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 0x000002B6197A8208>
```

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.

[16]:
```
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 $<$?

[17]:
```
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 .

[18]:
```
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 0x000002B619855F08>
```

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

[19]:
```
combined = point + other
print(combined.x, combined.y)
```

```
11 22
```

1.8 Making Custom Containers

We have learned lists, sets, dictionaries and so on, but there are times that we want to create our own custom container types. For example, here we have this class, *tagcloud*, with which we can keep track of the number of various tags on a blog. For example, how many articles do we have that are tagged with "python" or "javascript" and so on.

```
class TagCloud:
    .......

cloud = TagCloud()

len(cloud) # get the number of items in this container
cloud["python"] = 10 # get an item by its key, also set that to some number
for tag in cloud: # iterate over that container
    print(tag)
```

We want to achieve these operations supported in this custom container type. How to implement a class like this?

1.8.1 add

[20]:
```
class TagCloud:
    def __init__(self): # first, define the constructor
        self.tags = {} # we initialize the tags attribute to an empty dictionary

    def add(self, tag):
```

```
        '''
        we should check to see if we have this tag in our dictionary. If we␣
↪don't have it,         we are going to set its value to 1. Otherwise, we are␣
↪going to increase it by 1.
        '''
        self.tags[tag] = self.tags.get(tag, 0) + 1
        # we use 'get' method in dictionary to get an item by its key, and␣
↪supply a                  default value if we don't have that. Then we add 1␣
↪and set the value fot this              key.

cloud = TagCloud()
cloud.add("python")
cloud.add("python")
cloud.add("python")
print(cloud.tags)
```

```
{'python': 3}
```

Why we create a custom class instead of using a plain dictionary? Because we want to make it a little bit smarter than a typical dictionary. What if we add the 'python' tag with a capital 'P'? It's the same tag as the lowercase python.

[21]:
```
cloud = TagCloud()
cloud.add("python")
cloud.add("Python")
cloud.add("python")
print(cloud.tags)
```

```
{'python': 2, 'Python': 1}
```

That's not what we expect. We get two separate items. This is how a typical dictionary behaves. Let's take care of the case sensitivity as follows,

[22]:
```
class TagCloud:
    def __init__(self):
        self.tags = {}

    def add(self, tag):
        self.tags[tag.lower()] = self.tags.get(tag.lower(), 0) + 1


cloud = TagCloud()
cloud.add("python")
cloud.add("Python")
cloud.add("python")
print(cloud.tags)
```

```
{'python': 3}
```

We can see now we no long worry about the lowercase or uppercase characters. All that complexity is encapsulated in the 'TagCloud' class. It's not visible to the rest of our program.

Then, we want to read the count of a tag using square brackets like this, `cloud["python"]`. To do this, we need to implement a magic method, called `__getitem__()`.

1.8.2 getitem

[23]:
```
class TagCloud:
    def __init__(self):
        self.tags = {}

    def add(self, tag):
        self.tags[tag.lower()] = self.tags.get(tag.lower(), 0) + 1
        # this get method is built in a dictionary

    def __getitem__(self, tag): # __getitem(self, key)
        return self.tags.get(tag.lower(), 0) # if we don't have it, return 0.

cloud = TagCloud()
print(cloud["python"])
cloud.add("python")
cloud.add("Python")
cloud.add("python")
print(cloud["python"])
print(cloud.tags)
```

```
0
3
{'python': 3}
```

Now, as you can see, we can get the number of a given tag. We can't do this with a typical dictionary. If we don't have the 'python' tag, our typical dictionary will throw an error.

1.8.3 setitem

Let's take this to the next level. We also want to set a tag to some value. Another magic method called `__setitem__()` is needed.

[24]:
```
class TagCloud:
    def __init__(self):
        self.tags = {}

    def add(self, tag):
        self.tags[tag.lower()] = self.tags.get(tag.lower(), 0) + 1
        # this get method is built in a dictionary

    def __getitem__(self, tag): # __getitem(self, key)
        return self.tags.get(tag.lower(), 0) # if we don't have it, return 0.
```

```
    def __setitem__(self, tag, count): # __setitem(self, key, value)
        self.tags[tag.lower()] = count

cloud = TagCloud()
print(cloud["python"])
cloud["python"] = 10
cloud.add("python")
cloud.add("Python")

print(cloud["python"])
print(cloud.tags)
```

```
0
12
{'python': 12}
```

1.8.4 len

In order to be able to get the number of tags in this tag 'cloud', we should implement the `__len__()` magic method.

[25]:
```
class TagCloud:
    def __init__(self):
        self.tags = {}

    def add(self, tag):
        self.tags[tag.lower()] = self.tags.get(tag.lower(), 0) + 1
        # this get method is built in a dictionary

    def __getitem__(self, tag): # __getitem(self, key)
        return self.tags.get(tag.lower(), 0) # if we don't have it, return 0.

