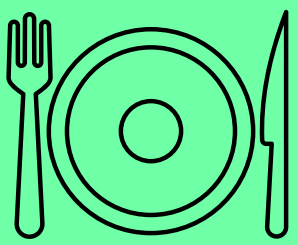
11 Essential Concepts You need to know about OOP in Python

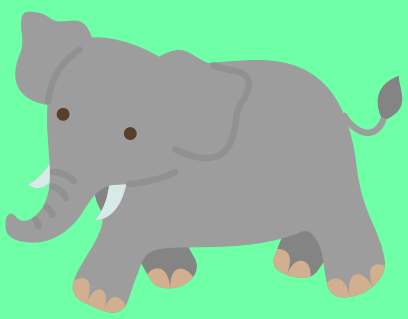
https://www.linkedin.com/in/mattdinhx/

Table of Contents

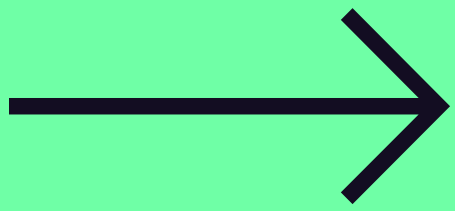
https://www.linkedin.com/in/mattdinhx/

1. Introduction to Object Oriented Programming (OOP)





- Think of OOP as a way to organize a zoo. The zoo has different types of animals (like lions, elephants, snake, giraffe, etc), and there are different actions these animals can do (eat, sleep, make sounds, etc.).
- In OOP, these animals are like objects, and the actions are like methods.



https://www.linkedin.com/in/mattdinhx/

2. Classes and Objects

Class

Characteristics:
color, weight, age, etc

Action:
eat, roar, sleep, etc

Object

Hi, I'm Simba. I'm a lion.

- A class is like a blueprint. For example, consider a "Lion" in the zoo. All lions have common characteristics like color, weight, and age, and common actions like roar, eat, and sleep. So, "Lion" is a class with attributes (color, weight, age) and methods (roar, eat, sleep).
- An object is a specific instance of a class. For example, "Simba" is an object of the "Lion" class.

https://www.linkedin.com/in/mattdinhx/

2. Classes and Objects

- **Code Example:**

```python
# Define a class
class Lion:
    def __init__(self, color, weight, age):
        self.color = color
        self.weight = weight
        self.age = age

    def roar(self):
        return "ROAR!"

    def eat(self):
        return "The lion is eating."

    def sleep(self):
        return "The lion is sleeping."

# Create an object of the Lion class
simba = Lion("golden", 190, 5)
```

https://www.linkedin.com/in/mattdinhx/

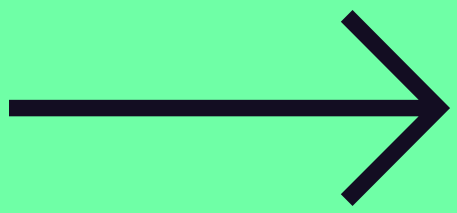
3. Attributes and Methods

```python
# Using attributes
print(simba.color)  # Output: golden
print(simba.weight)  # Output: 190
print(simba.age)  # Output: 5

# Using methods
print(simba.roar())  # Output: ROAR!
print(simba.eat())  # Output: The lion is eating.
print(simba.sleep())  # Output: The lion is sleeping.
```

- **Attributes are the characteristics of the class.** For the "Lion" class, attributes could be color, weight, and age.

- **Methods are the actions that can be performed on the class.** For the "Lion" class, methods could be roar(), eat(), and sleep().



https://www.linkedin.com/in/mattdinhx/

4. Inheritance

Class Lion

Class Cub

I'm a lion but I can "play"

- Inheritance is like creating a specialized version of a class. For example, we could create a "Cub" class that inherits from the "Lion" class.
- It has all the attributes and methods of "Lion", but we could add new ones, like play().

https://www.linkedin.com/in/mattdinhx/

4. Inheritance

- **Code Example**

```python
# Define a new class that inherits from the Lion class
class Cub(Lion):
    def __init__(self, color, weight, age):
        super().__init__(color, weight, age)

    def play(self):
        return "The cub is playing."

# Create an object of the Cub class
nala = Cub("golden", 50, 1)

# Nala can use the methods from the Lion class
print(nala.roar())  # Output: ROAR!

# And also the new method from the Cub class
print(nala.play())  # Output: The cub is playing.
```

https://www.linkedin.com/in/mattdinhx/

5. Polymorphism

I eat meat

I eat plants

- Polymorphism is like being able to use the same method in different ways for different classes. For example, the method eat() could behave slightly differently for a "Lion" object and a "Giraffe " object, reflecting the different diets of these animals.

https://www.linkedin.com/in/mattdinhx/

5. Polymorphism

- Code Example

```python
# Define a new class
class Giraffe:
    def __init__(self, color, weight, age):
        self.color = color
        self.weight = weight
        self.age = age

    def make_sound(self):
        return "HUMMM!"

    def eat(self):
        return "The giraffe is eating leaves."

    def sleep(self):
        return "The giraffe is sleeping."

# Create an object of the Giraffe class
gerry = Giraffe("brown and white", 1200, 10)

# The eat() method behaves differently for the Lion and Giraffe classes
print(simba.eat())  # Output: The lion is eating.
print(gerry.eat())  # Output: The giraffe is eating leaves.
```

https://www.linkedin.com/in/mattdinhx/

6. Encapsulation

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

    def __prepare_food(self):
        # This is a private method that encapsulates the details of food preparation
        return "Zookeepers prepare a delicious meal."

