lesson_9

September 4, 2022

1 Lesson 9 ~ Magic Methods

[2]:
```
from functools import total_ordering
```

[8]:
```
@total_ordering
class Rectangle:
    def __init__(self, width, length):
        if width < 0 or length < 0:
            raise Exception("width and length should be positive numbers")
        self.width = width
        self.length = length

    def area(self):
        return self.calculate_area(self.width, self.length)

    @staticmethod
    def calculate_area(width, length):
        return width * length

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

    def __lt__(self, other):
        return self.area() < other.area()

    def __eq__(self, other):
        return self.area() == other.area()

    def __bool__(self):
        return self.area() > 0
```

[5]:
```
rectangle_1 = Rectangle(20, 50)
```

[6]:
```
print(rectangle_1)
```

```
Rectangle(20, 50)
```

```
[7]: rectangle_1.__repr__()
```

```
[7]: 'Rectangle(20, 50)'
```

```
[13]: rectangle_2 = Rectangle(12, 24)
```

```
[14]: rectangle_1 < rectangle_2
```

```
[14]: False
```

```
[15]: rectangle_1.area() + rectangle_2.area()
```

```
[15]: 1288
```

```
[16]: rectangle_1 + rectangle_2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [16], in <cell line: 1>()
----> 1 rectangle_1 + rectangle_2

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

```
[17]: @total_ordering
class Rectangle:
    def __init__(self, width, length):
        if width < 0 or length < 0:
            raise Exception("width and length should be positive numbers")
        self.width = width
        self.length = length

    def area(self):
        return self.calculate_area(self.width, self.length)

    @staticmethod
    def calculate_area(width, length):
        return width * length

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

    def __lt__(self, other):
        return self.area() < other.area()

    def __eq__(self, other):
        return self.area() == other.area()
    def __bool__(self):
        return self.area() > 0

    def __add__(self, other):
        return self.area() + other.area()
```

[18]:
```
rectangle_1 = Rectangle(20, 50)
rectangle_2 = Rectangle(12, 24)
```

[19]:
```
rectangle_1 + rectangle_2
```

[19]:
```
1288
```

[21]:
```
rectangle_1.area()
```

[21]:
```
1000
```

[22]:
```
rectangle_1 + 200
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [22], in <cell line: 1>()
----> 1 rectangle_1 + 200

Input In [17], in Rectangle.__add__(self, other)
     28 def __add__(self, other):
---> 29     return self.area() + other.area()

AttributeError: 'int' object has no attribute 'area'
```

[46]:
```
@total_ordering
class Rectangle:
    def __init__(self, width, length):
        if width < 0 or length < 0:
            raise Exception("width and length should be positive numbers")
        self.width = width
        self.length = length

    def area(self):
        return self.calculate_area(self.width, self.length)

    @staticmethod
    def calculate_area(width, length):
        return width * length
    def __repr__(self):
        return f"Rectangle({self.width}, {self.length})"

    def __lt__(self, other):
        return self.area() < other.area()

    def __eq__(self, other):
        return self.area() == other.area()

    def __bool__(self):
        return self.area() > 0

    def __add__(self, other):
        if isinstance(other, (int, float)):
            return self.area() + other
        elif isinstance(other, self.__class__):
            return self.area() + other.area()
        else:
            raise TypeError(f"Can't add Rectangle with {other.__class__}")
```

```
[47]: rectangle_1 = Rectangle(20, 50)
      rectangle_2 = Rectangle(12, 24)
```

```
[48]: rectangle_1 + 200
```

```
[48]: 1200
```

```
[49]: rectangle_1 + 3.14
```

```
[49]: 1003.14
```

```
[50]: rectangle_1 + rectangle_2
```

```
[50]: 1288
```

```
[51]: rectangle_1 + "test"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [51], in <cell line: 1>()
----> 1 rectangle_1 + "test"

Input In [46], in Rectangle.__add__(self, other)
     32     return self.area() + other.area()
     33 else:
---> 34     raise TypeError(f"Can't add Rectangle with {other.__class__}")
```

```
TypeError: Can't add Rectangle with <class 'str'>
```

[45]:
```python
isinstance(True, int)
```

[45]:
```
True
```

[52]:
```python
rectangle_1 + 200
```

[52]:
```
1200
```

[53]:
```python
200 + rectangle_1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [53], in <cell line: 1>()
----> 1 200 + rectangle_1