    def __setitem__(self, tag, count): # __setitem(self, key, value)
        self.tags[tag.lower()] = count

    def __len__(self):
        return len(self.tags) # len() is a built in function in Python

cloud = TagCloud()

cloud.add("java")
cloud.add("python")
cloud.add("Python")
print(len(cloud))
print(cloud.tags)
```

```
2
{'java': 1, 'python': 2}
```

1.8.5 iter

Finally, to make it iterable, so that we can iterate over it using a for loop, we need to implement another magic method, called `__iter__()`.

[26]:
```python
class TagCloud:
    def __init__(self):
        self.tags = {}

    def add(self, tag):
        self.tags[tag.lower()] = self.tags.get(tag.lower(), 0) + 1
        # this get method is built in a dictionary

    def __getitem__(self, tag): # __getitem(self, key)
        return self.tags.get(tag.lower(), 0) # if we don't have it, return 0.

    def __setitem__(self, tag, count): # __setitem(self, key, value)
        self.tags[tag.lower()] = count

    def __len__(self):
        return len(self.tags) # len() is a built in function in Python

    def __iter__(self):
        return iter(self.tags) # also a built in function, returns an iterator␣
↪object                                    which gives us one item at a␣
↪time in a for loop.

cloud = TagCloud()

cloud["matlab"] = 10
cloud.add("java")
cloud.add("python")
cloud.add("Python")
print(len(cloud))
print(cloud.tags)
for tag in cloud:
    print(f"{cloud[tag]} articles are tagged with {tag.title()}.")
```

```
3
{'matlab': 10, 'java': 1, 'python': 2}
10 articles are tagged with Matlab.
1 articles are tagged with Java.
2 articles are tagged with Python.
```

1.9 Private Members

The above 'TagCloud' class has a tiny problem.

[27]:
```
cloud = TagCloud()

cloud.add("python")
cloud.add("Python")
cloud.add("python")
print(cloud["PYTHON"])
```

```
3
```

The program works well. However, if we access the underlying dictionary of this class, our program is going to crash.

[28]:
```
print(cloud.tags)
```

```
{'python': 3}
```

[29]:
```
print(cloud.tags["PYTHON"])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-29-0e24b627caac> in <module>
----> 1 print(cloud.tags["PYTHON"])

KeyError: 'PYTHON'
```

We don't have this 'PYTHON' key in our dictionary, where everything is stored in lowercase. The problem with this class is that it gives us access to the underlying dictionary that is used to keep track of the count of text. To fix this problem, we need to hide this attribute from the outside, so we cannot access it.

[30]:
```
class TagCloud:
    def __init__(self):
        self.__tags = {}

    def add(self, tag):
        self.__tags[tag.lower()] = self.__tags.get(tag.lower(), 0) + 1
        # this get method is built in a dictionary

    def __getitem__(self, tag): # __getitem(self, key)
        return self.__tags.get(tag.lower(), 0) # if we don't have it, return 0.

    def __setitem__(self, tag, count): # __setitem(self, key, value)
        self.__tags[tag.lower()] = count

    def __len__(self):
```

```
        return len(self.__tags) # len() is a built in function in Python

    def __iter__(self):
        return iter(self.__tags) # also a built in function, returns an␣
↪iterator object                                        which gives us one␣
↪item at a time in a for loop.
```

Rename `self.tags` to `self.__tags` to make it private.

[31]:
```
cloud = TagCloud()

cloud.add("python")
cloud.add("Python")
cloud.add("python")

print(cloud.tags)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-31-77821a655675> in <module>
      5 cloud.add("python")
      6
----> 7 print(cloud.tags)

AttributeError: 'TagCloud' object has no attribute 'tags'
```

[32]:
```
print(cloud.__tags)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-32-43e4935a7f47> in <module>
----> 1 print(cloud.__tags)

AttributeError: 'TagCloud' object has no attribute '__tags'
```

Now we can see that we no long could access the underlying attribute 'tags'. So this is how we can make certain attributes or certain methods in a class private. **If you prefix them with __, they are private.**

Technically, these members are still accessible from the outside. It's just a little bie harder to access them. So the point of this practice is not security, it's more of a warning or alert to someone who is using this class. It's telling the consumer of this class, "hey, don't touch this, this is private." So how you can still access this?

[33]:
```
cloud = TagCloud()
```

```
cloud.add("python")
print(cloud.__dict__) # a dictionary that holds all the attributes in this class
```

```
{'_TagCloud__tags': {'python': 1}}
```

We can see in this class we have this attribute `_TagCloud__tags`.

[34]:
```
print(cloud._TagCloud__tags)
```

```
{'python': 1}
```

So in Python, unlike C# or Java, we don't really have the concept of private members. These private members are still accessible from the outside. Using double underscore is more of a convention to prevent accidential access of these private members.