Python Summer Training

Object-Oriented Programming

Python Team, BFCAI

Modules

- As our program grows bigger, it may contain many lines of code.
- Instead of putting everything in a single file, we can use modules to separate codes in separate files as per their functionality.
- This makes our code organized and easier to maintain.
- Module is a file that contains code to perform a specific task.
- A module may contain variables, functions, classes etc.
- We can import modules with the statement:

```
import module_name
```

- Then accessed their features via each module's name and a dot (.).

Modules: Math Module

- Python has a set of built-in math functions, including an extensive `math` module, that allows you to perform mathematical tasks on numbers.
- You can import the Python math module using the following command:

```
import math
```

```
print(math.pi)                  # 3.141592653589793
print(math.log2(1024))          # 10.0
print(math.sqrt(100))           # 10.0
print(math.pow(2, 7))           # 128.0
print(math.factorial(5))        # 120
```

Modules: Math Module

Function	Description
ceil(x)	Returns the smallest integer greater than or equal to x.
fabs(x)	Returns the absolute value of x
factorial(x)	Returns the factorial of x
floor(x)	Returns the largest integer less than or equal to x
fmod(x, y)	Returns the remainder when x is divided by y
isfinite(x)	Returns True if x is neither an infinity nor a NaN (Not a Number)
isinf(x)	Returns True if x is a positive or negative infinity
isnan(x)	Returns True if x is a NaN
modf(x)	Returns the fractional and integer parts of x

Modules: Math Module

exp(x)	Returns e**x
log(x[, b])	Returns the logarithm of `x` to the base `b` (defaults to e)
log2(x)	Returns the base-2 logarithm of x
log10(x)	Returns the base-10 logarithm of x
pow(x, y)	Returns x raised to the power y
sqrt(x)	Returns the square root of x
acos(x)	Returns the arc cosine of x
asin(x)	Returns the arc sine of x
atan(x)	Returns the arc tangent of x
atan2(y, x)	Returns atan(y / x)

Modules: Math Module

cos(x)	Returns the cosine of x
sin(x)	Returns the sine of x
tan(x)	Returns the tangent of x
degrees(x)	Converts angle x from radians to degrees
radians(x)	Converts angle x from degrees to radians
cosh(x)	Returns the hyperbolic cosine of x
sinh(x)	Returns the hyperbolic cosine of x
tanh(x)	Returns the hyperbolic tangent of x
pi	Mathematical constant, the ratio of circumference of a circle to it's diameter (3.14159...)
e	mathematical constant e (2.71828...)

Modules: Importing Multiple Identifiers from a Module

- Using the `from … import …` statement you can import a comma-separated list of identifiers from a module then use them in your code without having to precede them with the module name and a dot (.):

```
from math import pi

print(pi)
# Output: 3.141592653589793
```

Modules: Importing Multiple Identifiers from a Module

- Using the `from … import …` statement you can import a comma-separated list of identifiers from a module then use them in your code without having to precede them with the module name and a dot (.):

```
from math import pi, log2, sqrt, pow, factorial

print(pi)                    # 3.141592653589793
print(log2(1024))            # 10.0
print(sqrt(100))             # 10.0
print(pow(2, 7))             # 128.0
print(factorial(5))          # 120
```

Modules: Avoid Wildcard Imports

- You can import all identifiers defined in a module with a wildcard import of the form

  ```
  from module_name import *
  ```

- This makes all of the module's identifiers available for use in your code. Importing a module's identifiers with a wildcard import can lead to errors.
- It's considered a dangerous practice that you should avoid.

Modules: Avoid Wildcard Imports

- Consider the following snippets:

```
from math import *


print(pi)
# Output: 3.141592653589793


pi = 'Raspberry Pi'
print(pi)
# Output: Raspberry Pi
```

Modules: Avoid Wildcard Imports

- Consider the following snippets:

```
pi = 'Raspberry Pi'
print(pi)
# Output: Raspberry Pi


from math import *
print(pi)
# Output: 3.141592653589793
```

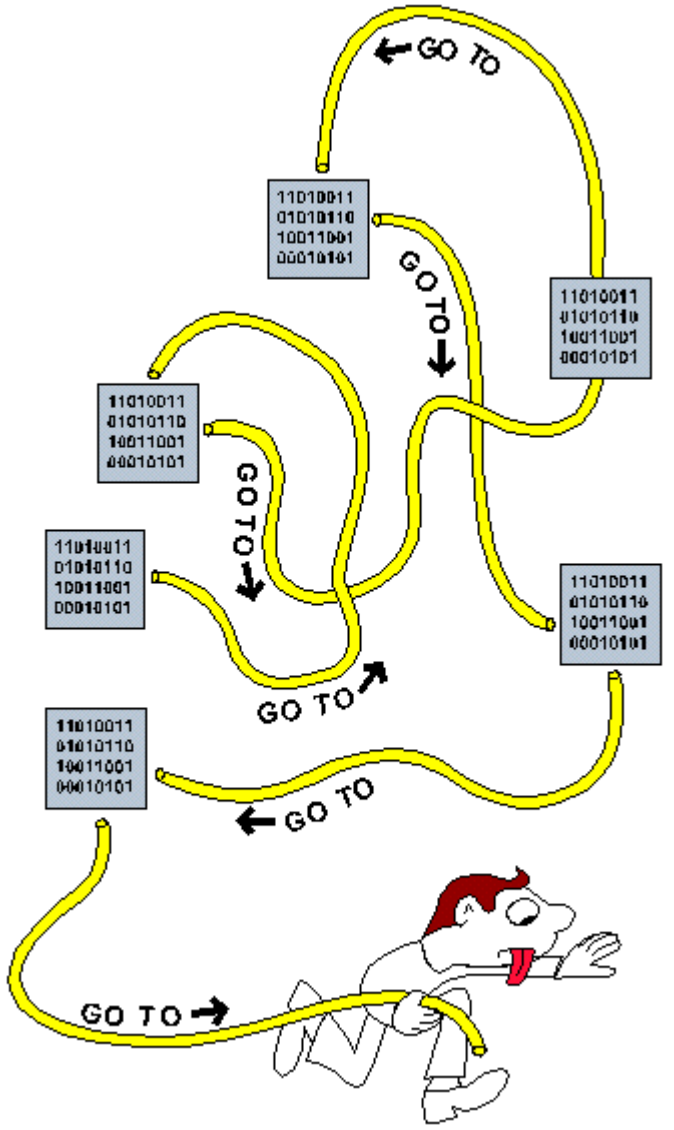
Modules: Binding Names for Modules

- Sometimes it's helpful to import a module and use an abbreviation for it to simplify your code.
- The import statement's `as` clause allows you to specify the name used to reference the module's identifiers.

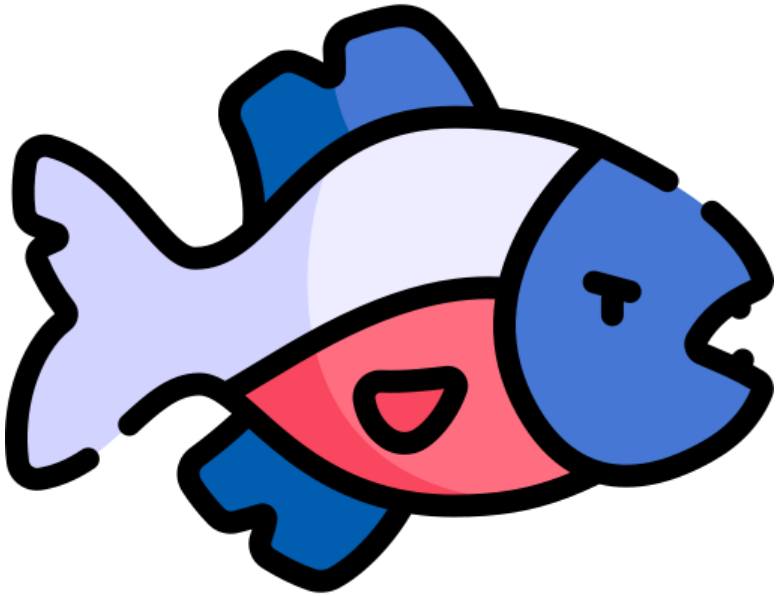
```
import math as m
print(m.pi)
# Output: 3.141592653589793
```

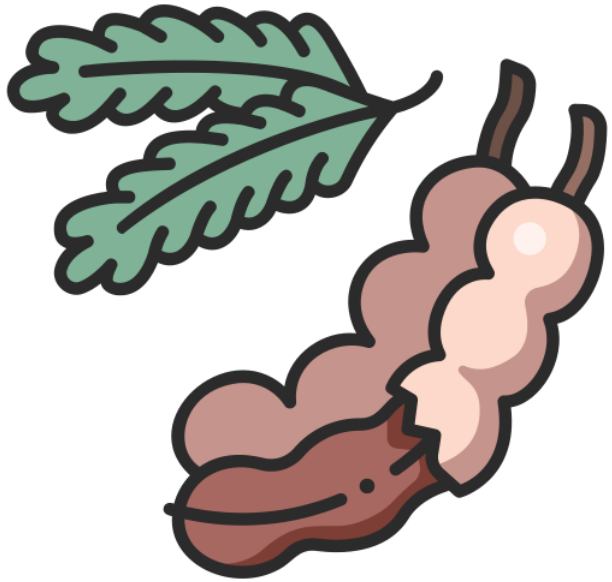
Speghatti Code Level

```
import math

radius = 5

area = math.pi * radius**2
circ = 2 * math.pi * radius

print('Area:', area)
print('Circ:', circ)
```



Speghatti Code Level





Procedural Level

```
import math

def area(radius):
    return math.pi * radius**2

def circ(radius):
    return 2 * math.pi * radius

def print_circle(radius):
    print('Area:', area(radius))
    print('Circ:', circ(radius))

radius = 5
print_circle(radius)
```

Modular Level

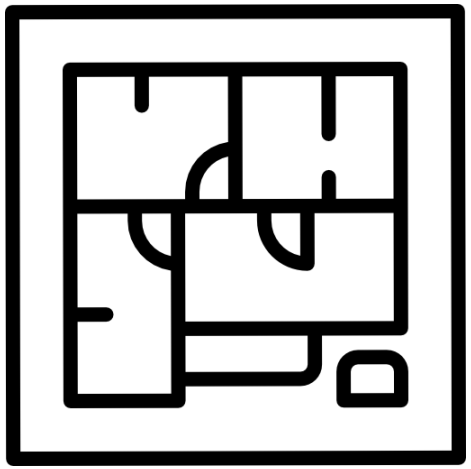
```
import circle

radius = 5
circle.print_circle(radius)
```

```
# circle.py

import math

def area(radius):
    return math.pi * radius**2

def circ(radius):
    return 2 * math.pi * radius

def print_circle(radius):
    print('Area:', area(radius))
    print('Circ:', circ(radius))
```

