Day-4

Agenda:

Magic Methods in python, Object as an argument, Instances as Return Values, namespaces

Magic methods

- Magic methods in Python are the special methods that **start and end with the double underscores**.
- They are also called dunder methods.
- Magic methods are not meant to be invoked directly by you, but the invocation happens internally from the class on a certain action.
- For example, when you add two numbers using the + operator, internally, the __add__() method will be called.

- Built-in classes in Python define many magic methods.
- Use the dir() function to see the number of magic methods inherited by a class.

For example, the following lists all the attributes and methods defined in the int class.

Example: dir(int)

['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__', '__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__',

'__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']

- If you use + operator on two integers, it adds two numbers. But if you use + operator with two strings then it concatenates two strings.
- We can not use + operator with an integer and a string. Because both belongs to different classes.
- The __add__ method is a magic method of class **int** which gets called when we **add two numbers** using the + operator.
- The __add__ method is a magic method of class **str** which gets called when we **concatenate two strings** using the + operator.

For numbers:

```
#using + operator
num=10
x=num + 5
print(x)
```

```
15
```

```
# Addition of two numbers using __add__()
num = 10
x = num.__add__(5)
print(x)
```

```
15
```

For strings:

```
# using + operator
st = '10'
x = st + '5'
print(x)
```

```
105
```

```
# concatenation using __add__()
st = '10'
x = st.__add__('5')
print(x)
```

```
105
```

Example:

Define a custom class to represent a mass, and then use the __add__ magic method to define how to add two instances of this class. This will allow us to add masses in kilograms and grams properly.

```python
class Mass:
    def __init__(self, kg=0, g=0):
        # Ensure that grams are always less than 1000
        self.kg = kg + g // 1000
        self.g = g % 1000

    def __add__(self, other):
        # Add the kilograms and grams separately
        total_kg = self.kg + other.kg
        total_g = self.g + other.g

        # If grams exceed 1000, convert to kilograms
        if total_g >= 1000:
            total_kg += total_g // 1000
            total_g = total_g % 1000

        return Mass(total_kg, total_g)

    def __repr__(self):
        return f"{self.kg} kg and {self.g} g"

# Example usage
mass1 = Mass(2, 500)   # 2 kg 500 g
mass2 = Mass(1, 750)   # 1 kg 750 g

mass3 = mass1 + mass2  # Using the __add__ magic method
print(mass3)           # Output: 4 kg and 250 g
```

```
4 kg and 250 g
```

One similar example to be done by students:

Define a custom class to represent a distance(Km and m), and then use the `__add__` magic method to define how to add two instances of this class. This will allow us to add distances in kilometers and meters properly.

Magic Method __gt__() with Class

Suppose we want to compare two objects of a user defined class. Here m1 and m2 are marks of two subjects and we have to compare their total marks.

```
class Student:
    def __init__(self, m1, m2):
        self.m1 = m1
        self.m2 = m2
    def __gt__(self, other):
        r1 = self.m1 +  self.m2
        r2 = other.m1 + other.m2
        if r1 > r2:
            return True
        else:
            return False

s1 = Student(58, 69)
s2 = Student(69, 65)
if s1 > s2:
    print("s1 is greater")
else:
    print("S2 is greater")
```

```
S2 is greater
```

Magic Method __str__()

- This tells what you will see when you print an Object…print(obj)
- We can implement the __str__() method to customize the string representation of an instance of a class.
- It is overridden to return a printable string representation of any user defined class.
- The python __str__() method returns the string representation of the object.
- The same method is also called when the str() or print() function is invoked on an object.

__str__() Method Example

In the following example we are implementing the __str__() method to display a custom string.

```
class Person:
    def __init__(self,name,age):
        self.name=name
        self.age=age
    def __str__(self):
        return F"The name is {self.name} and age is {self.age}"
p = Person("Mohit",35)
print(p)
```

```
The name is Mohit and age is 35
```

__new__() Method

- Languages such as Java and C# use the new operator to create a new instance of a class.
- In Python the **__new__() magic method is implicitly called before the __init__() method**.
- The __new__() method returns a new object, which is then initialized by __init__().

```
class Person:
    def __new__(self):
        print("This is a new method")
        self.__init__(self)
    def __init__(self):
        print("init method")

p = Person()
```

```
This is a new method
init method
```

Creating custom magic method

You can create your own custom magic method.

Example: Fraction Class with Custom Magic Methods

We'll create a `Fraction` class that supports addition, comparison, and string representation.

```python
import math

class Fraction:
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator
        self.simplify()

    def simplify(self):
        common_divisor = math.gcd(self.numerator, self.denominator)
        self.numerator //= common_divisor
        self.denominator //= common_divisor

    def __add__(self, other):
        new_numerator = self.numerator * other.denominator + other.numerator * self.denominator
        new_denominator = self.denominator * other.denominator
        return Fraction(new_numerator, new_denominator)

    def __eq__(self, other):
        return self.numerator == other.numerator and self.denominator == other.denominator

    def __lt__(self, other):
        return self.numerator * other.denominator < other.numerator * self.denominator

    def __repr__(self):
        return f"{self.numerator}/{self.denominator}"
```

```python
# Example usage
f1 = Fraction(1, 2)
f2 = Fraction(2, 3)

f3 = f1 + f2  # Fraction addition
print(f3)     # Output: 7/6

print(f1 == f2)  # Output: False
print(f1 < f2)   # Output: True
```

```
7/6
False
True
```

Object as an Argument

In Python, you can pass objects as arguments to functions. This allows for more flexible and reusable code.

