Python and Object-Oriented Programming

Scott Tremaine
Software Developer and Educator

Breakpoint Coding Tutorials

Contents

Introduction to Object-Oriented Programming (OOP)

Object-Oriented Programming, commonly abbreviated as OOP, is a programming paradigm that uses "objects" to design applications and computer programs. These objects can be thought of as real-world entities that have both attributes (which define their state) and behaviors (which define what they can do). OOP is based on several foundational principles that help in structuring software in a way that is both modular and reusable.

In traditional procedural programming, the focus is on functions or procedures that operate on data. In contrast, OOP shifts the focus to the data itself by encapsulating it within objects. This encapsulation helps in modeling complex systems more intuitively and naturally. For example, in a banking system, you can have objects like Customer, Account, and Transaction, each with its own attributes and behaviors.

Imagine a car manufacturing company. Instead of focusing solely on the processes (like assembling parts or painting), the company focuses on creating different models of cars, where each car model is an object with its own set of characteristics (color, model, engine type) and behaviors (drive, brake, honk).

Key Concepts of OOP: Classes and Objects

At the heart of OOP are two fundamental concepts: classes and objects.

Classes

A class can be thought of as a blueprint or template for creating objects. It defines a set of attributes and methods that the created objects will have. For instance, consider the Car class in our previous example. The Car class would define attributes like color, model, and engine_type, and methods like drive, brake, and honk.

Objects

An object is an instance of a class. When you create an object from a class, you are essentially creating a specific example based on the class blueprint. For instance, you could create an object `my_car` from the Car class with the attributes `color = red`, `model = sedan`, and `engine_type = electric`. This object can then use the methods defined in the Car class, such as `drive()` or `brake()`.

Here's a simple Python example to illustrate the concepts of classes and objects:

```python
class Car:
    def __init__(self, color, model, engine_type):
        self.color = color
        self.model = model
        self.engine_type = engine_type

    def drive(self):
        print(f"The {self.color} {self.model} is driving.")

    def brake(self):
        print(f"The {self.color} {self.model} is braking.")
```

```python
# Creating an object of the Car class
my_car = Car("red", "sedan", "electric")

# Using the methods of the Car class
my_car.drive()
my_car.brake()
```

In this example, `Car` is the class, and `my_car` is an object of the Car class. The `__init__` method is a special method called a constructor that initializes the object's attributes. The `drive` and `brake` methods define the behaviors of the Car class.

Benefits of OOP

OOP offers several advantages that make it a popular choice for software development:

- **Modularity:** By dividing the program into objects, OOP makes it easier to manage and understand complex systems. Each object can be developed and tested independently.
- **Reusability:** Classes can be reused across different programs. Once a class is written, it can be used to create multiple objects without rewriting the code. For example, the Car class can be used to create many car objects with different attributes.
- **Extensibility:** OOP allows for extending existing code without modifying it. This is achieved through inheritance, where new classes can inherit attributes and methods from existing classes. For instance, you could create a `SportsCar` class that inherits from the `Car` class and adds additional features like `turbo_boost`.
- **Maintainability:** OOP makes it easier to maintain and update code. Because objects encapsulate their data and behaviors, changes to one part of the system can be made with minimal impact on other parts. This encapsulation leads to a more organized and modular codebase.
- **Real-World Modeling:** OOP helps in creating systems that are closer to real-world entities and interactions. This makes it easier for developers to map their software to real-world scenarios, improving both design and implementation.
- **Security:** By using access modifiers (like private and protected attributes), OOP allows control over how data is accessed and modified. This data hiding ensures that objects are used only in intended ways, reducing the risk of unintended interactions.

In summary, OOP provides a robust framework for building scalable, reusable, and maintainable software. Its principles of encapsulation, inheritance, and polymorphism offer a powerful way to manage complexity and enhance the clarity of code. By focusing on objects rather than procedures, OOP aligns closely with how we perceive and interact with the world, making it an intuitive and effective approach to programming.

Creating Classes and Objects

Defining a Class

Classes are the fundamental building blocks of Object-Oriented Programming (OOP). They serve as a blueprint for creating objects, which are instances of the class, like a template, outlining the properties and behaviors that its objects will have.

To define a class in Python, you use the `class` keyword followed by the class name and a colon. By convention, class names typically use CamelCase notation, where the first letter of each word is capitalized.