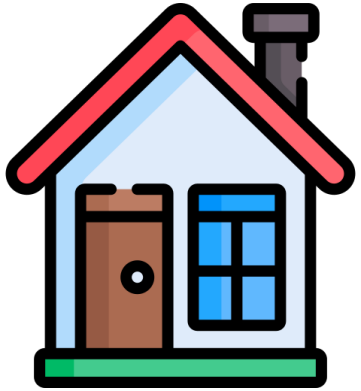
OOP Level

```
import math

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

    def area(self):
        return math.pi * self.__radius**2

    def circ(self):
        return 2 * math.pi * self.__radius

    def print_circle(self):
        print('Area:', self.area())
        print('Circ:', self.circ())

circle = Circle(5)
circle.print_circle()
```

Blueprint

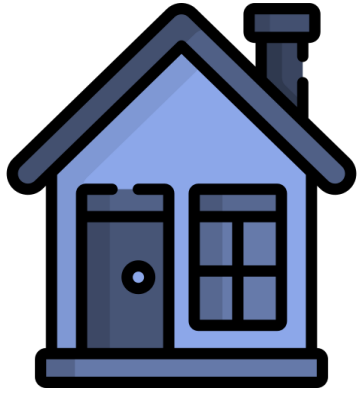


Class



Object

Object

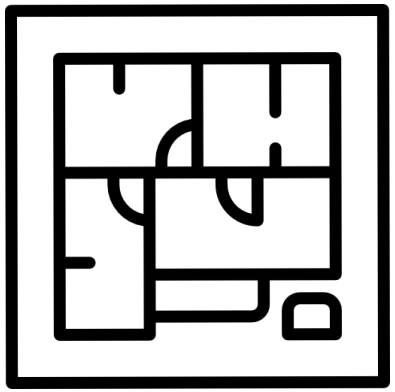


Object

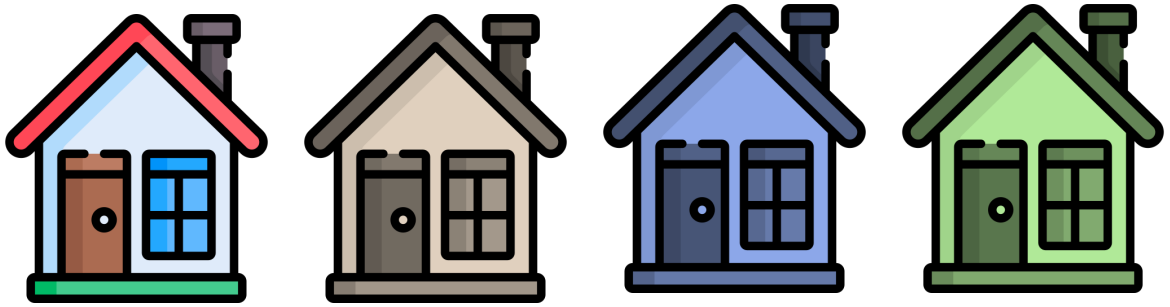
Object

Blueprint

- Think of a class as a "blueprint" from which objects may be created.
- We use the blueprint to build an actual house.
- We could say we are building an instance of the house described by the blueprint.
- We can build several identical houses from the same blueprint.



`Class`

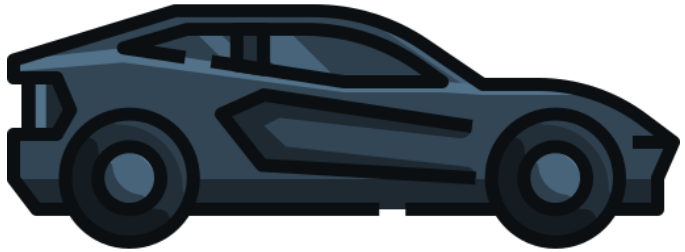


`Objects`

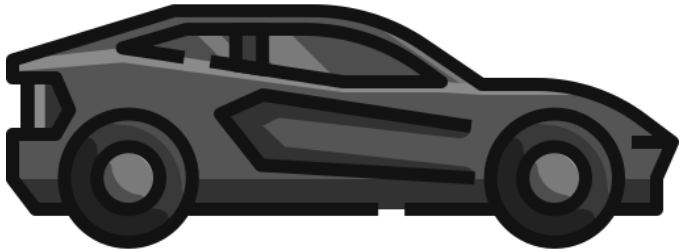
Class and Objects

Class

Object



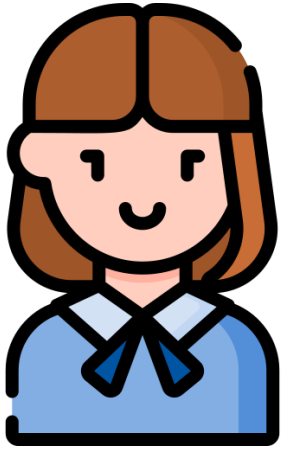
Object



Object

Class and Objects

Female Class



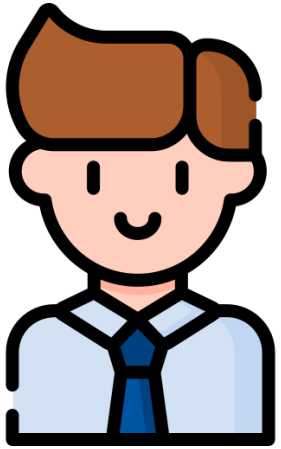
Noura

Mona

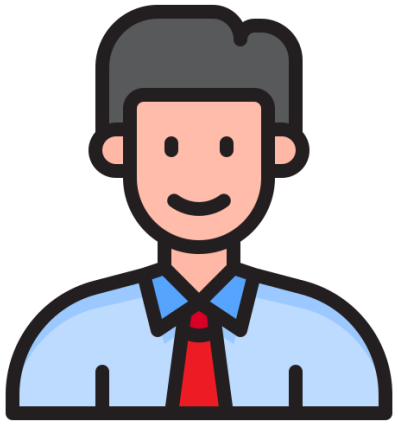
Fatma

Class and Objects

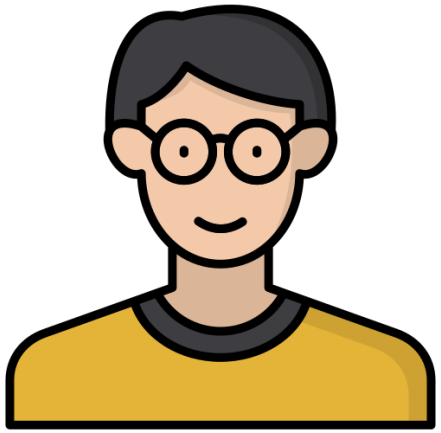
Male Class



Abdallah



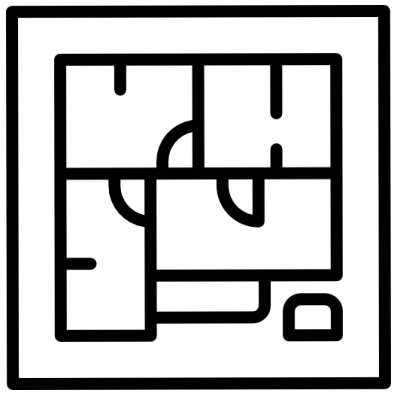
Ahmed



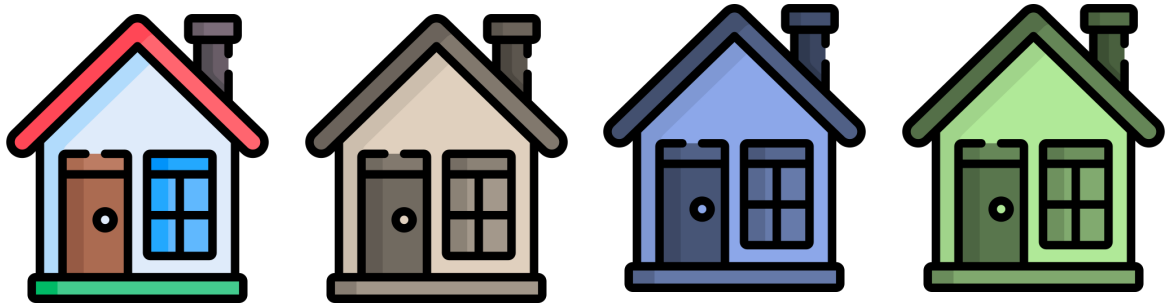
Kareem

Class and Objects

- A Class is the design or blueprint of any entity, which defines the core properties and functions.
- An Object is an instance of a class which has physical existence.



Class

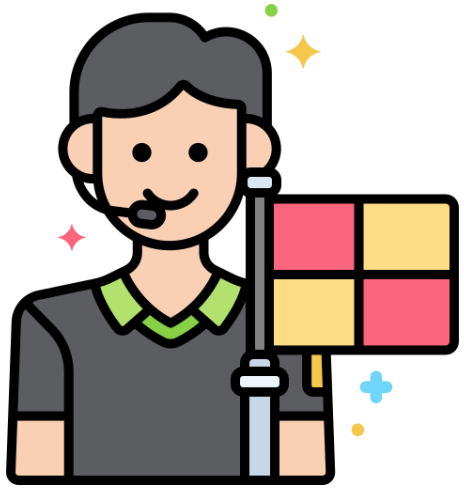


Objects

Football Game

Player

Ball



Referee

Stadium

Football Game: Player Class

Player Class
name number power position height
run() jump() pass() shoot()

Object #1
name: Zidane number: 5 power: 95 position: AMF height: 185
run() jump() pass() shoot()

Object #2
name: Benzema number: 9 power: 90 position: CF height: 185
run() jump() pass() shoot()

Object-Oriented Programming

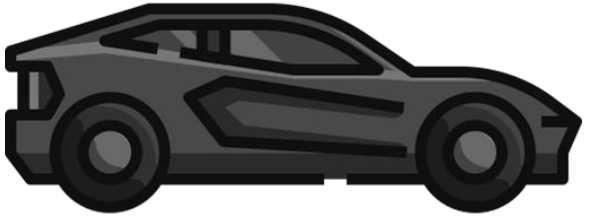
- Procedural (structured) programming is centered on creating procedures (functions).
- Object-oriented programming (OOP) is centered on creating objects.
- An object is a software entity that contains both data and procedures.
- An object is a collection of data with associated behaviors.
- The data contained in an object is known as data attributes.
- An object's data attributes are simply variables that reference data.
- An object's methods are functions that perform operations on the object's data attributes.

