Python Fundamentals day 56

Today's Agenda

- Magic methods
- Operator overloading

Magic Methods

Dunder or magic methods in Python are the methods having two prefix and suffix underscores in the method name. Dunder here means "Double Under (Underscores)". These are commonly used for **operator overloading**. Few examples for magic methods are: __init__, __add__, __len__, __repr__ etc.

Let us take an example and see how operator overloading is done

```
def main():
    a=5
    print(a)
    a=a+5 #a=a.__add__(5)
    print(a)
if __name__=='__main__':
    main()
```

Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
5
10
```

In the above example a=a+5 executes in the following manner

```
class int:
    .
    .
    .
    def  __add__(self,other):
        .
        .
        .
```

Similar to the above operation subtraction, multiplication, division and power also execute in same manner.

a=a-5 → a=a.__sub__(5)

a=a*5 → a=a.__mul__(5)

a=a/5 → a=a.__div__(5)

a=a**5 → a=a.__pow__(5)

So let us cross verify by using one of these magic method in the above code.

```
def main():
    a=5
    print(a)
    a=a.__add__(5)
    print(a)
if __name__=='__main__':
    main()
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
5
10
```

Let us next take example of concatenation and see how it works, because we use the same + operator for concatenation as well

```
def main():
    s='pyt'
    print(s)
    s=s+'hon' #s=s.__add__('hon')
    print(s)
if __name__=='__main__':
    main()
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
pyt
python
```

Note that here string class is executing not the int class.

```
class str:
    .
    .
    .
    def __add__(self,other):
        .
        .
        .
```

If we try to mix both integer and string and then use + operator, you will definitely get an error as shown below

```
def main():
    s='pyt'
    print(s)
    s=s+5 #s=s.__add__('5')
    print(s)
if __name__=='__main__':
    main()
```

Output:

```
  File "C:/Users/rooman/OneDrive/Desktop/python/test.py",
line 7, in <module>
    main()

  File "C:/Users/rooman/OneDrive/Desktop/python/test.py",
line 4, in main
    s=s+5 #s=s.__add__('5')

TypeError: can only concatenate str (not "int") to str
```

If we reverse the order then as well we will get the error

```
def main():
    s='pyt'
    print(s)
    s=5+s #s=5.__add__('s')
    print(s)
if __name__=='__main__':
    main()
```

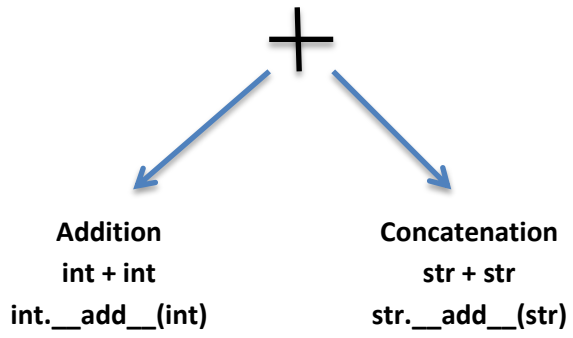
Output:

```
  File "C:/Users/rooman/OneDrive/Desktop/python/test.py",
line 7, in <module>
    main()

  File "C:/Users/rooman/OneDrive/Desktop/python/test.py",
line 4, in main
    s=5+s #s=5.__add__('s')

TypeError: unsupported operand type(s) for +: 'int' and
'str'
```

So be careful when using operators and do not mix the different types of data

Operator overloading

Let us take an example to understand what is operator overloading. In the above example we saw + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings. This feature in Python that allows the same operator to have different meaning according to the context is called operator overloading.



All the above examples are on built-in methods. Let us see how to achieve the same with user defined type.

```
class Point:

    def __init__(self,x,y):
        self.x=x
        self.y=y

    def display(self):
        print(self.__dict__)

def main():
    p1=Point(2,3)
    p2=Point(1,1)
    p1.display()
    p2.display()

if __name__=='__main__':
    main()
```

Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'x': 2, 'y': 3}
{'x': 1, 'y': 1}
```

Now let us see if we can get the third point which will be the added result of these two points.

```
class Point:

    def __init__(self,x,y):
        self.x=x
        self.y=y

    def display(self):
        print(self.__dict__)

def main():
    p1=Point(2,3)
    p2=Point(1,1)
    p3=p1+p2 #p3=p1.__add__(p2)

if __name__=='__main__':
    main()
```

Output:

```
  File "C:/Users/rooman/OneDrive/Desktop/python/test.py",
line 16, in <module>
    main()

  File "C:/Users/rooman/OneDrive/Desktop/python/test.py",
line 13, in main
    p3=p1+p2

TypeError: unsupported operand type(s) for +: 'Point' and
'Point'
```

Certainly above code will throw error because there is no __add__ method in point class. So let us see how to create a __add__ method and see if it works

```
class Point:

    def __init__(self,x,y):
        self.x=x
        self.y=y

    def __add__(self,other):
        return Point(self.x + other.x, self.y + other.y)

    def display(self):
        print(self.__dict__)

def main():
    p1=Point(2,3)
    p2=Point(1,1)
    p3=p1+p2 #p3=p1.__add__(p2)
    p3.display()

if __name__=='__main__':
    main()
```

Output:

```
In [8]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'x': 3, 'y': 4}
```

Let us see if we can print p1, p2, p3 using print() and not by the display().

```
class Point:

    def __init__(self,x,y):
        self.x=x
        self.y=y

    def __add__(self,other):
        return Point(self.x + other.x, self.y + other.y)

    def display(self):
        print(self.__dict__)

def main():
    p1=Point(2,3)
    p2=Point(1,1)
    p3=p1+p2 #p3=p1.__add__(p2)

    print(p1)
    print(p2)
    print(p3)

if __name__=='__main__':
    main()
```

Output:

```
In [10]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
<__main__.Point object at 0x0000027BA95EC9C8>
<__main__.Point object at 0x0000027BA95FB508>
<__main__.Point object at 0x0000027BA95FB548>
```

Instead of printing the values, it is printing the object. This is because of the absence of certain feature in __str__() because print() expects the arguments in it to be string type. If not it tries to convert it into string.

As __str__() is inherited by point class from object class, we can override it and modify according to our needs as shown below

```
class Point:

    def __init__(self,x,y):
        self.x=x
        self.y=y

    def __add__(self,other):
        return Point(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f'({self.x},{self.y})'

def main():
    p1=Point(2,3)
    p2=Point(1,1)
    p3=p1+p2 #p3=p1.__add__(p2)

    print(p1.__str__())
    print(p2)
    print(p3)

if __name__=='__main__':
    main()
```

Output:

```
In [12]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
(2,3)
(1,1)
(3,4)
```

But what if the point objects are identical as below and you are not sure if they are same objects or different objects. How to distinguish? Let us see

```
class Point:

    def __init__(self,x,y):
        self.x=x
        self.y=y

    def __add__(self,other):
        return Point(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f'({self.x},{self.y})'

    def __repr__(self):
        return f'{type(self)} {id(self)}'

def main():
    p1=Point(2,3)
    p2=Point(1,1)

    print(p1)
    print(p2)
    print(p1.__repr__())
    print(p2.__repr__())

if __name__=='__main__':
    main()
```

Output:

```
In [14]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
(2,3)
(1,1)
<class '__main__.Point'> 2730145810440
<class '__main__.Point'> 2730145283592
```

Great so now we know they are two different objects.