classes

March 11, 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 that 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, and 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)
```

Now 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 `__eq__()`. It's called when we compare two objects. See https://rszalski.github.io/magicmethods/#comparisons for details.

[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
```

How 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 `__gt__()`.

[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 removed 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
print(cloud["python"]) # get the value of an item by its key
cloud["python"] = 10 #  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 a dictionary to get the value of an item by its key, and supply a default value if we don't have that. Then we add 1 and set the value for this key.

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

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

Why did 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") # a capitalized p
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 longer 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 because 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 object '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) # iter() is also a built in function, returning␣
 ↪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 its 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) # iter() is also a built in function,␣
↪returning 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 longer 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 bit harder to access them. So the point of this practice is not security, but 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 can you 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.

Mar 10, 2021, Yu Sun

1.10 Properties

There are times that you want to have control over an attribute of a class. For example, we have a product class as follows,

[1]:
```
class Product:
    def __init__(self, price):
        self.price = price


product = Product(-50)
```

If we construct an instance of this class as above with price being -50, Python interpreter will not send an error. This is not good. How can we prevent this? We want to make sure the price doesn't have a negative value.

One solution is to set the attribute private, and define two methods for getting and setting the value of this attribute.

[2]:
```
class Product:
    def __init__(self, price):
        self.set_price(price)

    def get_price(self):
        return self.__price

    def set_price(self, value):
        if value < 0:
            raise ValueError("Price cannot be negative.")
        self.__price = value


product = Product(-50)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-2-348d540e75a0> in <module>
```

```
     12
     13
---> 14 product = Product(-50)

<ipython-input-2-348d540e75a0> in __init__(self, price)
      1 class Product:
      2     def __init__(self, price):
----> 3         self.set_price(price)
      4
      5     def get_price(self):

<ipython-input-2-348d540e75a0> in set_price(self, value)
      8     def set_price(self, value):
      9         if value < 0:
---> 10             raise ValueError("Price cannot be negative.")
     11         self.__price = value
     12

ValueError: Price cannot be negative.
```

The above solution is kind of ugly. It's what we consider "unpythonic", which means it's not using python's best practices, and it's not using python's language features to the fullest potential. The above code is kind of code that Java programmer writes when learning Python. In Python, we have a better way to achieve the same result. That's when we use a **property**.

A property is an object that sits in front of an attribute and allows us to get or set the value of an attribute. How to define a property here?

[6]:
```
class Product:
    def __init__(self, price):
        self.set_price(price)

    def get_price(self):
        return self.__price

    def set_price(self, value):
        if value < 0:
            raise ValueError("Price cannot be negative.")
        self.__price = value

    price = property(get_price, set_price) # Note that we are not calling these␣
↪methods like get_price(), but just passing the references to them.

product = Product(10)
print(product.price) # get the value

product.price = - 50 # set the value
```

```
print(product.price)
```

```
10
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-15913ed850f3> in <module>
     16 print(product.price) # get the value
     17
---> 18 product.price = - 50 # set the value
     19 print(product.price)

<ipython-input-6-15913ed850f3> in set_price(self, value)
      8     def set_price(self, value):
      9         if value < 0:
---> 10             raise ValueError("Price cannot be negative.")
     11         self.__price = value
     12

ValueError: Price cannot be negative.
```

After we defined these two methods, we define a class attribute with an ideal name. In that case, we want to call that 'price'. Here we call the built-in *property* function. This function takes four parameters and all these parameters are optional. The first parameter is a function for getting the value of an attribute, the second parameter is a function for setting the value of an attribute, the third parameter is a function for deleting that attribute, and the last parameter is for documentaion.

So a property looks like a regular attribute from outside, but internally it has two methods that we call a getter and a setter. While this price property solves our problem, the two methods we wrote are still accessible. That is, if we type `product.`, we can still see `get_price` and `set_price` methods. These methods are poluting the interface of our object. We want our classes, or objects to have minimal number of functions or methods exposed to the outside. To hide these, one solution is to make them private by double underscore. A better way to achieve the same result is to use a decorator. Earlier we use a decorator, called class method `@classmethod` to convert an instance method to a class method. We have another decorator for creating a property.