Car Class

Car Class
make speed color
accelerate() brake() stop()

Object: car1
make: KIA speed: 100 color: red
accelerate() brake() stop()

Object: car2
make: BMW speed: 200 color: black
accelerate() brake() stop()



Car Class: Class Definition

- The class definition starts with the `class` keyword.
- This is followed by a name identifying the class and is terminated with a colon.
- Since our first class doesn't actually add any data or behaviors, we simply use the `pass` keyword

```
class Car:
    pass
```

Car Class

Car Class: Creating Objects

- An Object is an instance of a class.

```
class Car:
    pass
```

Car Class

```
car1 = Car()
car2 = Car()
```

car1

car2

Car Class: Attributes

```
class Car:
    def __init__(self, make, speed, color):
        self.make = make
        self.speed = speed
        self.color = color
```

Car Class
make
speed
color

Car Class

Car Class: Attributes

```
car1 = Car('KIA', 100, 'red')

print(car1.make)
print(car1.speed)
print(car1.color)
```

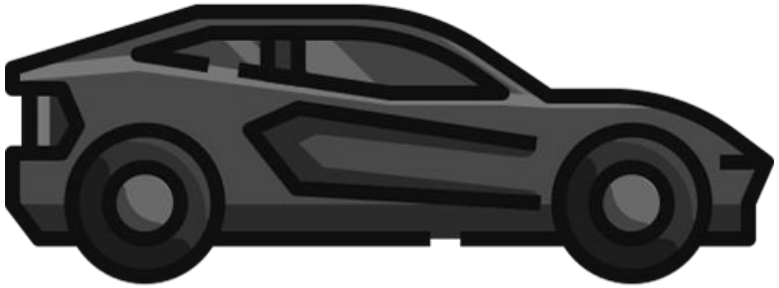
Object: car1
make: KIA
speed: 100
color: red

car1

Car Class: Attributes

```
car2 = Car('BMW', 200, 'black')

print(car2.make)
print(car2.speed)
print(car2.color)
```

Object: car2
make: BMW
speed: 200
color: black



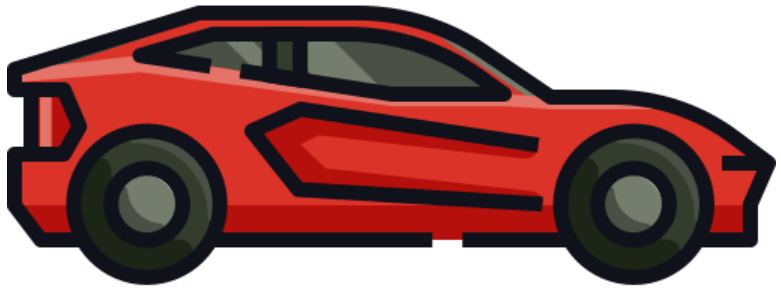
car2

Car Class: Methods

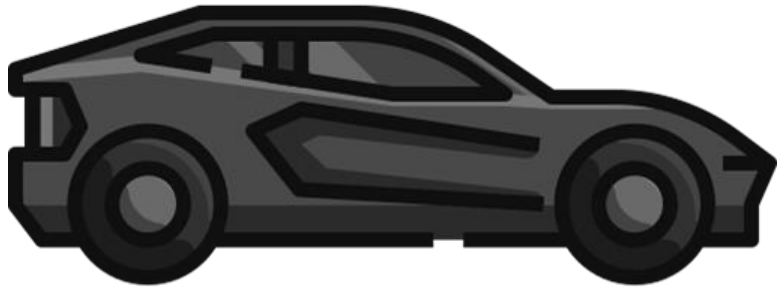
```
class Car:
    def __init__(self, make, speed, color):
        self.make = make
        self.speed = speed
        self.color = color

    def print_data(self):
        print('Make:', self.make)
        print('Speed:', self.speed)
        print('Color:', self.color)
```

Car Class: Methods

```
car1 = Car('KIA', 100, 'red')
car2 = Car('BMW', 200, 'black')

car1.print_data()
car2.print_data()
```



car1



car2

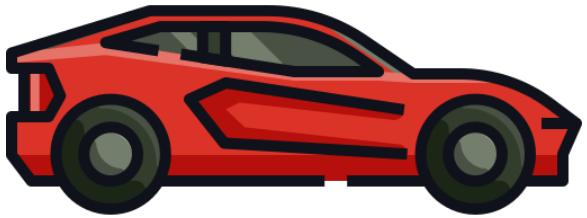
`self` Parameter

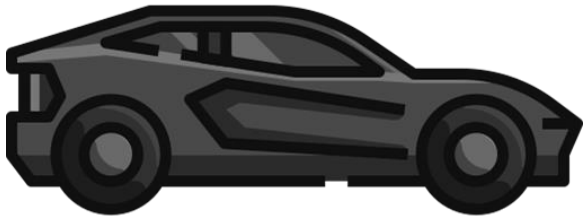
- The `self` parameter is required in every method of a class.
- Each instance of a class has its own set of data attributes.
- A method operates on a specific object's data attributes.
- When a method executes, it must have a way of knowing which object's data attributes it is supposed to operate on.
- That's where the `self` parameter comes in.
- When a method is called, Python makes the self parameter reference the specific object that the method is supposed to operate on.

self Parameter

Object: car1
make: KIA speed: 100 color: red
accelerate() brake() stop()

Object: car2
make: BMW speed: 200 color: black
accelerate() brake() stop()





Initializer Method

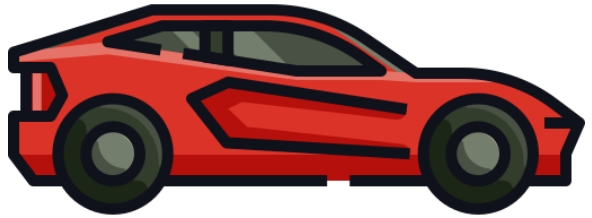
- Most Python classes have a special method named `__init__`, which is automatically executed when an instance of the class is created in memory.
- The `__init__` method is commonly known as an initializer method because it initializes the object's data attributes.
- Immediately after an object is created in memory, the `__init__` method executes, and the `self` parameter is automatically assigned the object that was just created.

Car Class: Initializing Objects With Default Values

```
class Car:
    def __init__(self, make='', speed=0, color='red'):
        self.make = make
        self.speed = speed
        self.color = color

    def print_data(self):
        print('Make:', self.make)
        print('Speed:', self.speed)
        print('Color:', self.color)
```

Car Class: Initializing Objects With Default Values

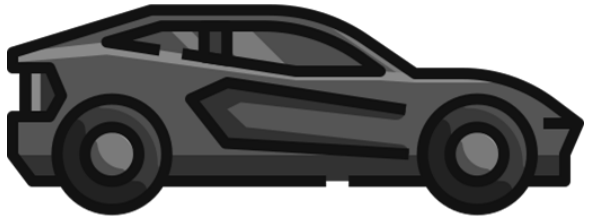
```
car1 = Car()
car1.print_data()


# Output
Make:
Speed: 0
Color: red
```



car1

Car Class: Initializing Objects With Default Values

```
car2 = Car(make='BMW', color='red')
car2.print_data()
```



car2

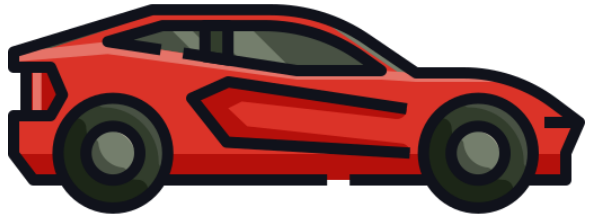
```
# Output
Make: BMW
Speed: 0
Color: red
```

Car Class: Adding More Methods

```
class Car:
    def __init__(self, make, speed, color):
        self.make = make
        self.speed = speed
        self.color = color

    def print_data(self):
        print('Make:', self.make)
        print('Speed:', self.speed)
        print('Color:', self.color)

    def accelerate(self):
        self.speed += 5

    def brake(self):
        self.speed -= 5

    def stop(self):
        self.speed = 0
```

Car Class: Adding More Methods

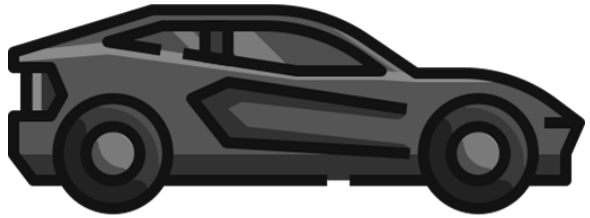
```
car1 = Car('KIA', 100, 'red')
print(car1.speed)          # 100

car1.accelerate()
print(car1.speed)          # 105

car1.brake()
print(car1.speed)          # 100

car1.stop()
print(car1.speed)          # 0
```



car1

Car Class: Adding More Methods

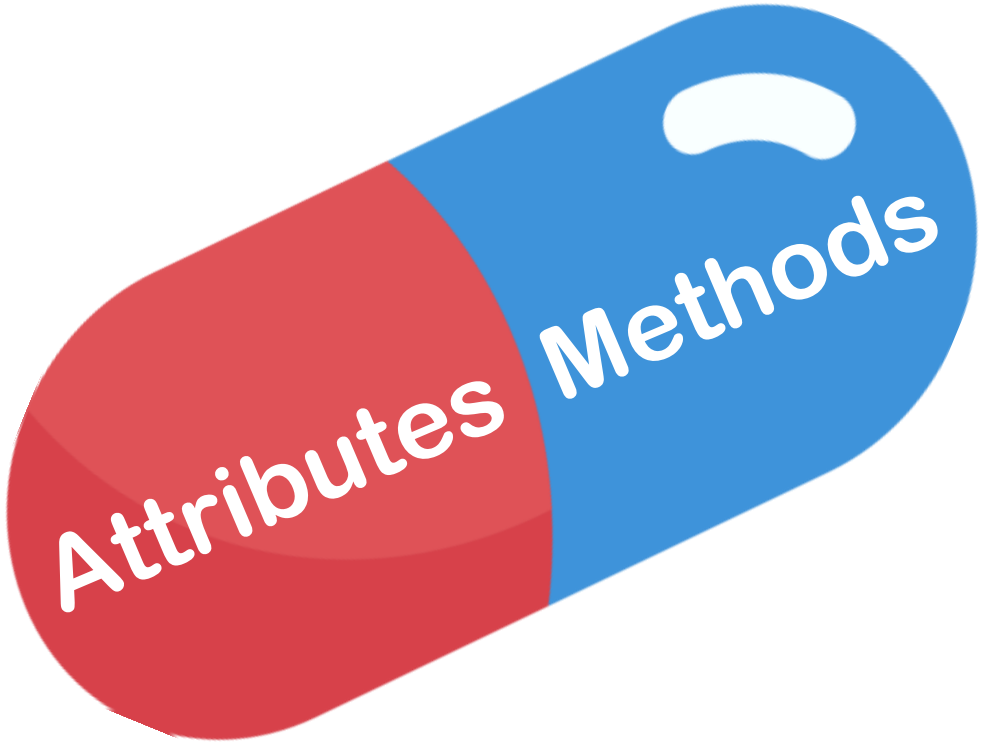
```
car2 = Car('BMW', 200, 'black')

car2.accelerate()
car2.accelerate()
print(car2.speed)       # 210

car2.brake()
print(car2.speed)       # 205

car2.stop()
print(car2.speed)       # 0
```



