Class : Class is a blueprint or a prototype for creating objects. Each class has a set of attributes and functionalities. Class is a user defined datatype for creating objects. No memory is allotted when a class is created .

Syntax :

class Animal:
    pass #It contains the attributes and functions

Defining a Class

Syntax:

```
class ClassName:
    # Constructor
    def __init__(self, attribute1, attribute2):
        self.attribute1 = attribute1
        self.attribute2 = attribute2

    # Method
    def some_method(self):
        return f"Attribute1 is {self.attribute1},
Attribute2 is {self.attribute2}"
```

Creating an Object

Syntax:

```
object_name = ClassName(value1, value2)
```

Key Points about Classes and Objects

1. **Class Definition**:
    - Defined using the `class` keyword.
    - Contains a **constructor** (`__init__`) to initialize object attributes.
    - Contains methods to define object behaviors.
2. **Objects**:
    - Created from classes; an instance of a class.
    - Access class attributes and methods using the dot (`.`) operator.

SOME IMPORTANT CONCEPTS :-

`self`

The `self` keyword in Python is used to represent the **current instance of the class**. It is a reference to the object calling the method or accessing attributes.

Key Points:

- It must be the first parameter of instance methods.
- It allows instance attributes and methods to be accessed.
- It is not a keyword in Python; you can name it anything, but the convention is to use `self`

Example:

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

    def introduce(self):
        return f"My name is {self.name} and I am {self.age} years old."

# Create object
person1 = Person("Alice", 25)

# Access instance attributes and methods using `self`
print(person1.introduce())  # Output: My name is Alice and I am 25 years old.
```

1) self

The `self` keyword in Python is used to represent the **current instance of the class**. It is a reference to the object calling the method or accessing attributes.

Key Points:

- It must be the first parameter of instance methods.
- It allows instance attributes and methods to be accessed.
- It is not a keyword in Python; you can name it anything, but the convention is to use `self`.

Example:

python

Copy code

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

    def introduce(self):
        return f"My name is {self.name} and I am {self.age} years old."

# Create object
person1 = Person("Alice", 25)
```

```
# Access instance attributes and methods using `self`

print(person1.introduce())  # Output: My name is Alice
and I am 25 years old.
```

---

2) Instance Attributes vs Class Attributes

Instance Attributes:

- Defined in the constructor (`__init__`) of class or directly at the time instance is created.
- Its value is unique to each object of the class.
- Accessed using `self`.

Class Attributes:

- Shared among all instances of a class.
- Defined directly in the class but outside methods.
- Can be accessed using the class name or `self`.

class attributes in Python behave similarly to **static variables** in Java

Example:

```
class Circle:
    pi = 3.14159  # Class attribute

    def __init__(self, radius):
        self.radius = radius  # Instance attribute

    def area(self):
        return Circle.pi * (self.radius ** 2)  # Access
class attribute using class name

# Create objects
circle1 = Circle(5)
circle2 = Circle(10)

# Access attributes
print(circle1.radius)  # Output: 5 (instance attribute)
print(Circle.pi)       # Output: 3.14159 (class
attribute shared by all instances)
```

What if the Name of Instance and Class Attribute is the Same?

- If an **instance attribute** has the same name as a **class attribute**, the instance attribute takes precedence for that specific object.
- The class attribute however, remains unaffected and can still be accessed using the class name.

Example:

```
class Example:

    attribute = "I am a class attribute"  # Class attribute


    def __init__(self):

        self.attribute = "I am an instance attribute"  # 
Instance attribute (overrides)


# Create an object

obj = Example()


# Access attributes

print(obj.attribute)     # Output: I am an instance 
attribute (instance takes precedence)

print(Example.attribute) # Output: I am a class attribute 
(class attribute remains intact)
```

4) Types of Methods

There are three types of methods in Python:

a) Instance Methods:

- Use `self` as the first parameter.
- Operate on instance attributes.
- Can access and modify the state of the object.

Example:

```
class Person:

    def __init__(self, name):

        self.name = name  # Instance attribute


    def greet(self):  # Instance method

        return f"Hello, {self.name}!"


person = Person("Alice")

print(person.greet())  # Output: Hello, Alice!
```

---

b) Class Methods:

- Use `@classmethod` decorator.
- Use `cls` as the first parameter (refers to the class).

- Operate on class attributes, not instance attributes.

Example:

```
class Circle:

    pi = 3.14159  # Class attribute


    @classmethod

    def description(cls):  # Class method

        return f"This is the Circle class with pi = {cls.pi}."


print(Circle.description())  # Output: This is the Circle class with pi = 3.14159.
```

Static Methods:

- Use `@staticmethod` decorator.
- Do not take `self` or `cls` as a parameter.
- Independent of class or instance; behaves like a standalone function within the class.

