Introduction to

PYTHON PROGRAMING

OBJECT-ORIENTED PROGRAMMING(OOP)

OOP, OR **OBJECT-ORIENTED PROGRAMMING**, IS A METHOD OF STRUCTURING A PROGRAM BY BUNDLING RELATED PROPERTIES AND BEHAVIORS IN TO INDIVIDUAL **OBJECTS**.

DEFINE A CLASS

CLASSES VS INSTANCES

CLASSES ARE USED TO CREATE USER-DEFINED DATA STRUCTURES. CLASSES ALSO HAVESPECIAL FUNCTIONS, CALLED **METHODS**,

AN **INSTANCE** IS AN OBJECT BUILT FROM A CLASS THAT CONTAINS REAL DATA.

HOW TO DEFINE A CLASS

ALL CLASS DEFINITIONS START WITH THE CLASS KEYWORD, WHICH IS FOLLOWED BY THE NAME OF THE CLASS AND A COLON.

```
class Dog:
    pass
```

Note

Unlike functions and variables, the convention for naming classes in Python is to use CamelCase notation, starting with a capital letter. For example, a class for a specific breed of a dog, like the Jack Russell Terrier, would be written as `JackRussellTerrier`.

Note

Functions that belong to a class are called **instance methods** because they belong to the instance of a class. For example, `list.append()` and `string.find()` are instance methods.

INSTANTIATE AN OBJECT

TO INSTANTIATE AN OBJECT, TYPE THE NAME OF THE CLASS,IN THE ORIGINAL CAMEL CASE,FOLLOWED BY PARENTHE SES CONTAINING ANY VALUES THAT MUST BE PASSED TO THECLASS'S.__INIT__() METHOD

```
>>> class Dog:
...     pass
...
```

```
>>> Dog()
<__main__.Dog object at 0x106702d30>
```

TO SEE THIS ANOTHER WAY, TYPE THE FOLLOWING:

```
>>> a = Dog()
>>> b = Dog()
>>> a == b
False
```

CLASS AND INSTANCE ATTRIBUTES

```
>>> class Dog:
...     species = "Canis familiaris"
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...
>>> buddy = Dog("Buddy", 9)
>>> miles = Dog("Miles", 4)
```

AFTER THE DOG INSTANCES ARE CREATED, YOU CAN ACCESS THEIR INSTANCE AT TRIBUTES BY USING **DOT NOTATION**:

```
>>> buddy.name
'Buddy'
```

```
>>> buddy.age
9


>>> miles.name
'Miles'
>>> miles.age
4
```

INSTANCE METHODS

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

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

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

    # Another instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

BOTH INSTANCE AND CLASS ATTRIBUTES CAN BE MODIFIED DYNAMICALLY:

```
>>> buddy.age = 10
>>> buddy.age
10


>>> miles.species = "Felis silvestris"
>>> miles.species
'Felis silvestris'
```

REVIEW EXERCISES

1. MODIFY THE DOG CLASS TO INCLUDE A THIRD INSTANCE ATTRIBUTE CALLED COAT_COLOR THAT STORES THE COLOR OF THE DOG'S COAT AS A STRING. STORE YOUR NEW CLASS IN A SCRIPT AND TEST IT OUT BY ADDING THE FOLLOWING CODE AT THE BOTTOM OF THE SCRIPT:

```
philo = Dog("Philo", 5, "brown")
print(f"{philo.name}'s coat is {philo.coat_color}.")
```

The output of your script should be:

```
Philo's coat is brown.
```

2. CREATE A CAR CLASS WITH TWO INSTANCE ATTRIBUTES: .COLOR, WHICH STORES THE NAME OF THE CAR'S COLOR AS A STRING, AND .MILEAGE, WHICH STORES THE NUMBER OF MILES ON THE CAR AS AN INTEGER. THEN INSTANTIATE TWO CAR OBJECTS—A BLUE CAR WITH 20,000 MILES, AND A RED CAR WITH 30,000MILES, AND PRINT OUT THEIR COLORS AND MILEAGE. YOUR OUTPUT SHOULD LOOK LIKE THE FOLLOWING:

```
The blue car has 20,000 miles.
The red car has 30,000 miles.
```

3. MODIFY THE CAR CLASS WITH AN INSTANCE METHOD CALLED . DRIVE() THAT TAKES A NUMBER AS AN ARGUMENT AND ADDS THAT NUMBER TO THE .MILEAGE ATTRIBUTE. TEST THAT YOUR SOLUTION WORKS BY INSTANTIATING A CAR WITH 0 MILES, THEN CALL .DRIVE(100) AND PRINT THE .MILEAGE ATTRIBUTE TO CHECK THAT IT IS SET TO 100.