car2

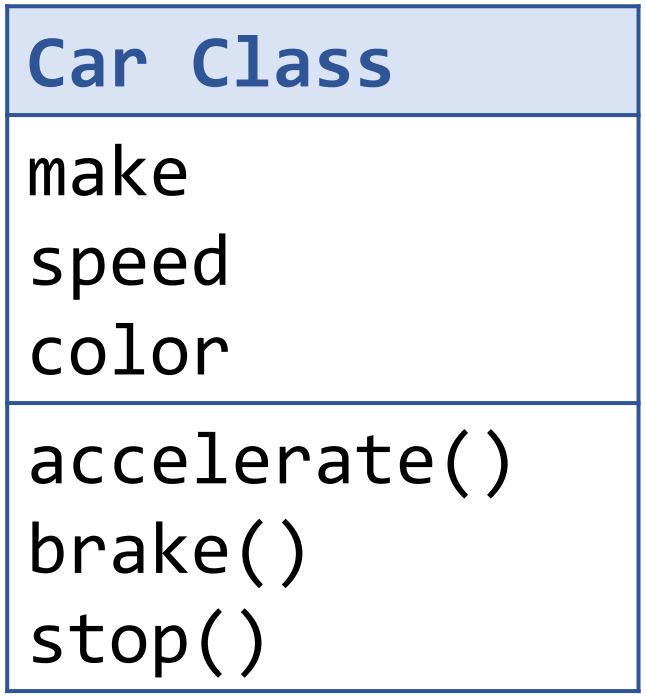
Encapsulation

- Encapsulation refers to the combining of data attributes and methods into a single object, and then controlling access to them.
- An object is a software entity that contains both data and procedures.



Encapsulation

- Encapsulation refers to the combining of data attributes and methods into a single object, and then controlling access to them.
- An object is a software entity that contains both data and procedures.

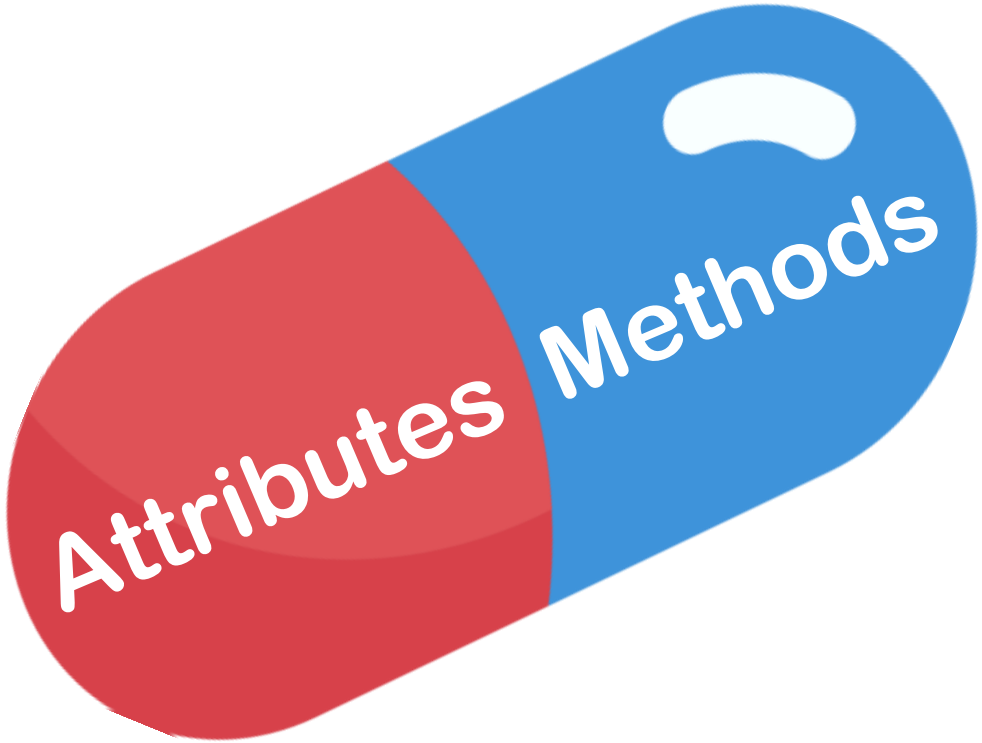


Encapsulation

```
class Car:
    def __init__(self, make, speed, color):
        self.make = make
        self.speed = speed
        self.color = color

    def print_data(self):
        print('Make:', self.make)
        print('Speed:', self.speed)
        print('Color:', self.color)

    def accelerate(self):
        self.speed += 5

    def brake(self):
        self.speed -= 5

    def stop(self):
        self.speed = 0
```

Encapsulation

- Encapsulation refers to the combining of data attributes and methods into a single object, and then controlling access to them.
- An object is a software entity that contains both data and procedures.



Public Attributes

```
class Car:
    def __init__(self, make='', speed=0, color='red'):
        self.make = make
        self.speed = speed
        self.color = color


car1 = Car()
car1.speed = -100

print(car1.speed)   # -100
```

Public Attributes

```
class Car:
    def __init__(self, make='', speed=0, color='red'):
        self.make = make
        self.speed = speed
        self.color = color


car1 = Car()
del car1.speed

print(car1.speed)
'Car' object has no attribute 'speed'
```

Public Attributes

```
import math
print(math.pi)

math.pi = 4
print(math.pi)

# Output
3.141592653589793
4
```

Hiding Attributes

- Object's data attributes should be private, so that only the object's methods can directly access them.
- This protects the object's data attributes from accidental corruption.
- In Python, you can hide an attribute by starting its name with two underscore characters.

```
self.make          # Public Attribute
self.__make        # Private Attribute
```

Hiding Attributes

```
class Car:
    def __init__(self, make='', speed=0, color='red'):
        self.__make = make
        self.__speed = speed
        self.__color = color


car1 = Car()
print(car1.color)
```

```
AttributeError: 'Car' object has no attribute 'color'
```

Hiding Attributes

```
class Car:
    def __init__(self, make='', speed=0, color='red'):
        self.__make = make
        self.__speed = speed
        self.__color = color


car1 = Car()
print(car1.__color)
```

```
AttributeError: 'Car' object has no attribute '__color'
```

Getters and Setters: Public Interfaces

- A common real-world example is the television.
- Our interface to the television is the remote control.
- Each button on the remote control represents a method that can be called on the television object.

Getters and Setters

- It is a common practice to make all of a class's data attributes private, and to provide public methods for accessing and changing those attributes.
- A method that returns a value from a class's attribute but does not change it is known as a getter method.
- A method that stores a value in a data attribute or changes the value of a data attribute in some other way is known as a setter method.

Getters and Setters

```
class Car:
    def __init__(self, make='', speed=0, color='red'):
        self.__make = make
        self.__speed = speed
        self.__color = color

    def get_speed(self):
        return self.__speed

    def set_speed(self, speed):
        if speed > 0:
            self.__speed = speed
        else:
            self.__speed = 0
```

Getters and Setters

```
car1 = Car('KIA', 100, 'red')
print(car1.get_speed())

car1.set_speed(200)
print(car1.get_speed())

car1.set_speed(-200)
print(car1.get_speed())

# Output
100
200
0
```

Getters: @property

```
class Car:
    def __init__(self, make='', speed=0, color='red'):
        self.__make = make
        self.__speed = speed
        self.__color = color

    @property
    def speed(self):
        return self.__speed

car1 = Car('KIA', 100, 'red')
print(car1.speed)

car1.speed = -100
AttributeError: can't set attribute
```

Setters: @attribute.setter

```
class Car:
    def __init__(self, make='', speed=0, color='red'):
        self.__make = make
        self.__speed = speed
        self.__color = color

    @property
    def speed(self):
        return self.__speed

    @speed.setter
    def speed(self, speed):
        if speed > 0:
            self.__speed = speed
        else:
            self.__speed = 0
```

Setters: @attribute.setter

```
car1 = Car('KIA', 100, 'red')
print(car1.speed)

car1.speed = 200
print(car1.speed)

car1.speed = -200
print(car1.speed)

# Output
100
200
0
```

Setters: @attribute.setter

```
del car1.speed

AttributeError: can't delete attribute
```