Here's a basic example of a class definition:

```python
class Dog:
    pass
```

In this example, `Dog` is a class with no properties or methods. The `pass` statement indicates that the class is empty. While this class doesn't do much, it serves as a starting point for defining more complex classes.

Classes in Python can contain several components:

- **Attributes:** Variables that hold data pertaining to the class.
- **Methods:** Functions defined within a class that describe the behaviors of the objects created from the class.

Let's enhance our `Dog` class by adding some attributes and methods.

```python
class Dog:
    # Class attribute
    species = "Canis familiaris"

    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Method to describe the dog
    def description(self):
        return f"{self.name} is {self.age} years old."

    # Method to simulate a dog barking
    def bark(self, sound):
        return f"{self.name} says {sound}"
```

In this improved version, we have:

- **Class Attribute:** `species` is shared by all instances of the `Dog` class.
- **Instance Attributes:** `name` and `age` are unique to each instance.
- **Methods:** `description` and `bark` perform actions using the object's data.

Creating Objects from Classes

Once you've defined a class, you can create objects, or instances, of that class. Each instance is an independent object with its own set of attributes and methods.

To create an instance of a class, you call the class using its name followed by parentheses. Inside the parentheses, you pass any arguments that the `__init__` method requires.

Here's how you can create instances of the `Dog` class:

```python
# Create instances of the Dog class
my_dog = Dog("Buddy", 3)
your_dog = Dog("Milo", 5)
```

In this example:

- `my_dog` is an instance of the `Dog` class with the name "Buddy" and age 3.
- `your_dog` is another instance with the name "Milo" and age 5.

Each object has its own unique attributes, though they share the same class attribute `species`.

You can access the attributes and methods of these instances using dot notation:

```python
print(my_dog.description())  # Output: Buddy is 3 years old.
print(your_dog.bark("Woof!"))  # Output: Milo says Woof!
```

This code demonstrates how to interact with the objects you've created, utilizing their attributes and methods.

The `__init__` Method

The `__init__` method, also known as the initializer or constructor, is a special method in Python classes. It's automatically called when a new instance of the class is created. The purpose of `__init__` is to initialize the object's attributes with values provided by the user.

The `__init__` method always takes at least one parameter, `self`, which refers to the instance being created. Additional parameters can be added to accept values that will initialize the object's attributes.

Here's a closer look at the `__init__` method from our `Dog` class:

```python
def __init__(self, name, age):
    self.name = name
    self.age = age
```

In this method:

- `self.name = name` assigns the value of the `name` parameter to the `name` attribute of the object.
- `self.age = age` does the same for the `age` attribute.

When you create a new `Dog` object, you provide values for `name` and `age`, which are passed to the `__init__` method and used to set the object's attributes.
Consider this example:

```python
new_dog = Dog("Charlie", 2)
print(new_dog.name)  # Output: Charlie
print(new_dog.age)  # Output: 2
```

In this snippet:

- A new `Dog` object named `new_dog` is created with the name "Charlie" and age 2.
- The `__init__` method sets the `name` and `age` attributes accordingly.
- You can access these attributes using dot notation.

The `__init__` method ensures that each instance of the class starts with a valid state, with all necessary attributes properly initialized.

Class Attributes and Methods

In Python's object-oriented programming paradigm, understanding the distinction between instance attributes and class attributes is crucial for creating robust and maintainable code.

Instance Attributes

Instance attributes are specific to each object created from a class. When you define an instance attribute, you do so within a method, typically the __init__ method. These attributes are unique to each instance of the class, meaning that each object can hold different values for these attributes.

Consider the following example:

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

# Creating instances of the Dog class
dog1 = Dog("Buddy", 3)
dog2 = Dog("Lucy", 5)

print(dog1.name)  # Output: Buddy
print(dog2.name)  # Output: Lucy
```

In this example, `name` and `age` are instance attributes. Each `Dog` instance has its own `name` and `age`, demonstrating that instance attributes are tied to the specific instance of the class.

Class Attributes

Class attributes, on the other hand, are shared across all instances of a class. They are defined directly within the class but outside any methods. Class attributes are typically used for constants or attributes that should be the same for every instance.

Here's an example to illustrate:

```
class Dog:
    species = "Canis familiaris"  # Class attribute

    def __init__(self, name, age):
        self.name = name
        self.age = age

# Creating instances of the Dog class
dog1 = Dog("Buddy", 3)
dog2 = Dog("Lucy", 5)

print(dog1.species)  # Output: Canis familiaris
print(dog2.species)  # Output: Canis familiaris
```

In this case, `species` is a class attribute. Regardless of how many `Dog` instances you create, the `species` attribute will always be `"Canis familiaris"`.

Understanding when to use instance versus class attributes is key to leveraging Python's OOP capabilities effectively. Instance attributes are ideal for data unique to each instance, while class attributes are useful for data shared across all instances.

Defining Methods

Methods in Python are functions defined within a class that describe the behaviors of the objects created from the class. Methods are essentially functions that operate on the data contained in the class (i.e., instance or class attributes) and typically modify or perform actions using this data.

Instance Methods

Instance methods are the most common type of method in Python classes. They operate on an instance of the class and have access to the instance through the `self` parameter. These methods can modify the object's state by changing its instance attributes.

Here's an example:

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

    def bark(self):
        return f"{self.name} is barking."

    def birthday(self):
        self.age += 1
        return f"Happy Birthday {self.name}! You are now {self.age}
   years old."

# Creating an instance of the Dog class
dog1 = Dog("Buddy", 3)
print(dog1.bark())  # Output: Buddy is barking.
print(dog1.birthday())  # Output: Happy Birthday Buddy! You are now 4
   years old.
```

In this example, `bark` and `birthday` are instance methods. They can access and modify the instance attributes (`name` and `age`) using the `self` parameter.

Class Methods

Class methods are methods that operate on the class itself rather than on instances of the class. They are defined using the `@classmethod` decorator and take a `cls` parameter instead of `self`. This parameter refers to the class, and through it, class methods can modify class attributes.

Example:

```python
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def set_species(cls, species_name):
        cls.species = species_name

# Changing the class attribute using the class method
Dog.set_species("Canis lupus familiaris")

# Creating an instance of the Dog class
dog1 = Dog("Buddy", 3)
print(dog1.species)  # Output: Canis lupus familiaris
```

In this example, `set_species` is a class method that changes the class attribute `species` for all instances of the `Dog` class.

Static Methods

Static methods are similar to class methods but don't operate on class or instance attributes. They are defined using the `@staticmethod` decorator and don't take a `self` or `cls` parameter. Static methods are utility functions that perform a task in isolation.

Example:

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

    @staticmethod
    def is_adult(age):
        return age >= 2

# Using the static method
print(Dog.is_adult(3))  # Output: True
print(Dog.is_adult(1))  # Output: False
```

In this example, `is_adult` is a static method that determines if a dog is an adult based on its age. It doesn't need to access any class or instance attributes to perform its function.

Using `self` in Methods

The `self` parameter is an essential part of defining instance methods in Python. It refers to the instance calling the method, allowing access to the instance's attributes and other methods.

Accessing Attributes

Within an instance method, `self` is used to access attributes and methods of the class. This is crucial for manipulating the state of an object and ensuring that each instance maintains its own data.

Consider this example:

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

    def description(self):
        return f"{self.name} is {self.age} years old."

# Creating an instance of the Dog class
dog1 = Dog("Buddy", 3)
print(dog1.description())  # Output: Buddy is 3 years old.
```

Here, `self.name` and `self.age` within the `description` method refer to the `name` and `age` attributes of the `dog1` instance.

Modifying Attributes

Instance methods often modify the state of an object using `self` to change the values of instance attributes. This allows for dynamic changes to an object's state during its lifecycle.

Example:

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

    def have_birthday(self):
        self.age += 1
        return f"Happy Birthday {self.name}! You are now {self.age}
   years old."

# Creating an instance of the Dog class
dog1 = Dog("Buddy", 3)
print(dog1.have_birthday())  # Output: Happy Birthday Buddy! You are
   now 4 years old.
```

In this case, `self.age += 1` increments the `age` attribute of the `dog1` instance by one, demonstrating how `self` is used to modify instance attributes.

Calling Other Methods

The `self` parameter also allows for calling other methods within the class. This promotes code reuse and modular design by enabling methods to leverage the functionality of other methods.

Example:

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

    def bark(self):
        return f"{self.name} is barking."

    def celebrate_birthday(self):
        self.age += 1
        return f"{self.bark()} Happy Birthday {self.name}! You are now
   {self.age} years old."

# Creating an instance of the Dog class
dog1 = Dog("Buddy", 3)
print(dog1.celebrate_birthday())  # Output: Buddy is barking. Happy
   Birthday Buddy! You are now 4 years old.
```

In this example, the `celebrate_birthday` method calls the `bark` method using `self.bark()`, demonstrating how methods can interact with each other through `self`.

