



PYTHON IS FOR EVERYONE

Tutorial 13:

PYTHON PROGRAMMING - INTRO TO
OBJECT-ORIENTED PROGRAMMING (OOP)
IN GOOGLE COLAB



Jeff Gentry

[.@www.linkedin.com/in/jefferycharlesgentry.](https://www.linkedin.com/in/jefferycharlesgentry)



Objectives


- Understand the principles of Object-Oriented Programming.
- Learn how to define classes and create objects in Python.
- Explore the concepts of inheritance, encapsulation, and polymorphism.
- Practice OOP concepts through hands-on exercises.



What is Object-Oriented Programming?



Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to represent data and methods to manipulate that data. OOP is based on several key principles:

- **Encapsulation:** Bundling data (attributes) and methods (functions) that operate on the data into a single unit (class).
 - **Inheritance:** Creating new classes based on existing classes, allowing for code reuse and the creation of hierarchical relationships.
 - **Polymorphism:** Allowing different classes to be treated as instances of the same class through a common interface.
- 



Defining a Class

A class is a blueprint for creating objects. You define a “class” using the class keyword.



Tutorial_13.ipynb



File Edit View Insert Runtime Tools Help

+ Code + Text



0s



```
class Dog:
    def __init__(self, name, age):
        self.name = name    # Attribute
        self.age = age      # Attribute

    def bark(self):         # Method
        return "Woof!"
```

Creating Objects



An object is an instance of a class. You create an object by calling the class as if it were a function.

```
> my_dog = Dog("Buddy", 3) # Creating an object of the Dog class
print(my_dog.name)        # Output: Buddy
print(my_dog.bark())      # Output: Woof!
```

```
> Buddy
Woof!
```





Inheritance

Inheritance allows a new class (child class) to inherit attributes and methods from an existing class (parent class). This promotes code reuse.

```
class Animal: # Parent class
    def __init__(self, species):
        self.species = species

    def make_sound(self):
        return "Some sound"

class Cat(Animal): # Child class
    def __init__(self, name, age):
        super().__init__("Cat") # Call the parent class constructor
        self.name = name
        self.age = age

    def make_sound(self): # Overriding the parent method
        return "Meow!"

my_cat = Cat("Whiskers", 2)
print(my_cat.species)      # Output: Cat
print(my_cat.make_sound()) # Output: Meow!
```

```
Cat
Meow!
```

Encapsulation



Encapsulation restricts access to certain attributes and methods, promoting data hiding. You can use underscores to indicate private attributes.

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance # Private attribute

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance

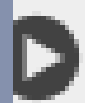
account = BankAccount(1000)
account.deposit(500)
print(account.get_balance()) # Output: 1500
# print(account.__balance) # This will raise an AttributeError
```

1500



Polymorphism

Polymorphism allows methods to do different things based on the object calling them. It can be achieved through method overriding and duck typing.



```
class Bird:
    def make_sound(self):
        return "Chirp!"

class Dog:
    def make_sound(self):
        return "Woof!"

def animal_sound(animal):
    print(animal.make_sound())

my_bird = Bird()
my_dog = Dog()

animal_sound(my_bird)    # Output: Chirp!
animal_sound(my_dog)    # Output: Woof!
```



```
Chirp!
Woof!
```



Practice Exercises



Create a Class: Define a class called “Car” with attributes like “make”, “model”, and “year”. Include a method to display the car's information.

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def display_info(self):
        return f"{self.year} {self.make} {self.model}"

my_car = Car("Toyota", "Corolla", 2020)
print(my_car.display_info()) # Output: 2020 Toyota Corolla
```

```
→ 2020 Toyota Corolla
```



Practice Exercises

Inheritance: Create a class “ElectricCar” that inherits from the “Car” class and adds an attribute for battery size. Include a method to display the battery size.

```
class ElectricCar(Car):
    def __init__(self, make, model, year, battery_size):
        super().__init__(make, model, year)
        self.battery_size = battery_size

    def display_battery(self):
        return f"Battery size: {self.battery_size} kWh"

my_electric_car = ElectricCar("Tesla", "Model S", 2021, 100)
print(my_electric_car.display_info()) # Output: 2021 Tesla Model S
print(my_electric_car.display_battery()) # Output: Battery size: 100 kWh
```

```
2021 Tesla Model S
Battery size: 100 kWh
```

Practice Exercises

Encapsulation: Create a class “Person” with private attributes for “name” and “age”. Include methods to set and get these attributes.

```
> class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def set_name(self, name):
        self.__name = name

    def get_name(self):
        return self.__name

    def set_age(self, age):
        self.__age = age

    def get_age(self):
        return self.__age

person = Person("Jeff", 42)
print(person.get_name()) # Output: Jeff
person.set_age(43)
print(person.get_age()) # Output: 43
```

```
Jeff
43
```

Practice Exercises

Polymorphism: Create a function that takes different shapes (like “Circle” and “Square” classes) and calculates their area using polymorphism.

```
> class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * (self.radius ** 2)

class Square:
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side ** 2

def print_area(shape):
    print("Area:", shape.area())

my_circle = Circle(5)
my_square = Square(4)

print_area(my_circle)    # Output: Area: 78.5
print_area(my_square)    # Output: Area: 16
```

```
> Area: 78.5
Area: 16
```

Practice Exercises

Create a Simple Banking System: Define a class “Bank” that allows creating accounts, depositing, and withdrawing money. Use encapsulation to protect the balance.

```
class Bank:
    def __init__(self):
        self.__accounts = {}

    def create_account(self, account_number):
        self.__accounts[account_number] = 0

    def deposit(self, account_number, amount):
        if account_number in self.__accounts:
            self.__accounts[account_number] += amount

    def withdraw(self, account_number, amount):
        if (account_number in self.__accounts and
            self.__accounts[account_number] >= amount):
            self.__accounts[account_number] -= amount

    def get_balance(self, account_number):
        return self.__accounts.get(account_number, "Account not found")

bank = Bank()
bank.create_account("12345")
bank.deposit("12345", 500)
print(bank.get_balance("12345"))  # Output: 500
bank.withdraw("12345", 200)
print(bank.get_balance("12345"))  # Output: 300
```

500

300



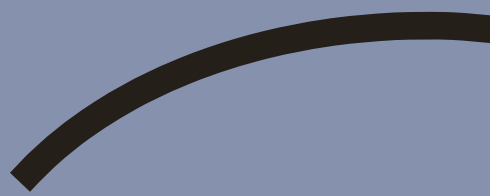
Conclusion

In this tutorial, you learned the fundamental concepts of Object-Oriented Programming in Python, including classes, objects, inheritance, encapsulation, and polymorphism. Understanding OOP principles is essential for writing modular and maintainable code, allowing for better organization and reuse of code in your Python projects.



Next Steps

In tutorial 14, you will apply the concepts you've learned throughout the previous tutorials to create a comprehensive project. This project will involve building a simple command-line-based library management system.



FOLLOW ME

for more tips you
didn't know you
needed



Jeff Gentry

[.@www.linkedin.com/in/jefferycharlesgentry](https://www.linkedin.com/in/jefferycharlesgentry)