Abstraction

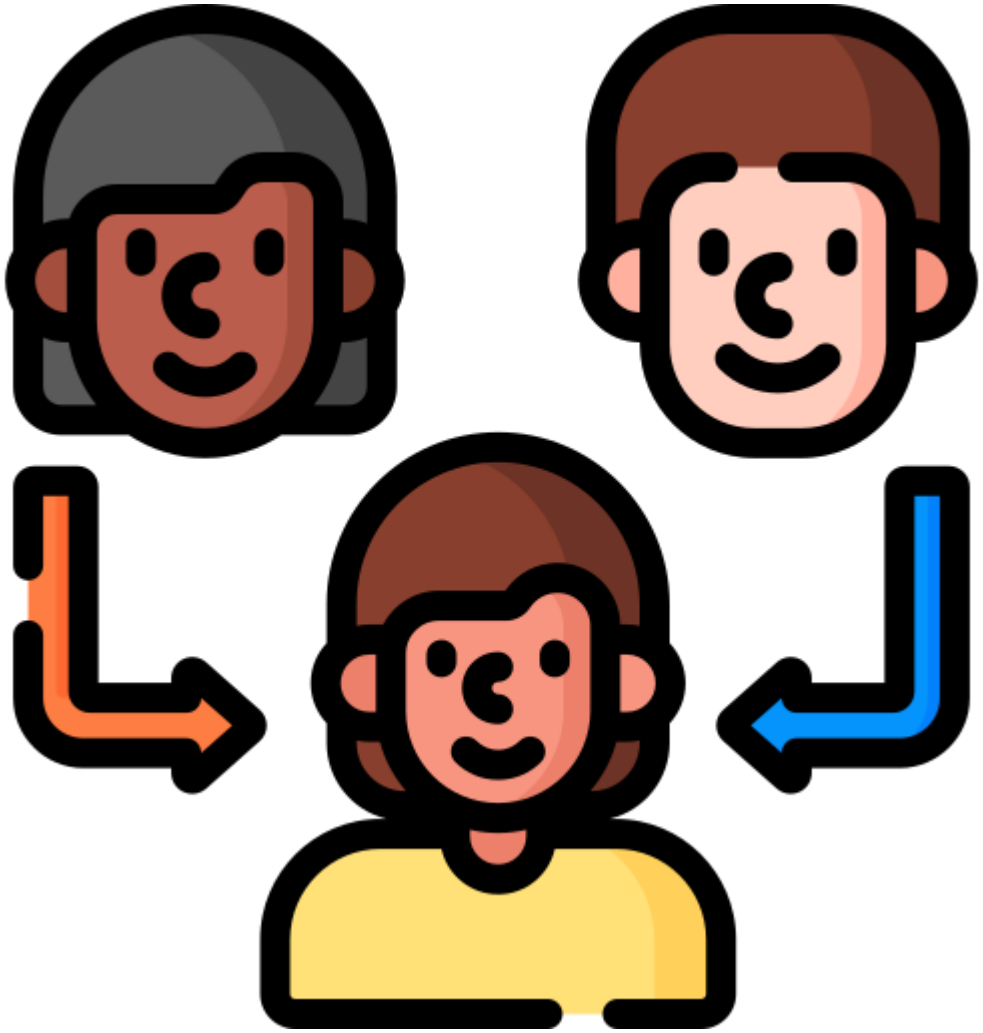
- Abstraction means hiding the details from the outside world.

Abstraction

- Abstraction means hiding the details from the outside world.

```
numbers = [4, 7, 3, 5, 2]

numbers.append(10)      # how?
print(numbers)

numbers.reverse()       # how?
print(numbers)

numbers.sort()          # how?
print(numbers)
```

Inheritance



Inheritance

- In the programming world, duplicate code is considered evil.
- We should not have multiple copies of the same, or similar, code in different places.
- When we fix a bug in one copy and fail to fix the same bug in another copy, we've caused no end of problems for ourselves.
- Inheritance allows a new class to extend an existing class.
- The new class inherits the members of the class it extends.
- Inheritance allows us to create "is-a" relationships between two or more classes.

Single Inheritance: Employee is a Person

Person

Employee

Employee is a Person

Single Inheritance: Employee is a Person

- The superclasses are also called base classes.
- The subclasses are also called derived classes.

Person

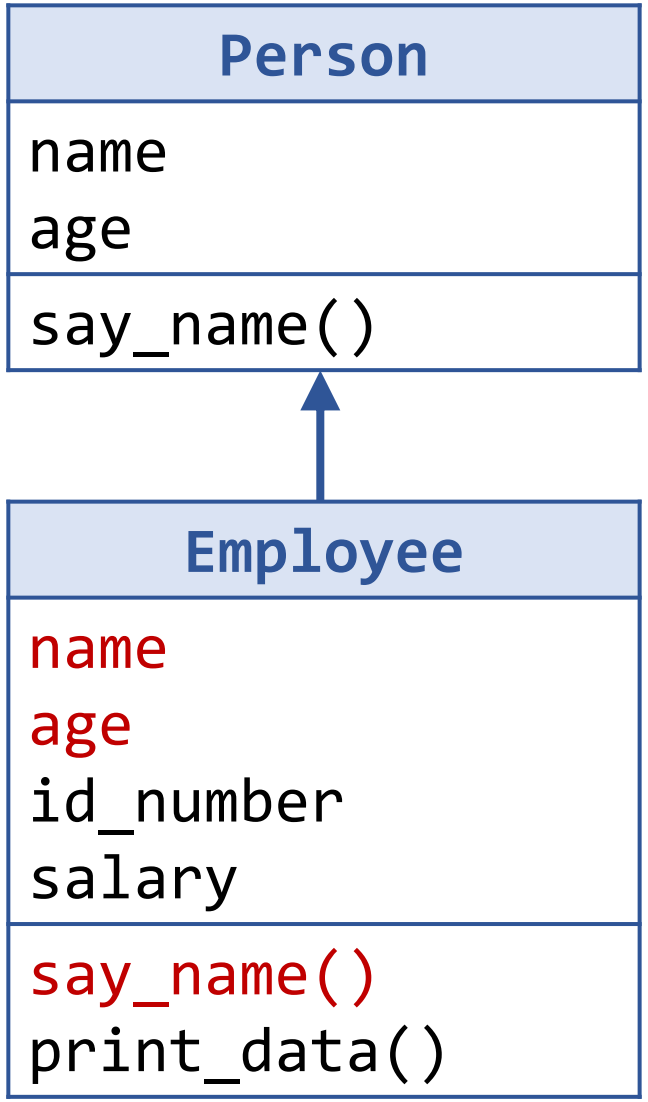
Superclass

Employee is a Person

Employee

Subclass

Single Inheritance: Employee is a Person



Single Inheritance: Employee is a Person

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

    def say_name(self):
        print("I'm", self.name)


class Employee(Person):
    def __init__(self, name, age, id_number, salary):
        super().__init__(name, age)
        self.id_number = id_number
        self.salary = salary

    def print_data(self):
        print(self.name, self.age, self.id_number, self.salary)
```

Single Inheritance: Employee is a Person

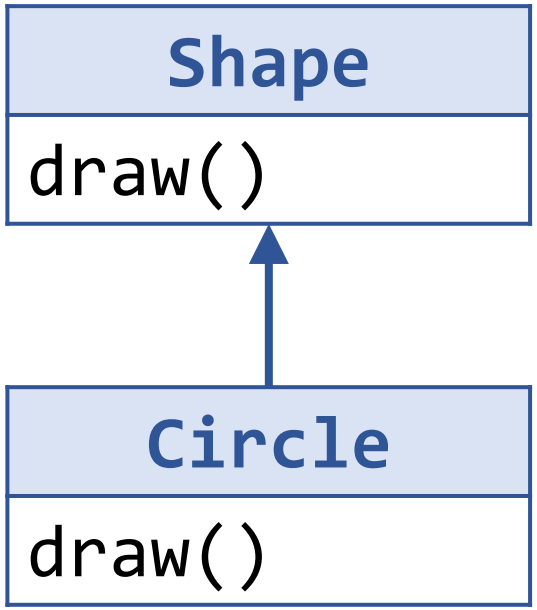
```
emp1 = Employee('Ahmed', 25, 10, 7000)

emp1.say_name()
emp1.print_data()

# Output
I'm Ahmed
Ahmed 25 10 7000
```

Single Inheritance: Employee is a Person

```
emp2 = Employee('Mohamed', 30, 20, 8000)

emp2.name = 'Mohamed Ali'
emp2.age += 1
emp2.salary += 2000
emp2.id_number = 22
emp2.print_data()

# Output
Mohamed Ali 31 22 10000
```

Inheritance: Method Overriding

- So, inheritance is great for adding new behavior to existing classes.
- What about changing behavior?
- Overriding means altering or replacing a method of the superclass with a new method (with the same name) in the subclass.



Inheritance: Method Overriding

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

    def say_name(self):
        print("I'm", self.name)

    def print_data(self):
        print(self.name, self.age)

class Employee(Person):
    def __init__(self, name, age, id_number, salary):
        super().__init__(name, age)
        self.id_number = id_number
        self.salary = salary

    def print_data(self):
        print(self.name, self.age, self.id_number, self.salary)
```

Inheritance: Method Overriding

```
emp3 = Employee('Omar Kareem', 22, 11, 5000)
emp3.print_data()


# Output
Omar Kareem 22 11 5000
```

Single Inheritance

Person

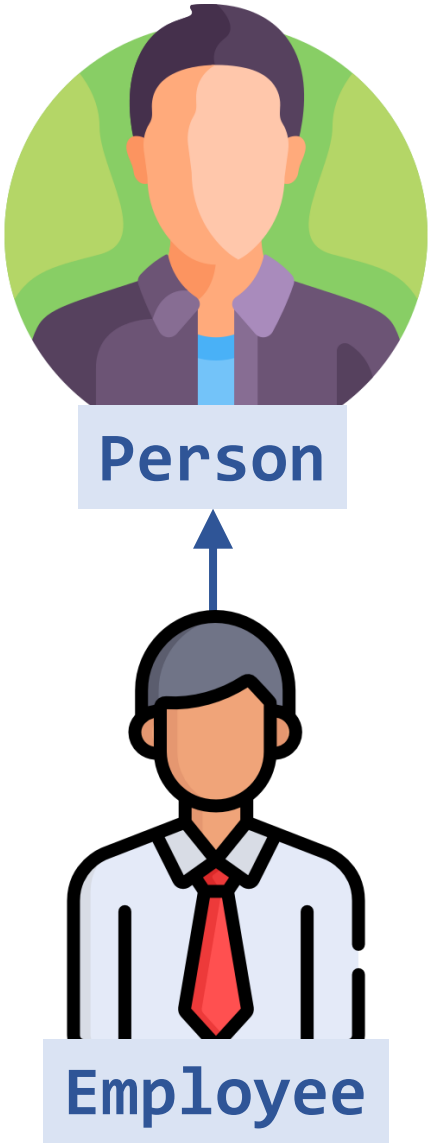
Employee

Employee is a Person

Single Inheritance

```
class Person:
    pass
```

```
class Employee(Person):
    pass
```