Example

```
class MyClass:
    def __init__(self, value):
        self.value = value

def print_value(obj):
    print(f"The value is: {obj.value}")

obj = MyClass(10)
print_value(obj)
```

```
The value is: 10
```

Example:

Let's define a class Person and another class Greeting. The Greeting class will have a method that takes a Person object as an argument and generates a personalized greeting message.

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

    def __repr__(self):
        return f"Person({self.name})"

class Greeting:
    def generate_greeting(self, person):
        return f"Hello, {person.name}! Welcome!"

# Example usage
person = Person("Alice")
greeting = Greeting()

message = greeting.generate_greeting(person)
print(message)
```

```
Hello, Alice! Welcome!
```

Instances as Return Values

- Function can return objects.
- Functions in Python can return instances of classes. This allows for the creation of new objects as the result of function calls.

Example:

```
class MyClass:
    def __init__(self, value):
        self.value = value

def create_object(value):
    return MyClass(value)

obj = create_object(10)
print(obj.value)
```

```
10
```

Example: Define a class Rectangle that represents a rectangle with width and height. The Rectangle class will have a method that takes another Rectangle object and returns a new Rectangle object whose area is same as the combined area of both rectangles. Please note, In result we are expecting the rectangle's width and height whose areas is the sum of the area of two rectangles.

```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def combine(self, other):
        # Combine areas of two rectangles
        new_area = self.area() + other.area()
        # Create a new rectangle with the combined area
        # For simplicity, let's assume the width of the new rectangle
        # is the same as the first rectangle and we calculate
        # the new height accordingly.
        new_width = self.width
        new_height = new_area / new_width
        return Rectangle(new_width, new_height)

    def __repr__(self):
        return f"Rectangle({self.width}, {self.height})"

# Example usage
rect1 = Rectangle(4, 5)
rect2 = Rectangle(3, 6)

combined_rect = rect1.combine(rect2)
print(combined_rect)
```

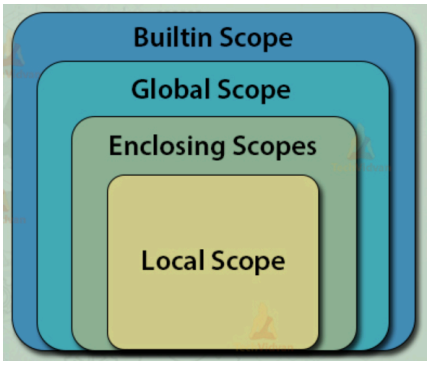
```
Rectangle(4, 9.5)
```

Namespace

- Namespace is about scope and lifetime of identifiers.
    - Identifiers: names of variables, functions, classes, etc.
    - Scope: The region of the code where a namespace is accessible.

    - Lifetime: The duration for which a namespace exists.
- Namespace acts as a boundary, ensuring that names are unique and avoiding naming conflicts.
- The namespace is a container that holds identifiers (names of variables, functions, classes, etc.) and maps them to their corresponding objects.
- Python provides multiple types of namespaces:

1. **Local Namespace:** refers to the names defined within a function.
2. **Non-local Namespace:** Also called enclosing namespace. It corresponds to the namespaces of enclosing functions (for nested functions: Function within function is called nested function).
3. **Global Namespace:** Encompasses the names defined at the top level of a module or script.
4. **Built-in Namespace:** Contains the names of built-in Python functions and objects.



1. Local Namespace

- The local namespace is within function.
- It is created whenever a function is called and is destroyed when the function exits.
- It contains the names of variables and parameters that are defined within the function.
- These variables are only accessible within the function's body.
- When the function is called again, a new local namespace is created.

```
def my_function():
    x = 5          # x IS LOCAL VARIABLE
    print(x)

my_function()

print(x)    # PRINTING x HERE WILL GIVE ERROR
```

```
5
```

2. Enclosing Namespace:

- Enclosing (or nonlocal) scope is a special scope that only exists for nested functions.
- Enclosing Namespace means the inner function can access variables from the outer function's namespace.

```
def outer_function():
    x = 20

    def inner_function():
       print(x)  # Accessing variable from the enclosing(outer) namespace
    inner_function()

outer_function()
```

```
20
```

This is same as:

```
def outer_function():
    x = 20

    def inner_function():
        nonlocal x
        print(x)  # Accessing variable from the enclosing(outer) namespace
    inner_function()

outer_function()
```

```
20
```

3. Global Namespace:

- The global namespace covers the entire module or script.
- Variables defined at the top level of the module(outside of all the function) belong to the global namespace and are accessible from anywhere within the module.

```
x = 50

def my_function():
    print(x)  # Accessing variable from the global namespace

my_function()
print(x)
```

```
50
50
```

4. Built-in Namespace:

- The built-in namespace contains Python's built-in functions and objects.
- These names are always available without the need for importing or defining them.
- Examples include functions like print(), len() and objects like int and lists.

```
print(len([1, 2, 3]))

# Trying to redefine a built-in function
# This will raise a SyntaxError
# def len(x):
#  return 42
```

```
3
```