Example:

```
class Calculator:

    @staticmethod

    def add(x, y):  # Static method
```

```
        return x + y

print(Calculator.add(5, 10))  # Output: 15
```

Comparison of Methods

Feature	Instance Method	Class Method	Static Method
Parameter	`self`	`cls`	No specific parameter
Access Instance Attributes?	Yes	No	No
Access Class Attributes?	Yes	Yes	No
Call Without Instance?	No	Yes	Yes

Constructors in Python

A **constructor** is a special method in Python used to initialize an object's attributes when the object is created. In Python, the constructor is defined using the `__init__` method.

Key Points About Constructors

1. **Purpose**:
    - Automatically called when an object is created.
    - Used to set initial values for instance attributes.

Syntax:

```
def __init__(self, parameters):
    # Initialize instance attributes
```

Types of Constructors:

- **Default Constructor**: No parameters except `self`.
- **Parameterized Constructor**: Takes arguments to initialize attributes.
- **Multiple Constructors**: Python doesn't support true method overloading, but default parameters or `@classmethod` can mimic this.

Special Notes:

- A class **can have only one `__init__` method**, but it can handle multiple scenarios using optional/default parameters.
- Unlike Java, constructors in Python don't need to match the class name.

1) Default Constructor

- A constructor without any parameters other than `self`.
- Initializes the object with default values.

Example:

python Copy code

```python
class Person:
    def __init__(self):  # Default constructor
        self.name = "Unknown"
        self.age = 0

# Create object
person = Person()
print(person.name)  # Output: Unknown
print(person.age)   # Output: 0
```

2) Parameterized Constructor

- A constructor that takes arguments to initialize attributes.

Example:

python Copy code

```python
class Person:
    def __init__(self, name, age):  # Parameterized constructor
        self.name = name
        self.age = age

# Create object
person = Person("Alice", 25)
print(person.name)  # Output: Alice
print(person.age)   # Output: 25
```

3) Simulating Multiple Constructors

Python doesn't support true constructor overloading. You can use:

- **Default Parameters**: Handle optional arguments in a single constructor.
- **Class Methods**: Create alternative ways to instantiate an object.

Using Default Parameters

python Copy code

```python
class Person:
    def __init__(self, name="Unknown", age=0):  # Single constructor with defaults
        self.name = name
        self.age = age

# Create objects
person1 = Person()
person2 = Person("Alice", 25)

print(person1.name, person1.age)  # Output: Unknown 0
print(person2.name, person2.age)  # Output: Alice 25
```

4) Constructor with Variable Number of Arguments

You can use `*args` or `**kwargs` to handle variable-length arguments.

Using `*args`:

python Copy code

```python
class Person:
    def __init__(self, *args):
        self.name = args[0] if len(args) > 0 else "Unknown"
        self.age = args[1] if len(args) > 1 else 0

# Create objects
person1 = Person("Alice", 25)
person2 = Person()

print(person1.name, person1.age)  # Output: Alice 25
print(person2.name, person2.age)  # Output: Unknown 0
```

Using `**kwargs`:

python Copy code

```python
class Person:
    def __init__(self, **kwargs):
        self.name = kwargs.get('name', 'Unknown')
        self.age = kwargs.get('age', 0)

# Create objects
person1 = Person(name="Alice", age=25)
person2 = Person()

print(person1.name, person1.age)  # Output: Alice 25
print(person2.name, person2.age)  # Output: Unknown 0
```

5) Constructor Chaining

Python does not have built-in constructor chaining like Java, but you can call other constructors explicitly.

Example:

python Copy code

```python
class A:
    def __init__(self):
        print("Constructor of A")

class B(A):
    def __init__(self):
        super().__init__()  # Calls the constructor of class A
        print("Constructor of B")

# Create object
b = B()
# Output:
# Constructor of A
# Constructor of B
```

Important Facts About Constructors

1. **`__init__` is not mandatory**: If no constructor is defined, Python provides a default one.
2. **Not a true constructor**: Unlike Java or C++, `__init__` is technically an initializer, as the object is created before it is called.
3. **Cannot return values**: The `__init__` method always returns `None`.
4. **Multiple ways to instantiate**: You can use class methods to simulate different constructors.
5. **Class vs Instance Initialization**:
    - Class attributes are initialized outside the constructor.
    - Instance attributes are initialized inside the constructor.

Step 1: Understanding Default Parameters

- In Python, you can set **default values** for parameters in a method. This means if a value is not provided when calling the method, the default value is used.
- In constructors (`__init__`), this feature allows us to make some arguments optional.

Syntax:

python

Copy code

```
def method_name(parameter=default_value):
    # Method logic
```

Example:

python

Copy code

```
def greet(name="World"):
    print(f"Hello, {name}!")

greet()           # Output: Hello, World! (uses default value)
greet("Alice")    # Output: Hello, Alice! (overrides default value)
```

Step 2: Default Parameters in Constructors

When applied to constructors, default parameters allow us to create objects with some attributes automatically assigned default values.

Code Breakdown:

python

Copy code

```
class Person:

    def __init__(self, name="Unknown", age=0):  # Constructor with default values

        self.name = name  # Instance attribute name

        self.age = age    # Instance attribute age
```

- If you don't provide `name` or `age` while creating the object, it will use the defaults `"Unknown"` and `0`.
- If you provide values for `name` and `age`, those will override the defaults.