Employee is a Person

Multilevel Inheritance

Person

Employee

Employee is a Person

Manager

Manager is an Employee

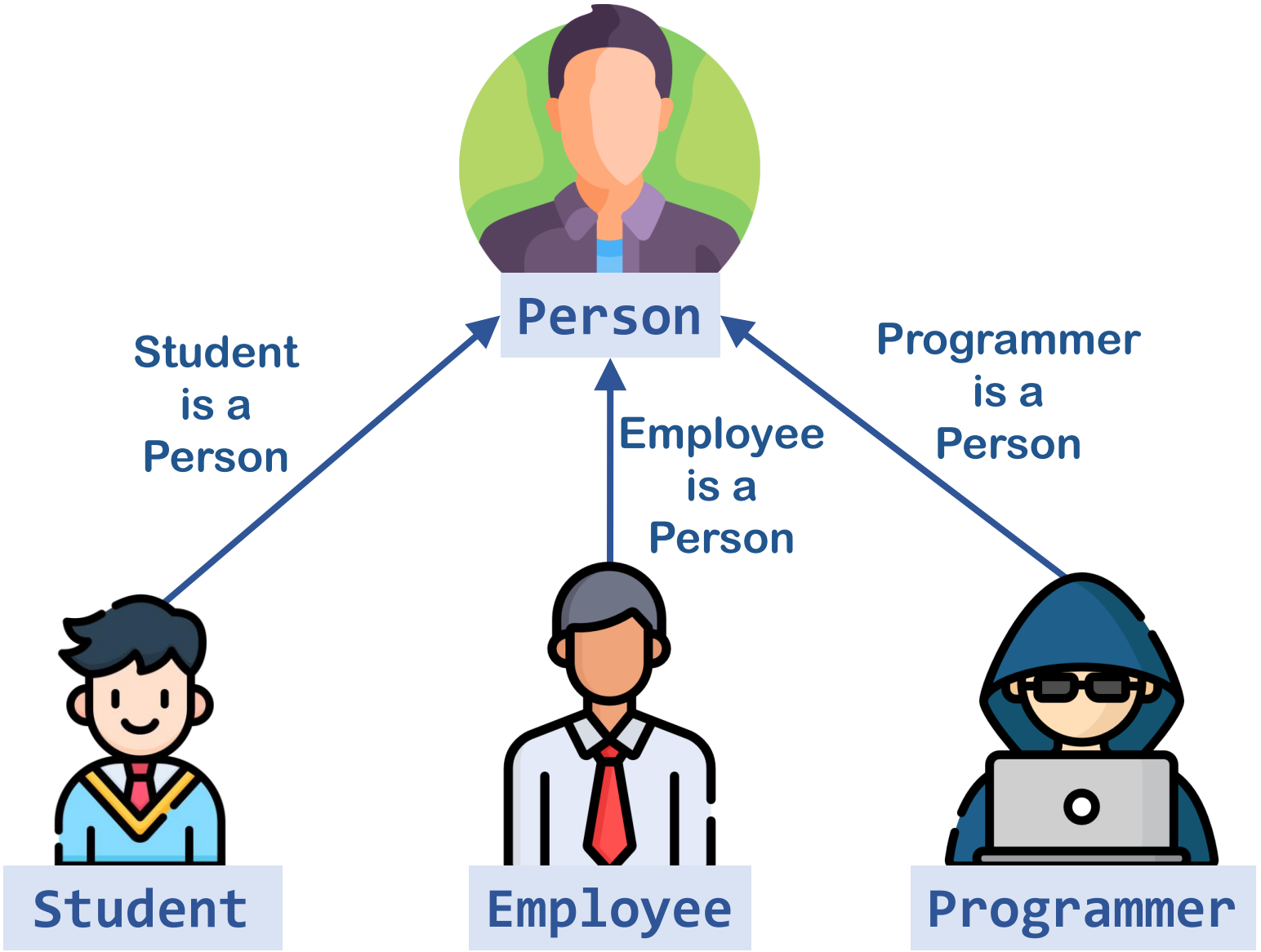
Multilevel Inheritance

```
class Person:
    pass


class Employee(Person):
    pass


class Manager(Employee):
    pass
```

Employee is a Person

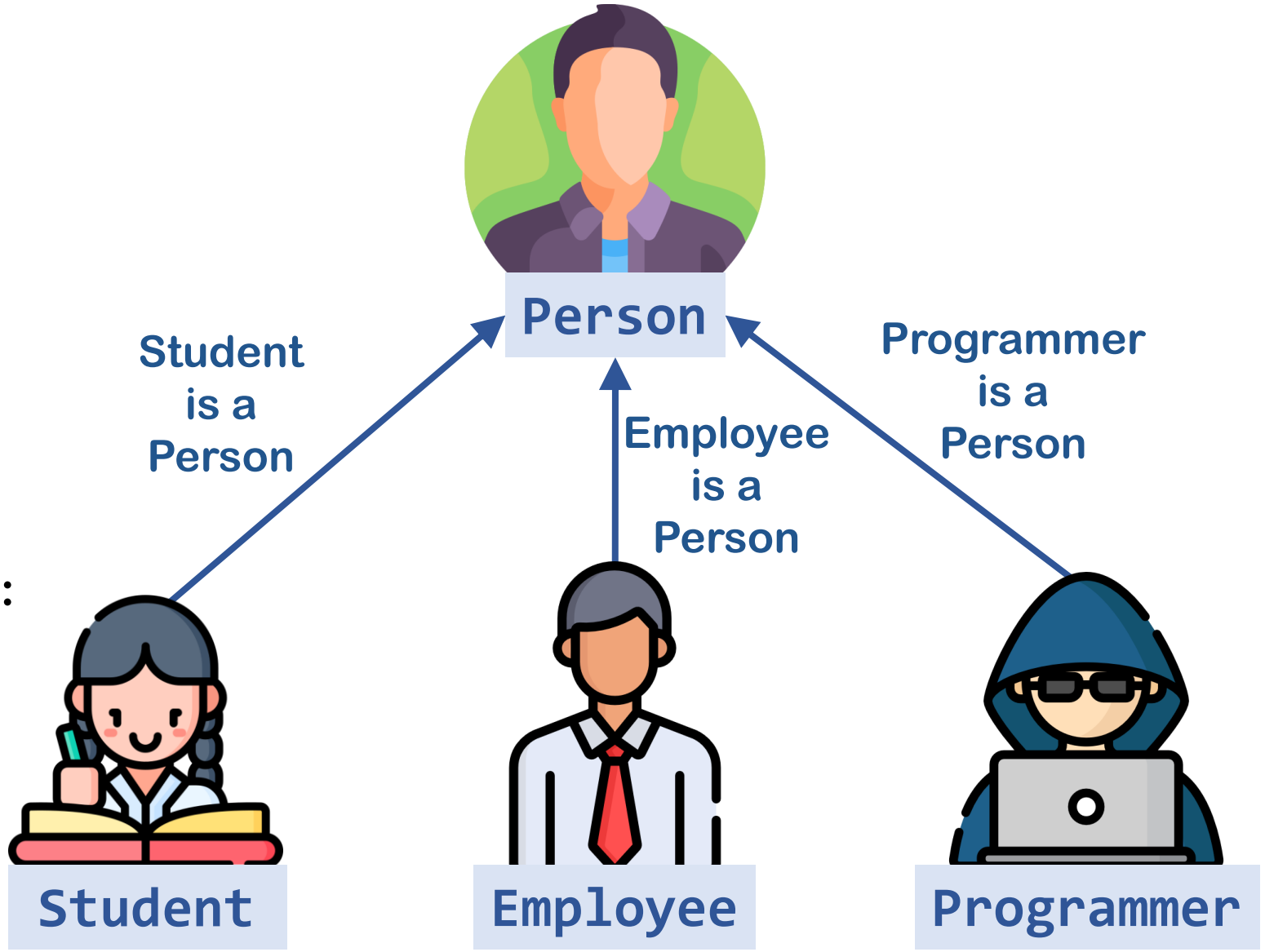
Manager is an Employee

Hierarchical Inheritance

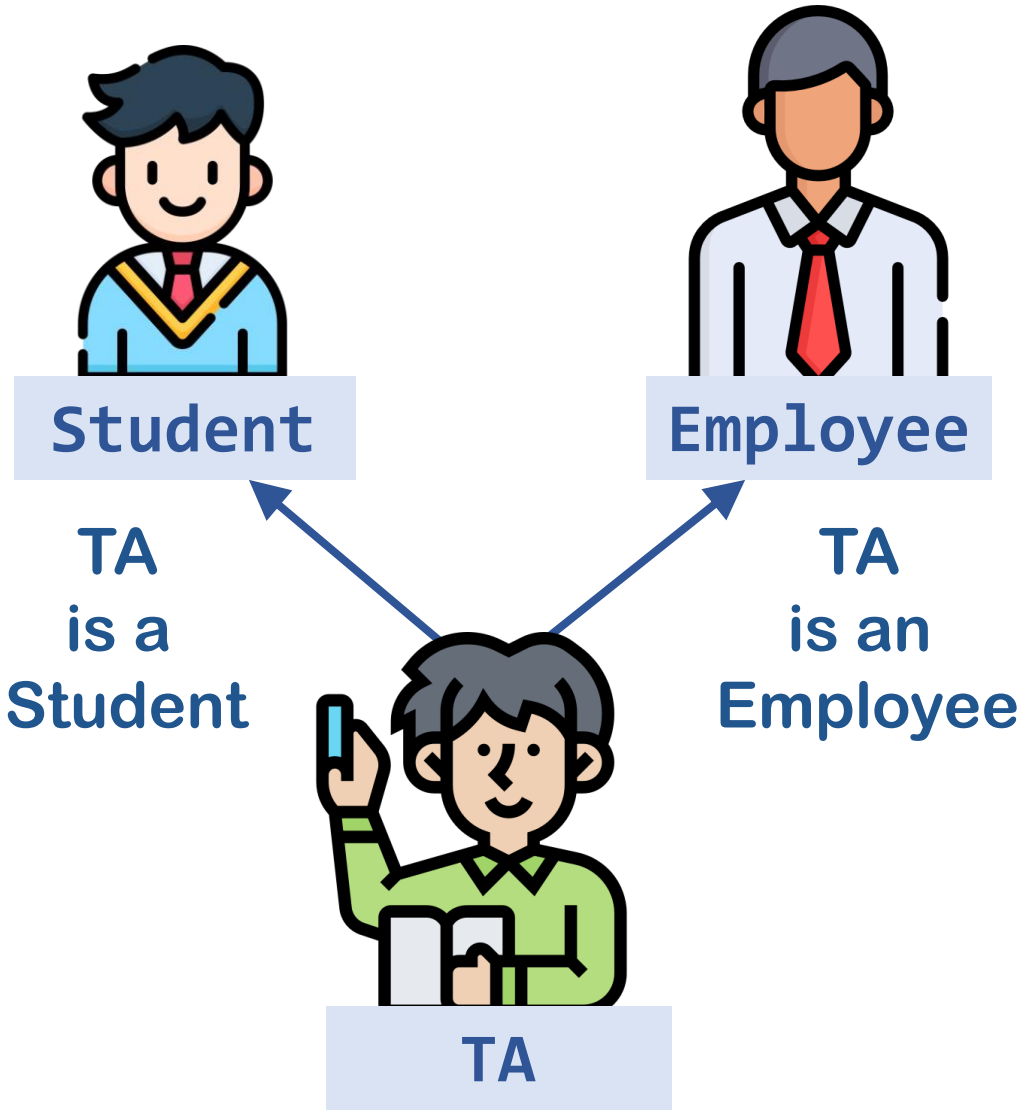


Hierarchical Inheritance

```
class Person:
    pass

class Employee(Person):
    pass

class Student(Person):
    pass

class Programmer(Person):
    pass
```