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

[54]:
```python
rectangle_3 = Rectangle(42, 10)
```

[55]:
```python
rectangle_1 + rectangle_2 + rectangle_3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [55], in <cell line: 1>()
----> 1 rectangle_1 + rectangle_2 + rectangle_3

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

[76]:
```python
@total_ordering
class Rectangle:
    def __init__(self, width, length):
        if width < 0 or length < 0:
            raise Exception("width and length should be positive numbers")
        self.width = width
        self.length = length

    def area(self):
        return self.calculate_area(self.width, self.length)

    @staticmethod
    def calculate_area(width, length):
        return width * length
    def __repr__(self):
        return f"Rectangle({self.width}, {self.length})"

    def __lt__(self, other):
        return self.area() < other.area()

    def __eq__(self, other):
        return self.area() == other.area()

    def __bool__(self):
        return self.area() > 0

    def __add__(self, other):
        if isinstance(other, (int, float)):
            return self.area() + other
        elif isinstance(other, self.__class__):
            return self.area() + other.area()
        else:
            raise TypeError(f"Can't add Rectangle with {other.__class__}")

    def __radd__(self, other):
        return self.__add__(other)

    def __iadd__(self, other):
        raise TypeError("+= not supported for this object")

    def __int__(self):
        return int(self.area())
```

[77]:
```
rectangle_1 = Rectangle(20, 50)
rectangle_2 = Rectangle(12, 24)
rectangle_3 = Rectangle(42, 10)
```

[78]:
```
200 + rectangle_1
```

[78]:
```
1200
```

[79]:
```
rectangle_1 + rectangle_2 + rectangle_3
```

[79]:
```
1708
```

[80]:
```
rectangle_1 += rectangle_2 # rectangle_1 = rectangle_1 + rectangle_2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [80], in <cell line: 1>()
----> 1 rectangle_1 += rectangle_2
```

```
Input In [76], in Rectangle.__iadd__(self, other)
     39 def __iadd__(self, other):
---> 40     raise TypeError("+= not supported for this object")

TypeError: += not supported for this object
```

[81]:
```
rectangle_1
```

[81]:
```
Rectangle(20, 50)
```

[82]:
```
-rectangle_1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [82], in <cell line: 1>()
----> 1 -rectangle_1

TypeError: bad operand type for unary -: 'Rectangle'
```

[83]:
```
int(rectangle_1)
```

[83]:
```
1000
```

[84]:
```
float(rectangle_1)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [84], in <cell line: 1>()
----> 1 float(rectangle_1)

TypeError: float() argument must be a string or a number, not 'Rectangle'
```

[85]:
```
str(rectangle_1)
```

[85]:
```
'Rectangle(20, 50)'
```

[121]:
```
class Building:
    def __init__(self, name, floors):
        self.name = name
        self.floors = floors
        self.occupants = [None] * floors

    def occupy(self, floor_num, occupant_name):
        self.occupants[floor_num] = occupant_name
    def get_occupant(self, floor_num):
        if self.floors < floor_num < 0:
            raise Exception(f"This building has {self.floors} floors")
        return self.occupants[floor_num]
```

[122]:
```
building_1 = Building("Empire State", 3)
```

[123]:
```
building_1
```

[123]:
```
<__main__.Building at 0x112791370>
```

[124]:
```
building_1.occupy(2, "Adam's Home")
```

[125]:
```
building_1.get_occupant(2)
```

[125]:
```
"Adam's Home"
```

[126]:
```
building_1.get_occupant(11)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Input In [126], in <cell line: 1>()
----> 1 building_1.get_occupant(11)

Input In [121], in Building.get_occupant(self, floor_num)
     11 if self.floors < floor_num < 0:
     12     raise Exception(f"This building has {self.floors} floors")
---> 13 return self.occupants[floor_num]

IndexError: list index out of range
```

[127]:
```
building_1[2] = "Adam Smith"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [127], in <cell line: 1>()
----> 1 building_1[2] = "Adam Smith"

TypeError: 'Building' object does not support item assignment
```

[128]:
```
building_1[2]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
```

```
Input In [128], in <cell line: 1>()
----> 1 building_1[2]

TypeError: 'Building' object is not subscriptable
```

[138]:
```
class Building:
    def __init__(self, name, floors):
        self.name = name
        self.floors = floors
        self.occupants = [None] * floors

    def __setitem__(self, key, value):
        self.occupants[key] = value

    def __getitem__(self, key):
        if  key > self.floors or key < 0:
            raise Exception(f"This building has {self.floors} floors")
        return self.occupants[key]

    def __len__(self):
        return self.floors
```

[139]:
```
building_1 = Building(name="Empire State", floors=3)
```

[140]:
```
building_1[2] = "Adam Smith's Home"
```

[141]:
```
building_1[2]
```

[141]:
```
"Adam Smith's Home"
```

[142]:
```
building_1[0]
```

[143]:
```
building_1[1]
```

[144]:
```
building_1[11]
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
Input In [144], in <cell line: 1>()
----> 1 building_1[11]