    def eat(self):
        # This is a public method that the visitor can see
        food = self.__prepare_food()  # The visitor doesn't need to know about this
        return f"{self.name} is eating."

simba = Lion("Simba")

# The visitor can see Simba eat
print(simba.eat())  # Output: Simba is eating.

# But the visitor doesn't need to know how the food is prepared
# If they try to call the _prepare_food method, it won't work
try:
    print(simba.__prepare_food())
except AttributeError:
    print("The visitor can't see how the food is prepared.")
```

- **Encapsulation is like keeping the details of the animal's daily routine hidden from the zoo visitors.**
- **For example, a visitor can see a lion eat (call a method), but they don't need to know all the details of how the zookeepers prepare the food (the method's internal workings).**

https://www.linkedin.com/in/mattdinhx/

7. Abstraction

```
# We don't need to know how the make_sound, eat, and sleep methods work
# We just need to know that we can call them
print(simba.make_sound())  # Output: ROAR!
print(simba.eat())  # Output: The lion Simba is eating.
print(simba.sleep())  # Output: The lion Simba is sleeping.

```

- **Abstraction is like providing a simple interface for zoo visitors to interact with the zoo. They don't need to know how the zoo works, they just need to know what actions they can observe (watch a lion eat, hear a lion roar, etc.).**

https://www.linkedin.com/in/mattdinhx/

8. Method Overriding and Super()

```python
class Lion:
    def roar(self):
        return "ROAR!"

class Cub(Lion):
    # Override the roar method
    def roar(self):
        # Use super to call the roar method in the parent class
        parent_roar = super().roar()
        return parent_roar + " But I'm just a cub."

simba = Cub()
print(simba.roar())  # Output: ROAR! But I'm just a cub.

```

- Method Overriding is a feature that allows a subclass to provide a different implementation of a method that is already defined in its superclass. In our example, the Cub class overrides the roar method of the Lion class.
- The super() function is used to call a method from a parent class. In our example, super().roar() calls the roar method of the Lion class from within the Cub class.

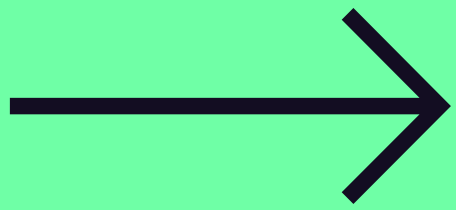
https://www.linkedin.com/in/mattdinhx/

9. Multiple Inheritance

I am a lion

I am a bird

I am a griffin



https://www.linkedin.com/in/mattdinhx/

9. Multiple Inheritance

```python
class Bird:
    def fly(self):
        return "I'm flying!"

# The Griffin class inherits from both Lion and Bird
class Griffin(Lion, Bird):
    pass

gryphon = Griffin()
print(gryphon.roar())  # Output: ROAR!
print(gryphon.fly())  # Output: I'm flying!
```

- Multiple Inheritance is a feature where a class can inherit from more than one classes. This means, a child class can inherit properties and methods from multiple parent classes.
- In our example, the Griffin class inherits from both Lion and Bird classes.

https://www.linkedin.com/in/mattdinhx/

10. Abstract Base Classes

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

class Lion(Animal):
    def make_sound(self):
        return "ROAR!"

simba = Lion()
print(simba.make_sound())  # Output: ROAR!
```

- An abstract base class is a class that cannot be used to create objects (should not be instantiated) and serves as an interface for other classes.
- The abstractmethod decorator specifies that the method must be overridden in any non-abstract child class. In our example, Animal is an abstract base class and make_sound is an abstract method.

https://www.linkedin.com/in/mattdinhx/

11. Class Variables vs. Instance Variables

```python
class Lion:
    # Class variable
    species = "Panthera leo"

    def __init__(self, name):
        # Instance variable
        self.name = name

simba = Lion("Simba")
nala = Lion("Nala")

print(Lion.species)  # Output: Panthera leo
print(simba.name)  # Output: Simba
print(nala.name)  # Output: Nala
```

- Class variables are shared by all instances of a class. They are defined within the class construction. Because they are shared by all instances, they have the same value in every instance. In our example, species is a class variable.

- An instance variable is a variable that is defined inside a method and belongs only to the current instance of a class. In our example, name is an instance variable.

https://www.linkedin.com/in/mattdinhx/

12. Class Methods and Static Methods

```python
class Lion:
    species = "Panthera leo"  # Class variable

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

    @staticmethod
    def calculate_speed(distance, time):
        return distance / time

    @classmethod
    def get_species(cls):
        return cls.species


simba = Lion("Simba")

# Calculate Simba's speed
distance = 100  # meters
time = 10  # seconds
speed = simba.calculate_speed(distance, time)
print(f"Simba's speed: {speed} m/s")  # Output: Simba's speed: 10.0 m/s

# Get the species of the Lion class
species = Lion.get_species()
print(f"Species: {species}")  # Output: Species: Panthera leo
```

- Class methods are methods that are bound to the class and not the instance of the class. They can access and modify the state of the class, which is shared among all instances of the class.
- Static methods are methods that are bound to the class, but unlike class methods, they can't access or modify the class or instance state. They work like regular functions but belong to the class's namespace.

https://www.linkedin.com/in/mattdinhx/

Give these concepts a try and let me know how it goes.

Leave a comment below

https://www.linkedin.com/in/mattdinhx/