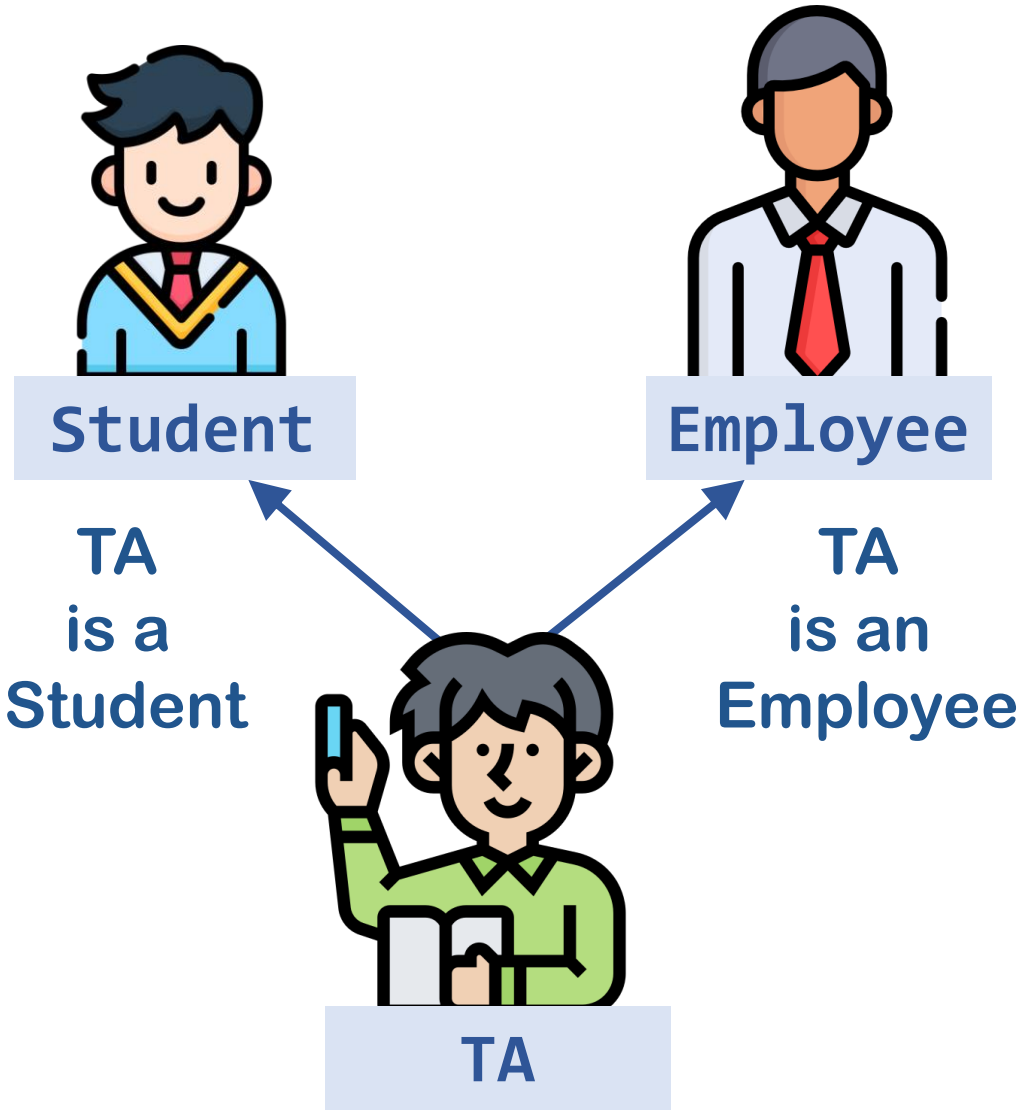


Multiple Inheritance

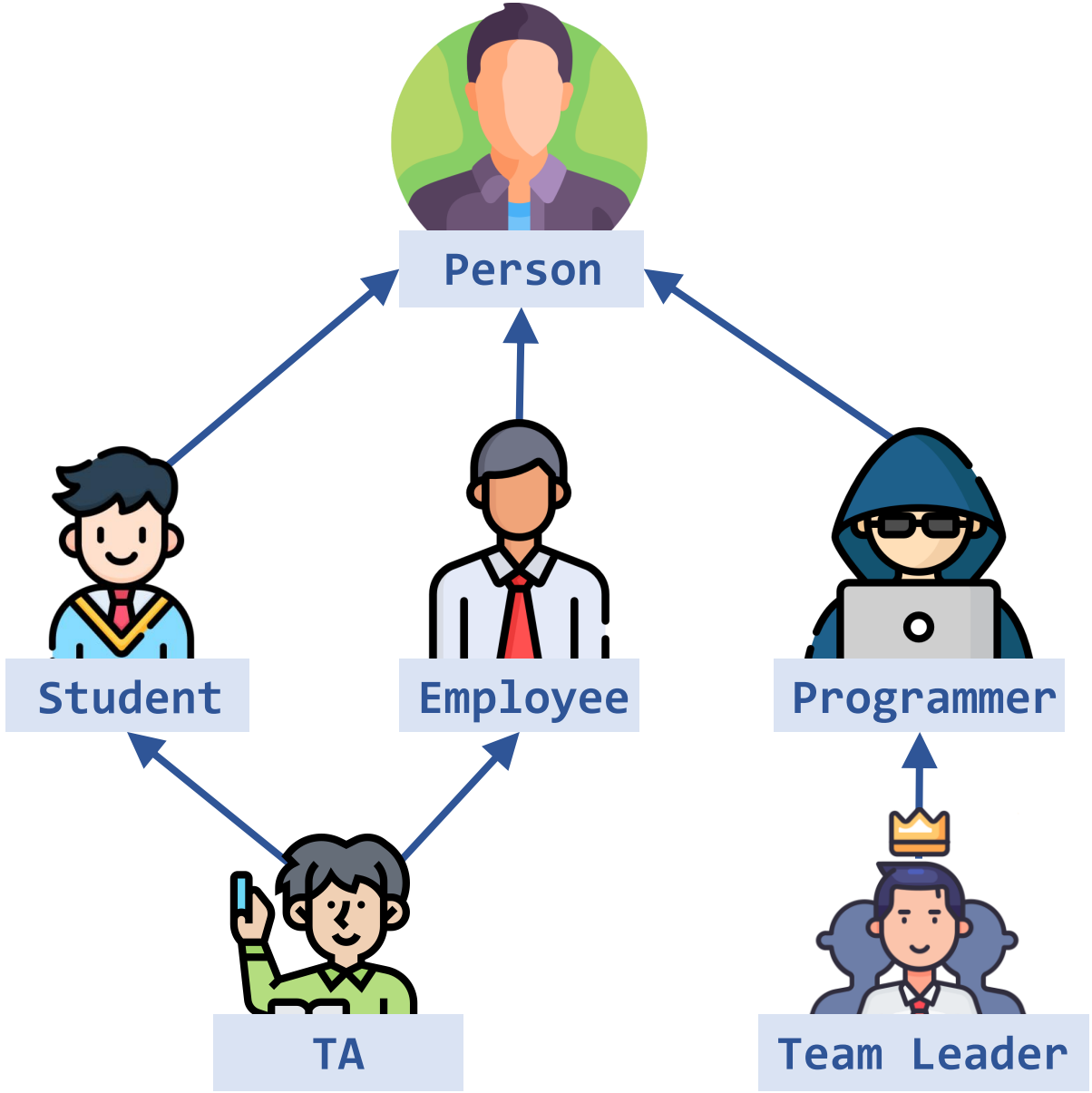


Multiple Inheritance

```
class Student:
    pass

class Employee:
    pass

class TA(Student, Employee):
    pass
```

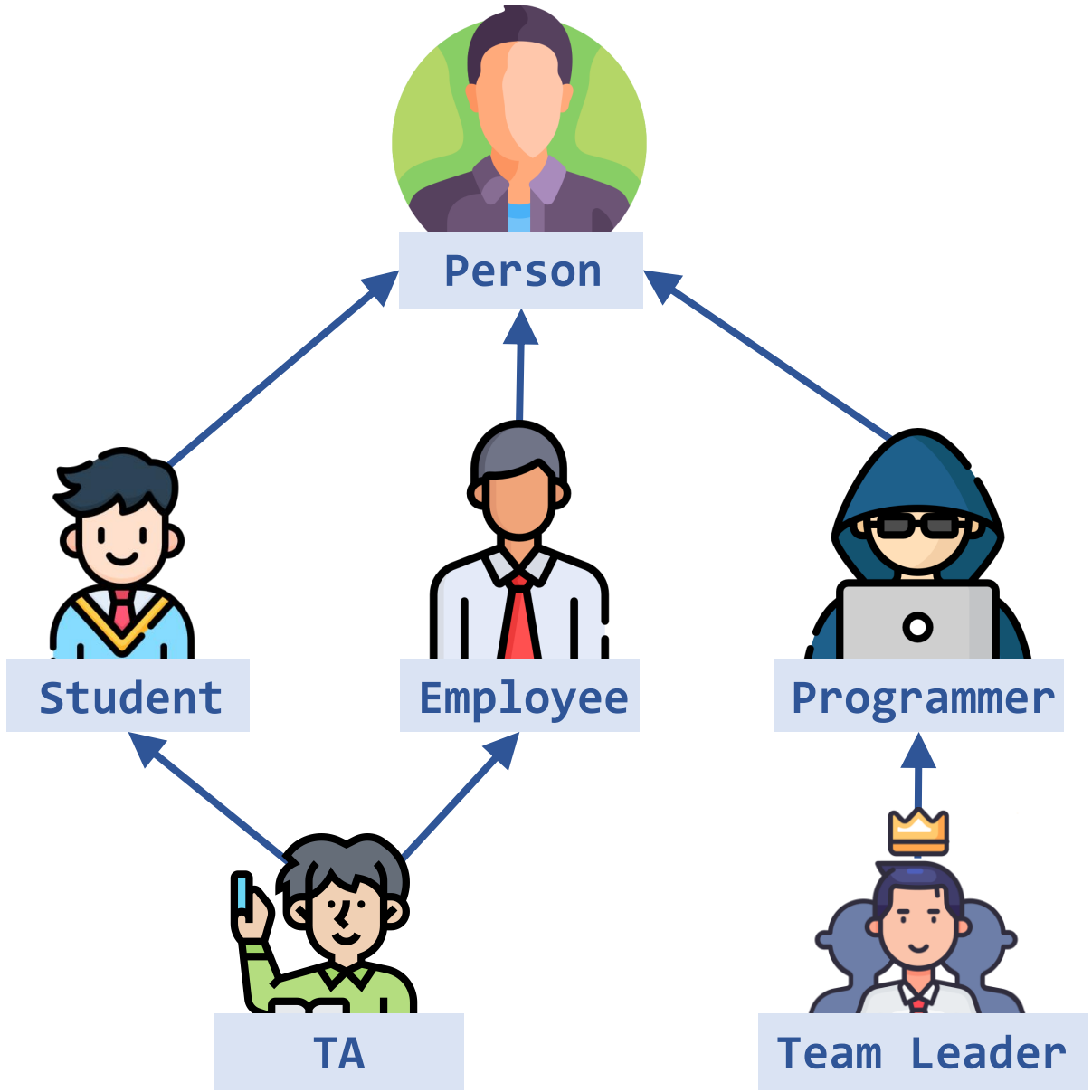


Hybird Inheritance



Hybird Inheritance

```
class Person:
    pass

class Employee(Person):
    pass

class Student(Person):
    pass

class Programmer(Person):
    pass

class TA(Student, Employee):
    pass

class Team Leader(Programmer):
    pass
```