In conclusion, `self` is a powerful mechanism that provides the flexibility to access and manipulate an object's attributes and methods, making it a cornerstone of Python's OOP capabilities.

Understanding Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class to inherit attributes and methods from another class. This mechanism promotes code reusability and establishes a natural hierarchical relationship between classes. Imagine you are working on a project involving various types of vehicles. Instead of defining common properties and methods for each vehicle type separately, you can define a general Vehicle class and let other specific vehicle classes, such as Car, Truck, and Motorcycle, inherit from it.

In Python, inheritance is implemented by defining a new class that takes an existing class as its parent. The parent class is often referred to as the base class or superclass, and the class that inherits from it is called the subclass or derived class.

Creating Subclasses

To create a subclass in Python, you simply define a new class and specify the superclass in parentheses. This setup allows the subclass to inherit all attributes and methods of the superclass.

Let's look at an example. Suppose we have a Vehicle class with some basic properties and methods:

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

    def start_engine(self):
        print(f"{self.year} {self.make} {self.model}'s engine started."
   )
```

Now, let's create a subclass called Car that inherits from Vehicle:

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

    def honk_horn(self):
        print("Beep beep!")
```

In this example, the Car class inherits from Vehicle. This means Car has access to the __init__ method and start_engine method defined in Vehicle. Additionally, Car introduces a new attribute, num_doors, and a new method, honk_horn.

When we create an instance of Car, it can utilize both its own methods and the inherited methods:

```
my_car = Car("Toyota", "Corolla", 2020, 4)
my_car.start_engine()  # Inherited method
my_car.honk_horn()     # Car's own method
```

Overriding Methods

Sometimes, a subclass may need to provide a specific implementation of a method that is already defined in its superclass. This is known as method overriding. When a method in a subclass has the same name as a method in the superclass, the subclass's method overrides the superclass's method.

Consider the following example, where the Truck class overrides the start_engine method to include additional behavior:

```
class Truck(Vehicle):
    def __init__(self, make, model, year, cargo_capacity):
        super().__init__(make, model, year)
        self.cargo_capacity = cargo_capacity

    def start_engine(self):
        print(f"{self.year} {self.make} {self.model}'s powerful engine
   started.")
```

In this case, the Truck class has its own implementation of the start_engine method, which provides a more specific message:

```python
my_truck = Truck("Ford", "F-150", 2021, 1300)
my_truck.start_engine()  # Overridden method
```

When my_truck.start_engine() is called, the output will be the message defined in the Truck class, not the one in the Vehicle class.

Using `super()`

To effectively manage method overriding and ensure that the superclass's methods are properly invoked, Python provides the `super()` function. The `super()` function returns a temporary object of the superclass that allows you to call its methods.

In our previous examples, we used `super()` in the __init__ method of the subclasses to initialize the attributes of the superclass. Here's a more detailed look at how `super()` works in method overriding:

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

    def start_engine(self):
        super().start_engine()  # Call the superclass's start_engine
   method
        print(f"{self.year} {self.make} {self.model} runs on a {self.
   battery_capacity}kWh battery.")
```

In this example, the ElectricCar class inherits from Car. The start_engine method in ElectricCar calls `super().start_engine()` to execute the start_engine method from the Car class, ensuring that the standard engine start message is printed before adding its specific message about the battery.

Using `super()` is a best practice because it allows you to extend the behavior of the superclass method rather than completely replacing it. This technique promotes code reuse and maintains a clear hierarchy and structure within your classes.

Encapsulation and Data Hiding

Encapsulation and data hiding are foundational concepts that ensure the integrity and security of data within a program. These principles promote modular design, making code more manageable, reusable, and resistant to errors. Encapsulation involves bundling data and methods that operate on the data within a single unit, typically a class, and restricting access to some of the object's components.

Private Attributes and Methods

Encapsulation in Python is primarily achieved through private attributes and methods. In Python, an attribute or method is considered private when it is intended to be accessed only within its class, shielding it from the outside world. While Python does not enforce strict access controls like some other languages, it follows a convention-based approach.

Understanding Private Attributes

Private attributes in Python are denoted by a single or double underscore prefix. This naming convention signals to other programmers that these attributes should not be accessed directly.

- **Single Underscore (_):** This is a weak "internal use" indicator. It suggests that the attribute is intended for internal use within the module or class, but it does not enforce strict privacy.
- **Double Underscore (__):** This triggers name mangling, where the interpreter changes the attribute name to include the class name, making it harder to access from outside the class. This provides a stronger form of encapsulation.

Consider the following example:

```python
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self._year = year  # Single underscore suggests internal use
        self.__odometer = 0  # Double underscore for stronger
   encapsulation

    def get_odometer(self):
        return self.__odometer

    def drive(self, miles):
        if miles > 0:
            self.__odometer += miles
        else:
            print("Miles must be greater than zero")

my_car = Car("Toyota", "Corolla", 2020)
print(my_car.make)  # Accessible
print(my_car._year)  # Accessible, but should be treated as internal
print(my_car.__odometer)  # AttributeError: 'Car' object has no
   attribute '__odometer'
```

In this example, `make` and `model` are public attributes, `_year` is a protected attribute intended for internal use, and `__odometer` is a private attribute.

Private Methods

Similarly, private methods are created using a double underscore prefix. These methods are intended to be used only within the class, providing a mechanism to hide internal logic.

```python
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model
    def start(self):
        self.__engage_ignition()
        print("Car started")

    def __engage_ignition(self):
        print("Ignition engaged")

my_car = Car("Toyota", "Corolla")
my_car.start()  # This works and calls the private method internally
my_car.__engage_ignition()  # AttributeError: 'Car' object has no
   attribute '__engage_ignition'
```

In this scenario, the `__engage_ignition` method is a private method, encapsulated within the `Car` class, and cannot be called directly from outside the class.

Why Use Private Attributes and Methods?

- **Encapsulation:** Private attributes and methods bundle data and methods into a single unit, ensuring that the internal state of an object can only be modified in a controlled way.
- **Data Integrity:** They help protect the internal state of an object from unwanted changes, which might lead to inconsistencies.
- **Implementation Hiding:** Private members hide the implementation details, exposing only what is necessary through public methods. This promotes modular design and makes the code easier to understand and maintain.

Property Decorators for Encapsulation

While private attributes and methods offer a way to restrict access, there are times when controlled access to attributes is necessary. This is where property decorators come into play. Property decorators in Python provide a way to define methods in a class that act like attributes, allowing you to encapsulate data and provide a controlled way to access and modify it.

Introduction to Property Decorators

Property decorators are a built-in feature in Python that allows you to define getter, setter, and deleter methods for an attribute. This enables you to add logic to attribute access and modification without changing the interface of the class.

Consider the following example:

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

    @property
    def make(self):
        return self._make
    @make.setter
    def make(self, value):
        if isinstance(value, str):
            self._make = value
        else:
            raise ValueError("Make must be a string")

    @property
    def model(self):
        return self._model

    @model.setter
    def model(self, value):
        if isinstance(value, str):
            self._model = value
        else:
            raise ValueError("Model must be a string")

my_car = Car("Toyota", "Corolla")
print(my_car.make)  # Access using getter
my_car.make = "Honda"  # Modify using setter
print(my_car.make)  # Access using getter
```

In this example, the `make` and `model` attributes are managed through property decorators, allowing controlled access and modification.

Benefits of Using Property Decorators

- **Controlled Access:** Property decorators allow you to control how attributes are accessed and modified. You can add validation logic to ensure that the attribute values are valid.
- **Read-Only Attributes:** You can create read-only attributes by defining only the getter method, preventing modifications from outside the class.
- **Consistency:** They help maintain a consistent interface, making the code easier to use and understand.

Defining Property Decorators

Property decorators consist of three methods:

- `@property`: Defines the getter method.
- `@<property>.setter`: Defines the setter method.
- `@<property>.deleter`: Defines the deleter method.