Input In [138], in Building.__getitem__(self, key)
     10 def __getitem__(self, key):
     11     if  key > self.floors or key < 0:
---> 12         raise Exception(f"This building has {self.floors} floors")
     13     return self.occupants[key]
```

```
Exception: This building has 3 floors
```

[145]:
```
len(building_1)
```

[145]:
```
3
```

[188]:
```
class Foo:
    def __init__(self, x):
        self.x = x

    def __setattr__(self, name, value):
        allowed_attributes = {"x", "test", "bar"}
        if name not in allowed_attributes:
            raise AttributeError(f"Attrubute not allowed. Allowed attributes␣
↪are {allowed_attributes}")
        object.__setattr__(self, name, value)

    # def __getattribute__(self, name):
        # print(f"Getting the attribute: {name}")
        # return object.__getattribute__(self, name)

    def __getattr__(self, name):
        print(f"Getting the attribute: {name} which does not exist")
        return 42
```

[189]:
```
a = Foo(42)
```

[190]:
```
a.x
```

[190]:
```
42
```

[191]:
```
a.y = 24
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [191], in <cell line: 1>()
----> 1 a.y = 24

Input In [188], in Foo.__setattr__(self, name, value)
      6 allowed_attributes = {"x", "test", "bar"}
      7 if name not in allowed_attributes:
----> 8     raise AttributeError(f"Attrubute not allowed. Allowed attributes ar␣
 ↪{allowed_attributes}")
      9 object.__setattr__(self, name, value)
```

```
AttributeError: Attrubute not allowed. Allowed attributes are {'x', 'test',␣
↪'bar'}
```

[193]:
```
a.y
```

```
Getting the attribute: y which does not exist
```

[193]: 42

[194]:
```
a.test = "test"
```

[195]:
```
a.test
```

[195]: 'test'

[196]:
```
a.bar = 42
```

[197]:
```
a.bar
```

[197]: 42

[198]:
```
a.another = "test"
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [198], in <cell line: 1>()
----> 1 a.another = "test"

Input In [188], in Foo.__setattr__(self, name, value)
      6 allowed_attributes = {"x", "test", "bar"}
      7 if name not in allowed_attributes:
----> 8     raise AttributeError(f"Attrubute not allowed. Allowed attributes are␣
↪{allowed_attributes}")
      9 object.__setattr__(self, name, value)

AttributeError: Attrubute not allowed. Allowed attributes are {'x', 'test',␣
↪'bar'}
```

[199]:
```
a.non_existant
```

```
Getting the attribute: non_existant which does not exist
```

[199]: 42

1.1 Slices

```
[200]: a = [1, 12, 24, 42]
```

```
[201]: a[0]
```

```
[201]: 1
```

```
[202]: a[1:3] # slice(1, 3, 1)
```

```
[202]: [12, 24]
```

```
[203]: a[0:3:2]  # slice(0, 3, 2)
```

```
[203]: [1, 24]
```

```
[204]: a[:3]  # slice(0, 3, 1)
```

```
[204]: [1, 12, 24]
```

```
[205]: a[2:]  # slice(2, len(a), 1)
```

```
[205]: [24, 42]
```

```
[206]: a[::2]  # slice(0, len(a), 2)
```

```
[206]: [1, 24]
```

```
[210]: my_slice = slice(0, len(a), 2)
```

```
[211]: my_slice
```

```
[211]: slice(0, 4, 2)
```

```
[212]: my_slice.start
```

```
[212]: 0
```

```
[213]: my_slice.stop
```

```
[213]: 4
```

```
[214]: my_slice.step
```

```
[214]: 2
```

```
[225]: class Building:
           def __init__(self, name, floors):
```

```
        self.name = name
        self.floors = floors
        self.occupants = [None] * floors

    def __setitem__(self, key, value):
        self.occupants[key] = value

    def __getitem__(self, key):
        if isinstance(key, slice):
            return self.occupants[key.start:key.stop:key.step]
        if  key > self.floors or key < 0:
            raise Exception(f"This building has {self.floors} floors")
        return self.occupants[key]

    def __len__(self):
        return self.floors
```

[226]:
```
building_2 = Building("Dvin", 4)
```

[227]:
```
building_2[2] = "Adam Smith"
```

[228]:
```
building_2[2]
```

[228]:
```
'Adam Smith'
```

[230]:
```
building_2[1:3]
```

[230]:
```
[None, 'Adam Smith']
```

[ ]:
```
```