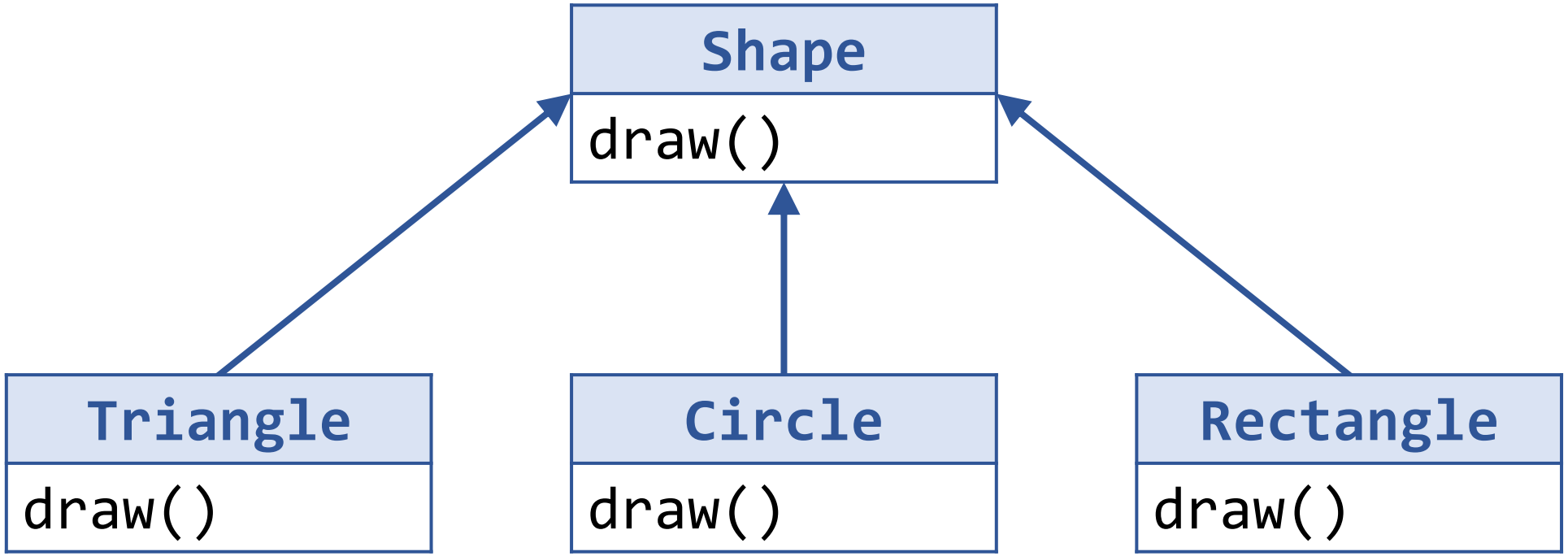


Polymorphism

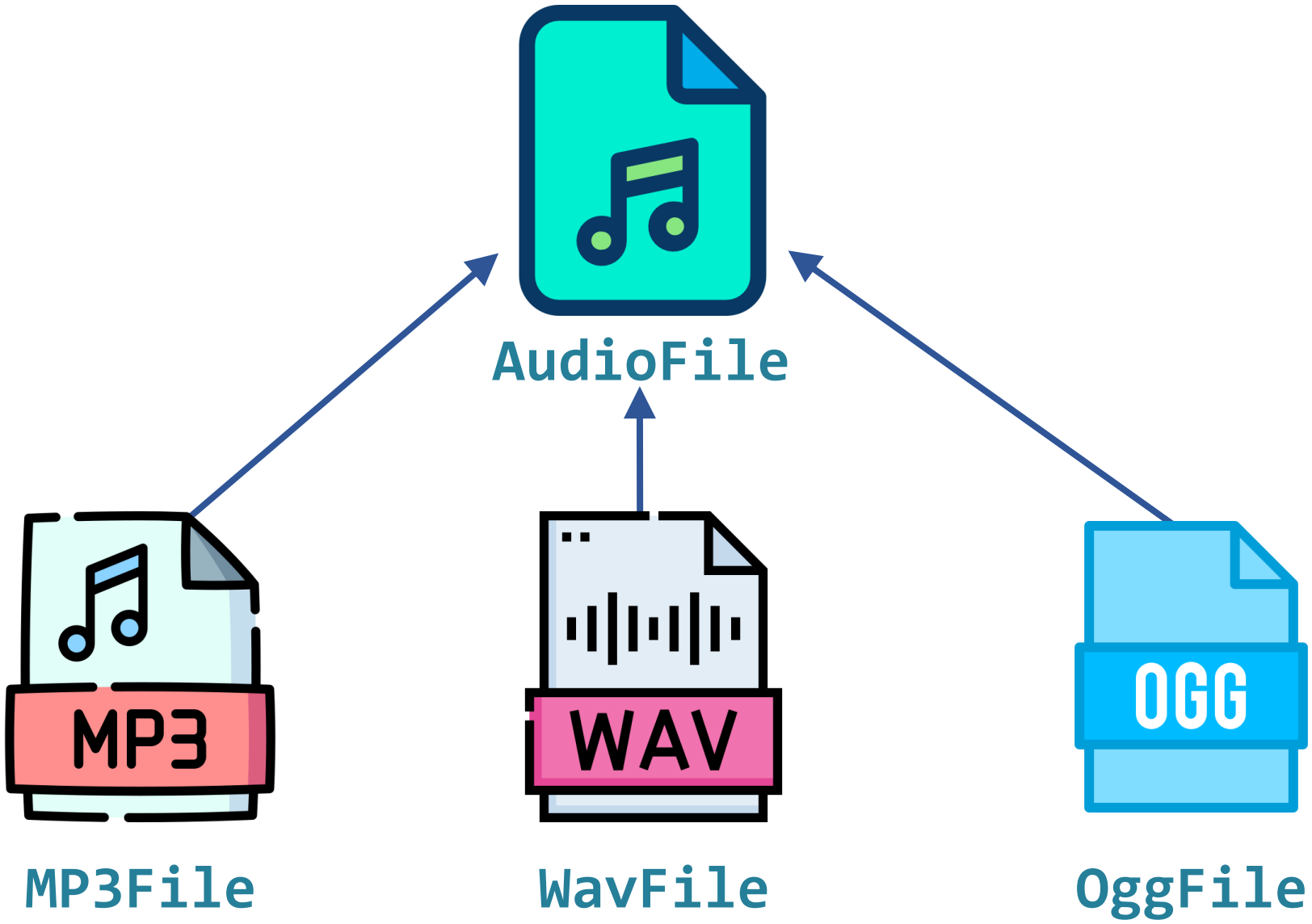
- Polymorphism means one interface with multiple implementations or simply "many forms".

Polymorphism

- Polymorphism means one interface with multiple implementations or simply “many forms”.



Polymorphism: Audio Files



Polymorphism: Audio Files

```
class AudioFile:
    pass

class MP3File(AudioFile):
    def play(self):
        print('Playing mp3 file.')

class WavFile(AudioFile):
    def play(self):
        print('Playing wav file.')

class OggFile(AudioFile):
    def play(self):
        print('playing ogg file.')
```

Polymorphism: Audio Files

```
file1 = MP3File()
file1.play()

file2 = WavFile()
file2.play()

file3 = OggFile()
file3.play()

# Output
Playing mp3 file.
Playing wav file.
playing ogg file.
```

Inheritance: Data Hiding

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

    @property
    def name(self):
        return self.__name

    @name.setter
    def name(self, name):
        self.__name = name

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        self.__age = age

    def say_name(self):
        print("I'm", self.__name)
```

Inheritance: Data Hiding

```
class Employee(Person):
    def __init__(self, name, age, id_number, salary):
        super().__init__(name, age)
        self.__id_number = id_number
        self.__salary = salary

    @property
    def id_number(self):
        return self.__id_number

    @property
    def salary(self):
        return self.__salary

    @salary.setter
    def salary(self, salary):
        if salary > 0:
            self.__salary = salary
        else:
            self.__salary = 0

    def print_data(self):
        print(self.name, self.age, self.id_number, self.salary)
```

Inheritance: Data Hiding

```
emp1 = Employee('Ahmed', 25, 10, 7000)
emp1.print_data()

emp2 = Employee('Mohamed', 30, 20, 9000)
emp2.name = 'Mohamed Ali'
emp2.age += 1
emp2.salary += 1000
emp2.print_data()

# Output
Ahmed 25 10 7000
Mohamed Ali 31 20 10000
```

Inheritance: Data Hiding

```
emp3 = Employee('Omar Kareem', 22, 11, 5000)
print(emp3.id_number)

emp3.id_number = 33

# Output
11
AttributeError: can't set attribute
```

Inheritance: Data Hiding

- The derived class inherits all members (with some exceptions) from the base class, and it can add to them.
- A public member in the base class becomes a public member in the derived class.
- A private member in the base class becomes an inaccessible (hidden) member in the derived class.
- The `@property` decorator is used to make the private attribute as property getter method.
- The `@name.setter` decorator is used to make the private attribute as property setter method.