Let's extend the previous example to include a read-only attribute and a deleter method:

```
class Car:
    def __init__(self, make, model, year):
        self._make = make
        self._model = model
        self._year = year
    @property
    def make(self):
        return self._make

    @make.setter
    def make(self, value):
        if isinstance(value, str):
            self._make = value
        else:
            raise ValueError("Make must be a string")

    @property
    def model(self):
        return self._model

    @model.setter
    def model(self, value):
        if isinstance(value, str):
            self._model = value
        else:
            raise ValueError("Model must be a string")

    @property
    def year(self):
        return self._year

    @year.deleter
    def year(self):
        del self._year

my_car = Car("Toyota", "Corolla", 2020)
print(my_car.year)  # Access using getter
del my_car.year  # Delete using deleter
```

In this extended example, the `year` attribute is read-only and can be deleted using the deleter method.

Polymorphism

Polymorphism refers to the ability of different objects to be treated as instances of the same class through a common interface. This concept allows objects to be manipulated based on their shared characteristics rather than their specific types, enabling more flexible and reusable code. In Python, polymorphism is primarily achieved through method overloading, method overriding, and duck typing.

Method Overloading

Method overloading is the ability to define multiple methods with the same name but different signatures within the same class. In many programming languages, method overloading is achieved by varying the number or type of parameters, allowing the same method name to perform different tasks based on the input arguments. However, Python does not support traditional method overloading as seen in languages like Java or C++.

In Python, a similar effect can be achieved using default arguments, variable-length arguments, or by checking the types and number of arguments within the method itself. Here's how we can simulate method overloading in Python:

```python
class MathOperations:
    def add(self, a, b, c=0):
        return a + b + c

math_op = MathOperations()
print(math_op.add(5, 10))  # Output: 15
print(math_op.add(5, 10, 15))  # Output: 30
```

In this example, the add method can take either two or three arguments. By providing a default value for the third parameter, the method can handle both scenarios seamlessly.

Another approach to achieve method overloading in Python is through variable-length arguments using *args and **kwargs:

```python
class MathOperations:
    def add(self, *args):
        return sum(args)

math_op = MathOperations()
print(math_op.add(5, 10))  # Output: 15
print(math_op.add(5, 10, 15))  # Output: 30
```

Here, the add method can accept any number of arguments and sums them up. This flexibility allows the method to handle different numbers of inputs, mimicking the behavior of method overloading.

Method Overriding

Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. This allows the subclass to customize or completely replace the behavior of the inherited method. Method overriding is a key feature in achieving polymorphism and enabling dynamic method dispatch, where the method to be invoked is determined at runtime based on the object's type.

Consider the following example of method overriding:

```python
class Animal:
    def speak(self):
        raise NotImplementedError("Subclasses must implement this
    method")

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

animals = [Dog(), Cat()]

for animal in animals:
    print(animal.speak())
```

In this example, the Animal class defines a generic speak method that raises an error, indicating that subclasses must implement it. The Dog and Cat classes override the speak method to provide specific implementations. When iterating over the list of animals and calling the speak method, Python dynamically determines the appropriate method to invoke based on the object's actual class, demonstrating polymorphism in action.

Duck Typing

Duck typing is a concept related to dynamic typing, where the type or class of an object is determined by its behavior (methods and properties) rather than its explicit inheritance or interface. The name comes from the saying, "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck." In Python, this means that if an object implements the required methods or properties, it can be used in place of another object, regardless of its actual type.

Duck typing emphasizes an object's capabilities rather than its inheritance hierarchy. Here's an example to illustrate duck typing:

```
class Duck:
    def quack(self):
        return "Quack!"

class Person:
    def quack(self):
        return "I'm pretending to be a duck!"

def make_it_quack(duck):
    return duck.quack()

duck = Duck()
person = Person()

print(make_it_quack(duck))  # Output: Quack!
print(make_it_quack(person))  # Output: I'm pretending to be a duck!
```

In this example, both the Duck and Person classes have a quack method. The make_it_quack function accepts any object and calls its quack method. Due to duck typing, both Duck and Person objects can be passed to the function, and it will work correctly regardless of their actual types.

Duck typing allows for greater flexibility and reduces the need for strict type checking, enabling more generic and reusable code. It aligns well with Python's dynamic nature and is a powerful tool for achieving polymorphism.