Usage  
rect = Rectangle(10, 20)  
print(rect.area()) # 200  
print(rect.perimeter()) # 60  
```  
  
---  
  
## \*\*4. Metaclasses\*\*  
  
Metaclasses are classes of classes. They define how classes behave. By default, Python uses the `type` metaclass.  
  
### \*\*a. Custom Metaclass\*\*  
You can create a custom metaclass by inheriting from `type`.  
  
```python  
class MyMeta(type):  
 def \_\_new\_\_(cls, name, bases, dct):  
 print(f"Creating class {name}")  
 return super().\_\_new\_\_(cls, name, bases, dct)  
  
class MyClass(metaclass=MyMeta):  
 pass  
  
# Output: Creating class MyClass  
```  
  
### \*\*b. Use Case: Enforcing Class Attributes\*\*  
You can use metaclasses to enforce rules, such as requiring certain attributes in a class.  
  
```python  
class AttributeEnforcer(type):  
 def \_\_new\_\_(cls, name, bases, dct):  
 if 'required\_attribute' not in dct:  
 raise TypeError(f"Class {name} must define 'required\_attribute'")  
 return super().\_\_new\_\_(cls, name, bases, dct)  
  
class MyClass(metaclass=AttributeEnforcer):  
 required\_attribute = "I am required"  
  
# This will work, but removing `required\_attribute` will raise an error.  
```  
  
---  
  
## \*\*5. Design Patterns in OOP\*\*  
  
### \*\*a. Singleton Pattern\*\*  
Ensure a class has only one instance.  
  
```python  
class Singleton:  
 \_instance = None  
  
 def \_\_new\_\_(cls, \*args, \*\*kwargs):  
 if not cls.\_instance:  
 cls.\_instance = super().\_\_new\_\_(cls, \*args, \*\*kwargs)  
 return cls.\_instance  
  
# Usage  
s1 = Singleton()  
s2 = Singleton()  
print(s1 is s2) # True  
```  
  
### \*\*b. Factory Pattern\*\*  
A factory method creates objects without specifying the exact class.  
  
```python  
class Shape:  
 def draw(self):  
 pass  
  
class Circle(Shape):  
 def draw(self):  
 return "Drawing a Circle"  
  
class Square(Shape):  
 def draw(self):  
 return "Drawing a Square"  
  
class ShapeFactory:  
 @staticmethod  
 def get\_shape(shape\_type):  
 if shape\_type == "circle":  
 return Circle()  
 elif shape\_type == "square":  
 return Square()  
 else:  
 raise ValueError("Unknown shape type")  
  
# Usage  
shape = ShapeFactory.get\_shape("circle")  
print(shape.draw()) # Drawing a Circle  
```  
  
---  
  
## \*\*6. Operator Overloading\*\*  
  
You can define how operators like `+`, `-`, `\*`, etc., work for your custom classes by overriding special methods.  
  
```python  
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)  
  
 def \_\_repr\_\_(self):  
 return f"Vector({self.x}, {self.y})"  
  
# Usage  
v1 = Vector(1, 2)  
v2 = Vector(3, 4)  
print(v1 + v2) # Vector(4, 6)  
```  
  
---  
  
## \*\*7. Composition vs. Inheritance\*\*  
  
### \*\*Composition\*\*  
Instead of inheriting from a class, you can use composition to include an instance of another class.  
  
```python  
class Engine:  
 def start(self):  
 return "Engine started"  
  
class Car:  
 def \_\_init\_\_(self):  
 self.engine = Engine()  
  
 def start(self):  
 return self.engine.start()  
  
# Usage  
car = Car()  
print(car.start()) # Engine started  
```  
  
---  
  
## \*\*8. Advanced OOP Tips\*\*  
  
- \*\*Avoid deep inheritance hierarchies\*\*: Use composition over inheritance when possible.  
- \*\*Use `\_\_slots\_\_` for memory optimization\*\*: Restrict the attributes of a class to save memory.  
- \*\*Leverage `dataclasses` for boilerplate code\*\*: Use the `dataclasses` module to simplify class creation.  
  