INHERIT FROM MOTHER CLASSES

NEWLY FORMED CLASSES ARE CALLED **CHILD CLASSES** AND THE CLASSES THAT CHILD CLASSES ARE DERIVED FROM ARE CALLED **PARENT CLASSES.**

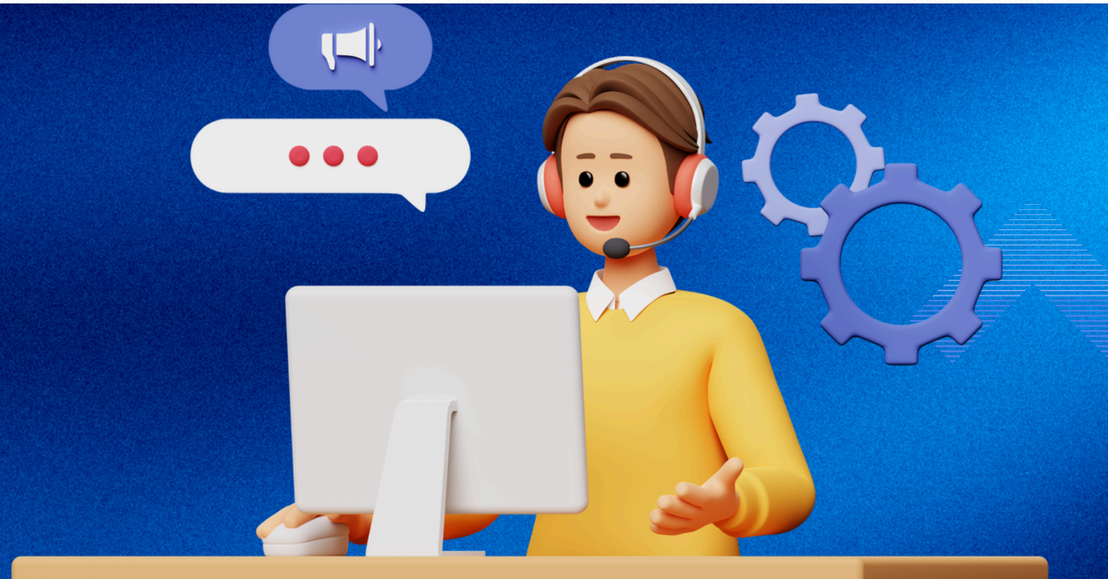
THE *OBJECT* CLASS

```
class Dog(object):
    pass


# In Python 3, this is the same as:


class Dog:
    pass
```

PARENT CLASSES VS CHILDCLASSES

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

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

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

    def speak(self, sound):
        return f"{self.name} says {sound}"
```

```python
class JackRussellTerrier(Dog):
    pass

class Dachshund(Dog):
    pass

class Bulldog(Dog):
    pass
```



EXTENDING THE FUNCTIONALITY OF A PARENT CLASS

```
class JackRussellTerrier(Dog):
    def speak(self, sound="Arf"):
        return f"{self.name} says {sound}"
```

REVIEW EXERCISES

1. CREATE A GOLDEN RETRIEVER CLASS THAT INHERITS FROM THE DOG CLASS. GIVE THE SOUND ARGUMENT OF THE GOLDEN RETRIEVER. SPEAK() METHOD A DEFAULT VALUE OF "BARK". USE THE FOLLOWING CODE FOR YOUR PARENT DOG CLASS:

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

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

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

    def speak(self, sound):
        return f"{self.name} says {sound}"
```

2. WRITE A RECTANGLE CLASS THAT MUST BE INSTANTIATED WITH TWO AT TRIBUTED: LENGTH AND WIDTH. ADD A .AREA() METHOD TO THE CLASS THAT RETURNS THE AREA (LENGTH * WIDTH) OF THE RECTANGLE. THEN WRITE A SQUARE CLASS THAT INHERITS FROM THE RECTANGLE CLASS AND THAT IS INSTANTIATED WITH A SINGLE ATTRIBUTE CALLED SIDE_LENGTH. TEST YOUR SQUARE CLASS BY INSTANTIATING A SQUARE WITH A SIDE_LENGTH OF 4. CALLING THE .AREA() METHOD SHOULD RETURN 16

CHALLENGE: MODEL A FARM

1. YOU SHOULD HAVE AT LEAST FOUR CLASSES: THE PARENT ANIMAL CLASS, AND THEN AT LEAST THREE CHILD ANIMAL CLASSES THAT INHERIT FROM ANIMAL.

2. EACH CLASS SHOULD HAVE A FEW ATTRIBUTES AND AT LEAST ONE METHOD THAT MODELS SOME BEHAVIOR APPROPRIATE FOR A SPECIFIC ANIMAL OR ALL ANIMALS—SUCH AS WALKING, RUNNING, EATING, SLEEPING, AND SO ON.

3. KEEP IT SIMPLE. UTILIZE INHERITANCE. MAKE SURE YOU OUTPUT DETAILS ABOUT THE ANIMALS AND THEIR BEHAVIORS.

THANK YOU