classes

March 4, 2021

Mar 3, 2021, Yu Sun

1 Classes

- Class: blueprint for creating new objects
- Object: instance of a class

For example, we have a class called *human*, and this class will define all the attributes of humans. Then we could create objects, like john, mary, jack and so on.

[1]:
```
x = 1
print(type(x))
```

```
<class 'int'>
```

How to create custom classes, like customer, shopping cart and point, etc.

1.1 Creating Classes

Convention: To name variables and functions, we use all lower case letters, and use an underscore to separate multiple words. However, when naming a class, the first letter of every word should be upper case, and we shouldn't use an underscore to separate multiple words.

Examples: * Variable / function: my_point * Class: MyPoint

[2]:
```
class Point:
    def draw(self):
        print("draw")
```

[3]:
```
point = Point()
point.draw()
print(type(point))
```

```
draw
<class '__main__.Point'>
```

The above "*__main__*" is the name of a module, which we will show later in this course.

[4]:
```
isinstance(point, Point), isinstance(point, int)
```

[4]:
```
(True, False)
```

1.2 Constructors

A class bundles data and functions related to that data into one unit.

A constructor is a special method that is called when we create a new object.

[5]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def draw(self):
        print(f"Point ({self.x}, {self.y})")

point = Point(1, 2)
point.draw()
print(point.x)
```

```
Point (1, 2)
1
```

This magic method "*__init__*" is called a constructor. "self" is the reference to the current point object.

1.3 Class vs Instance Attributes

[6]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def draw(self):
        print(f"Point ({self.x}, {self.y})")

point = Point(1, 2)
point.z = 10
point.draw()

another = Point(3, 4)
another.draw()
```

```
Point (1, 2)
Point (3, 4)
```

In the above, we can add an attribute (point.z) whenever we need. We defined two instances with different attributes x and y.

In some cases, all instances of a class share some common attributes, like all humans have one head.

[7]:
```
class Point:
    default_color = "red"

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

    def draw(self):
        print(f"Point ({self.x}, {self.y})")

point = Point(1, 2)
print(point.default_color)
print(Point.default_color)
point.draw()

another = Point(3, 4)
another.draw()
```

```
red
red
Point (1, 2)
Point (3, 4)
```

[8]:
```
Point.default_color = "yellow"

point = Point(1, 2)
print(point.default_color)
print(Point.default_color)
point.draw()

another = Point(3, 4)
print(another.default_color)
another.draw()
```

```
yellow
yellow
Point (1, 2)
yellow
Point (3, 4)
```

Class-level attributes are shared across all instances of that class.

Mar 4, 2021, Yu Sun

1.4 Class vs Instance Methods

[9]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def draw(self):
        print(f"Point ({self.x}, {self.y})")

point = Point(1, 2)
point.draw()
```

```
Point (1, 2)
```

In the above example, both these methods that we have defined in the Point class are instance methods, so we can call them using an instance of the point class, like `point.draw()`.

But there are times that you don't really need an existing object, and that's when we use a class method. For example, let's say in our program, there are a lot of cases where we want to create a point with initial values (0, 0).

[10]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    @classmethod
    # this is called a decorator, a way to extend the behavior of a method or␣
 ↪function
    def zero(cls): # 'cls' is short for 'class', which is a reference for a␣
 ↪class
        return cls(0, 0) # exactly like calling 'Point(0, 0)'


    def draw(self):
        print(f"Point ({self.x}, {self.y})")

# point = Point(0, 0) # this is a way to get an object with values (0, 0)
point = Point.zero()
point.draw()
```

```
Point (0, 0)
```

In above, we refer to this zero method as a factory method, because it's like a factory, it creates a new object. In some cases, when we need to repeatedly generate some objects with some magical values, like "Point(0, 0, 'a', 'good')", using the above class method is a good alternative way.

1.5 Magic Methods

Magic methods are the methods that have two underscores at the beginning and end of their name, and they are called automatically by Python interpreter, depending on how we use our objects and classes.

For example, in the 'Point' class, we have the `__init__()` magic method, we don't directly call it, it's called automatically by Python interpreter when we create a new point object.

`__str__()` is a useful one, which is called when we try to convert an object to a string. Let's show how it works.

[11]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def draw(self):
        print(f"Point ({self.x}, {self.y})")

point = Point(1, 2)
print(point)
```

```
<__main__.Point object at 0x000001A77CF46D08>
```

If we print the above point object on the terminal, we get the name of our module followed by the class name, and the address of this point object in memory. This is the default implementation of the `__str__()` magic method in the point object. Now let's modify this magic method as follows,

[12]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

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

    def draw(self):
        print(f"Point ({self.x}, {self.y})")

point = Point(1, 2)
print(point)
print(str(point))
```

```
(1, 2)
(1, 2)
```

we can get the values of the point object, which is same to the string conversion of this object.

`__init__()` and `__str__()` are two useful magic methods in Python. In this course, we'll also learn other magic methods. A guide to Python's magic methods can be accessed in

https://rszalski.github.io/magicmethods/ .

1.6 Comparing Objects

There are times that we need to compare two objects. For example, here are two point objects,

[13]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y


point = Point(1, 2)
other = Point(1, 2)
print(point == other)
```

```
False
```

The reason we get False is that by default this equality operator compares the references or addresses of these two objects in memory. To solve this problem, we need a magic method. It's called when we compare two objects. https://rszalski.github.io/magicmethods/#comparisons

[14]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y


point = Point(1, 2)
other = Point(1, 2)
print(point == other)
```

```
True
```

What about the > operator?

[15]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y


point = Point(1, 2)
```

```
other = Point(1, 2)
print(point > other)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-6d6f688eab0b> in <module>
     10 point = Point(1, 2)
     11 other = Point(1, 2)
---> 12 print(point > other)

TypeError: '>' not supported between instances of 'Point' and 'Point'
```

The solution is to define another magic method.

[17]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __gt__(self, other):
        return self.x > other.x and self.y > other.y


point = Point(10, 20)
other = Point(1, 2)
print(point > other)
```

```
True
```

What if we change $>$ to $<$?

[18]:
```
print(point < other)
```

```
False
```

As you see, we don't have to explicitly implement each of these operators, when you implement the greater than magic method, Python will automatically figure out what to do if you use the less than operator.

1.7 Performing Arithmetic Operations

What if we want to do arithmetic operations on the two point objects? We need to take advantage of some numeric magic methods, see https://rszalski.github.io/magicmethods/#numeric .

[19]:
```
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)


point = Point(10, 20)
other = Point(1, 2)
print(point + other)
```

```
<__main__.Point object at 0x00000181881DA7C8>
```

Here, we remove the `__str__()` magic mtehod, so we only get an address. To show we have successfully added two points together,

[20]:
```
combined = point + other
print(combined.x, combined.y)
```

```
11 22
```

1.8 Making Custom Containers