```python  
from dataclasses import dataclass  
  
@dataclass  
class Point:  
 x: int  
 y: int  
  
p = Point(1, 2)  
print(p) # Point(x=1, y=2)  
```  
  
---  
  
This tutorial covers advanced OOP concepts in Python. Let me know if you'd like to dive deeper into any specific topic!

Here’s an extended tutorial that includes \*\*decorators\*\* and \*\*magic methods (dunder methods)\*\* in Python, which are essential for advanced object-oriented programming.  
  
---  
  
## \*\*1. Decorators in Python\*\*  
  
Decorators are a powerful feature in Python that allow you to modify the behavior of functions or methods. They are often used in OOP to enhance or control the behavior of class methods.  
  
### \*\*a. Function Decorators\*\*  
A decorator is a function that takes another function as input and returns a modified version of it.  
  
```python  
def my\_decorator(func):  
 def wrapper(\*args, \*\*kwargs):  
 print("Before the function call")  
 result = func(\*args, \*\*kwargs)  
 print("After the function call")  
 return result  
 return wrapper  
  
@my\_decorator  
def say\_hello(name):  
 print(f"Hello, {name}!")  
  
# Usage  
say\_hello("Alice")  
# Output:  
# Before the function call  
# Hello, Alice!  
# After the function call  
```  
  
---  
  
### \*\*b. Class Method Decorators\*\*  
You can use decorators to modify the behavior of class methods.  
  
```python  
def log\_method\_call(func):  
 def wrapper(\*args, \*\*kwargs):  
 print(f"Calling method: {func.\_\_name\_\_}")  
 return func(\*args, \*\*kwargs)  
 return wrapper  
  
class MyClass:  
 @log\_method\_call  
 def greet(self, name):  
 print(f"Hello, {name}!")  
  
# Usage  
obj = MyClass()  
obj.greet("Bob")  
# Output:  
# Calling method: greet  
# Hello, Bob!  
```  
  
---  
  
### \*\*c. Class Decorators\*\*  
A class decorator modifies the behavior of an entire class.  
  
```python  
def add\_str\_method(cls):  
 def \_\_str\_\_(self):  
 return f"Instance of {cls.\_\_name\_\_} with attributes {self.\_\_dict\_\_}"  
 cls.\_\_str\_\_ = \_\_str\_\_  
 return cls  
  
@add\_str\_method  
class Person:  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
# Usage  
p = Person("Alice", 30)  
print(p)  
# Output: Instance of Person with attributes {'name': 'Alice', 'age': 30}  
```  
  
---  
  
### \*\*d. Property Decorators\*\*  
The `@property` decorator is used to define getter, setter, and deleter methods for attributes.  
  
```python  
class Rectangle:  
 def \_\_init\_\_(self, width, height):  
 self.\_width = width  
 self.\_height = height  
  
 @property  
 def area(self):  
 return self.\_width \* self.\_height  
  
 @property  
 def width(self):  
 return self.\_width  
  
 @width.setter  
 def width(self, value):  
 if value < 0:  
 raise ValueError("Width must be positive")  
 self.\_width = value  
  
# Usage  
rect = Rectangle(10, 20)  
print(rect.area) # 200  
rect.width = 15  
print(rect.area) # 300  
```  
  
---  
  
## \*\*2. Magic Methods (Dunder Methods)\*\*  
  
Magic methods (also called dunder methods because they are surrounded by double underscores) allow you to define how objects of your class behave with built-in operations like addition, comparison, string representation, etc.  
  
---  
  
### \*\*a. `\_\_init\_\_` and `\_\_repr\_\_`\*\*  
- `\_\_init\_\_`: Initializes an object.  
- `\_\_repr\_\_`: Provides an unambiguous string representation of the object.  
  
```python  
class Point:  
 def \_\_init\_\_(self, x, y):  
 self.x = x  
 self.y = y  
  
 def \_\_repr\_\_(self):  
 return f"Point({self.x}, {self.y})"  
  
# Usage  
p = Point(1, 2)  
print(p) # Output: Point(1, 2)  
```  
  
---  
  
### \*\*b. Operator Overloading\*\*  
You can define how operators like `+`, `-`, `\*`, etc., work for your custom classes.  
  
#### \*\*Addition (`\_\_add\_\_`)\*\*  
```python  
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)  
  
 def \_\_repr\_\_(self):  
 return f"Vector({self.x}, {self.y})"  
  
# Usage  
v1 = Vector(1, 2)  
v2 = Vector(3, 4)  
print(v1 + v2) # Output: Vector(4, 6)  
```  
  
#### \*\*Comparison (`\_\_lt\_\_`, `\_\_eq\_\_`)\*\*  
```python  
class Person:  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
 def \_\_lt\_\_(self, other):  
 return self.age < other.age  
  
 def \_\_eq\_\_(self, other):  
 return self.age == other.age  
  
# Usage  
p1 = Person("Alice", 30)  
p2 = Person("Bob", 25)  
print(p1 < p2) # False  
print(p1 == p2) # False  
```  
  
---  
  
### \*\*c. Customizing String Representation\*\*  
- `\_\_str\_\_`: Defines a user-friendly string representation.  
- `\_\_repr\_\_`: Defines a developer-friendly string representation.  
  