Step 3: Object Creation

Now let's look at how the object creation works.

python

Copy code

```
person1 = Person()  # No arguments provided

person2 = Person("Alice", 25)  # Arguments provided
```

1. **For `person1`:**
    - The constructor uses the default values (`name="Unknown", age=0`).
    - `self.name` becomes `"Unknown"`, and `self.age` becomes `0`.
2. **For `person2`:**
    - The constructor uses the provided values (`name="Alice", age=25`).
    - `self.name` becomes `"Alice"`, and `self.age` becomes `25`.

Step 4: Accessing Attributes

When you access attributes using `person1.name` or `person2.age`, Python fetches the values assigned during object initialization.

python

Copy code

```
print(person1.name, person1.age)  # Output: Unknown 0

print(person2.name, person2.age)  # Output: Alice 25
```

Step 5: Strengthening Your Foundation

Here's a simplified analogy to relate:

- Think of a **constructor** as a recipe for making sandwiches:
    - Default ingredients (bread and butter) are always there.
    - You can specify additional ingredients (cheese, veggies, etc.) if you want.

Code Equivalent:

python

Copy code

```
class Sandwich:

    def __init__(self, ingredient1="Bread",
ingredient2="Butter"):

        self.ingredient1 = ingredient1

        self.ingredient2 = ingredient2


# Default sandwich

sandwich1 = Sandwich()

print(sandwich1.ingredient1, sandwich1.ingredient2)  # Output:
Bread Butter


# Custom sandwich

sandwich2 = Sandwich("Cheese", "Tomato")

print(sandwich2.ingredient1, sandwich2.ingredient2)  # Output:
Cheese Tomato
```

Hands-on Exercise

Try writing this simple class with default parameters yourself:

python

Copy code

```
class Animal:
    def __init__(self, species="Dog", sound="Bark"):
        self.species = species
        self.sound = sound

# Test the default values
animal1 = Animal()
print(f"{animal1.species} says {animal1.sound}")

# Override defaults
animal2 = Animal("Cat", "Meow")
print(f"{animal2.species} says {animal2.sound}")
```

Yes, it's absolutely possible to specify only one value (e.g., for `age`) while leaving the other (e.g., `name`) as its default value. Python allows you to achieve this using **keyword arguments**. Here's how:

Using Positional Arguments

If you use positional arguments, the values are assigned in the order of the parameters defined in the constructor.

python

Copy code

```
person = Person(30)  # Positional argument for `name`
```

In this case:

- `name` will be "Unknown" because you passed a value for `age` explicitly.

Using Keyword Arguments

You can use **keyword arguments** to specify a particular parameter explicitly. This is very flexible and allows you to skip parameters you don't want to specify.

___**Syntax KeyPhrase End Braces in Following Punctuation

code ==="Python Person": >

Here's how you can specify only one parameter, such as `age`, while leaving `name` as its default value:

Example with Keyword Arguments

python

Copy code

```
class Person:

    def __init__(self, name="Unknown", age=0):  # Constructor
with default values

        self.name = name

        self.age = age


# Create an object and only specify 'age'
```

```
person = Person(age=25)


# Access the attributes
print(person.name)  # Output: Unknown
print(person.age)   # Output: 25
```

Explanation

- By using `age=25`, you're telling Python to specifically set the `age` attribute to `25` while leaving the `name` attribute as its default value, `"Unknown"`.
- Keyword arguments let you skip over arguments you don't want to explicitly define.

Key Points About Keyword Arguments

1. **Flexibility**: You can pass arguments in any order when using keyword arguments.
2. **Defaults Preserved**: Any parameters not specified will retain their default values.
3. **Combine with Positional Arguments**: You can mix positional and keyword arguments, but positional arguments must come first.

Example:

```
person = Person("Alice", age=30)  # Positional for 'name',
keyword for 'age'


print(person.name)  # Output: Alice
```

```
print(person.age)   # Output: 30
```

If You Skip Both Arguments

```
person = Person()

print(person.name)  # Output: Unknown

print(person.age)   # Output: 0
```

Using Keyword Arguments

You can change the order when using **keyword arguments**, as you explicitly specify the parameter name:

python Copy code

```python
# Changing the order with keyword arguments
person = Person(age=25, name="Alice")  # You can provide 'age' first
print(person.name)  # Output: Alice
print(person.age)   # Output: 25
```

This flexibility makes keyword arguments ideal for scenarios where you only want to specify some parameters or provide them in a different order.

Mixing Positional and Keyword Arguments

If you mix positional and keyword arguments:

1. Positional arguments must come **first**.
2. Keyword arguments can follow.

```
person = Person("Alice", age=30) # Positional for 'name', keyword for 'age'
```

Error When Changing Order with Positional Arguments

If you try to change the order without using keywords:

```
person = Person(25, "Alice") # Error: 'name' gets 25, 'age' gets "Alice"
```

Summary

- Use **positional arguments** when you want to follow the order as defined in the constructor of class.
- Use **keyword arguments** when you want flexibility in the order or to skip some parameters.