```python  
class Book:  
 def \_\_init\_\_(self, title, author):  
 self.title = title  
 self.author = author  
  
 def \_\_str\_\_(self):  
 return f"'{self.title}' by {self.author}"  
  
 def \_\_repr\_\_(self):  
 return f"Book(title={self.title}, author={self.author})"  
  
# Usage  
b = Book("1984", "George Orwell")  
print(str(b)) # '1984' by George Orwell  
print(repr(b)) # Book(title=1984, author=George Orwell)  
```  
  
---  
  
### \*\*d. Context Managers (`\_\_enter\_\_` and `\_\_exit\_\_`)\*\*  
Magic methods `\_\_enter\_\_` and `\_\_exit\_\_` allow you to use objects in a `with` statement.  
  
```python  
class FileManager:  
 def \_\_init\_\_(self, filename, mode):  
 self.filename = filename  
 self.mode = mode  
  
 def \_\_enter\_\_(self):  
 self.file = open(self.filename, self.mode)  
 return self.file  
  
 def \_\_exit\_\_(self, exc\_type, exc\_value, traceback):  
 self.file.close()  
  
# Usage  
with FileManager("test.txt", "w") as f:  
 f.write("Hello, world!")  
# File is automatically closed after the block  
```  
  
---  
  
### \*\*e. Callable Objects (`\_\_call\_\_`)\*\*  
The `\_\_call\_\_` method allows an instance of a class to be called like a function.  
  
```python  
class Multiplier:  
 def \_\_init\_\_(self, factor):  
 self.factor = factor  
  
 def \_\_call\_\_(self, value):  
 return value \* self.factor  
  
# Usage  
double = Multiplier(2)  
print(double(5)) # Output: 10  
```  
  
---  
  
### \*\*f. Length and Indexing (`\_\_len\_\_` and `\_\_getitem\_\_`)\*\*  
- `\_\_len\_\_`: Defines the behavior of `len()`.  
- `\_\_getitem\_\_`: Allows indexing.  
  
```python  
class CustomList:  
 def \_\_init\_\_(self, items):  
 self.items = items  
  
 def \_\_len\_\_(self):  
 return len(self.items)  
  
 def \_\_getitem\_\_(self, index):  
 return self.items[index]  
  
# Usage  
cl = CustomList([1, 2, 3, 4])  
print(len(cl)) # Output: 4  
print(cl[2]) # Output: 3  
```  
  
---  
  
### \*\*g. Dynamic Attribute Access (`\_\_getattr\_\_`, `\_\_setattr\_\_`, `\_\_delattr\_\_`)\*\*  
- `\_\_getattr\_\_`: Called when an attribute is not found.  
- `\_\_setattr\_\_`: Called when setting an attribute.  
- `\_\_delattr\_\_`: Called when deleting an attribute.  
  
```python  
class DynamicAttributes:  
 def \_\_init\_\_(self):  
 self.attributes = {}  
  
 def \_\_getattr\_\_(self, name):  
 return self.attributes.get(name, f"{name} not found")  
  
 def \_\_setattr\_\_(self, name, value):  
 if name == "attributes":  
 super().\_\_setattr\_\_(name, value)  
 else:  
 self.attributes[name] = value  
  
 def \_\_delattr\_\_(self, name):  
 if name in self.attributes:  
 del self.attributes[name]  
 else:  
 raise AttributeError(f"{name} not found")  
  
# Usage  
obj = DynamicAttributes()  
obj.name = "Alice"  
print(obj.name) # Output: Alice  
del obj.name  
print(obj.name) # Output: name not found  
```  
  
---  
  
This tutorial covers \*\*decorators\*\* and \*\*magic methods\*\* in Python, which are essential for advanced OOP. Let me know if you’d like to explore any of these topics further!