[10]:
```
class Product:
    def __init__(self, price):
        self.price = price

    @property
    def price(self):
        return self.__price

    @price.setter
    def price(self, value):
        if value < 0:
```

```
            raise ValueError("Price cannot be negative.")
        self.__price = value


product = Product(50)
print(product.price)
product = Product(-10)
print(product.price)
```

```
50
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-10-6b53730598aa> in <module>
     16 product = Product(50)
     17 print(product.price)
---> 18 product = Product(-10)
     19 print(product.price)

<ipython-input-10-6b53730598aa> in __init__(self, price)
      1 class Product:
      2     def __init__(self, price):
----> 3         self.price = price
      4
      5     @property

<ipython-input-10-6b53730598aa> in price(self, value)
     10     def price(self, value):
     11         if value < 0:
---> 12             raise ValueError("Price cannot be negative.")
     13         self.__price = value
     14

ValueError: Price cannot be negative.
```

Instead of explicitly calling the property function to creat a property object, we can apply the property decorator to this method and rename this method with an ideal name,

```
@property
def price(self):
       return self.__price
```

In this case, we want to call it price. When Python interpreter see this code, it will automatically creat a property object called price.

Now similarly, we need to apply another decorator to the 'set_price' method. The name of that decorator starts with the name of our property, and then `.setter`. Also, we need to rename this method to 'price', so our implementation is clearner and less noisy. How this works internally is a

little bit complicated, and we're going to look at that later in the future. For now, all we need to know is that with these two decorators, we can easily create a property.

One more place to modify is that in our constructor, we no longer have `set_price`, but we can use our price property like a regular attribute. So we set `self.price = price`.

One last thing before finish this lecture. When defining properties, you don't always have to define a getter and a setter. In this case, if we comment out the section of 'set_price', we will have a read-only property. So we can only read the value of thie price. Once we set it, we cannot change it.

[14]:
```
class Product:
    def __init__(self, price):
        self.price = price

    @property
    def price(self):
        return self.__price

    # @price.setter
    # def price(self, value):
    #     if value < 0:
    #         raise ValueError("Price cannot be negative.")
    #     self.__price = value


product = Product(10)
print(product.price)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-14-ed68cd777d07> in <module>
     14
     15
---> 16 product = Product(10)
     17 print(product.price)

<ipython-input-14-ed68cd777d07> in __init__(self, price)
      1 class Product:
      2     def __init__(self, price):
----> 3         self.price = price
      4
      5     @property

AttributeError: can't set attribute
```

1.11 Inheritance

As you build various classes, you will notice that some of the classes may have one or more features or functions in common. Here is an example. All mammals should be able to eat and walk. All fish should be able to eat and swim.

[ ]:
```
class Mammal:
    def eat(self):
        print("eat")

    def walk(self):
        print("walk")


class Fish:
    def eat(self):
        print("eat")

    def swim(self):
        print("swim")
```

We find that 'eat' method appears in both classes. In real-world program, we will have a method of 5 or 10 lines of code, and we might have repeated that in multiple classes. This is bad, because if there is a bug in one of these methods, you have to fix it in multiple places. Or similarly, if you need to change the behavior of this method, you have to do it in multiple places. In programming, we have the concept of **DRY**, namely don't repeat yourself. To solve this problem, we have two solutions. We can use inheritance or composition. We talk about inheritance and will talk about composition later in this section.

Inheritance is a mechanism that allows us to define the common behavior or common functions in one class, and then inherit them in other classes. Here's how it works. Define a separate class called 'Animal', and then move the 'eat' method to this class. When defining its child or sub class, add a parentheses and input the parent or base class's name.

[15]:
```
class Animal:
    def eat(self):
        print("eat")

# Animal: Parent/ Base class
# Mammal: Child/ Sub class
class Mammal(Animal):
    def walk(self):
        print("walk")


class Fish(Animal):
    def swim(self):
        print("swim")
```

```
m = Mammal()
m.eat()
```

```
eat
```

This verifies that the mammal class inherits from the animal class. This inheritance is not limited to methods, but we can also inherit the attributes of a base class.

[16]:
```
class Animal:
    def __init__(self):
        self.age = 1

    def eat(self):
        print("eat")

# Animal: Parent/ Base class
# Mammal: Child/ Sub class
class Mammal(Animal):
    def walk(self):
        print("walk")


m = Mammal()
m.eat()
print(m.age)
```

```
eat
1
```

In the above code, when we create an mammal object, it will automatically have the age attribute initialized to 1.

1.12 The Object Class

First, we show some usefult functions.

[17]:
```
isinstance(m, Mammal)
```

[17]:
```
True
```

Obviously, this 'm' is an instance of 'Mammal' class. What if we pass 'Animal'?

[18]:
```
isinstance(m, Animal)
```

[18]:
```
True
```

We still get True. Because mammal inherits from animal, so an instance of the mammal class is also an animal. Actually, this animal class inherits from another class called 'object', even though

we didn't write it out explicitly. **We have a class called object, and that is the base class for all classes in Python**.

[19]:
```
isinstance(m, object)
```

[19]: True

[21]:
```
o = object() # create an empty object
# o.
# m.
```

If we type `o.`, we can see many magic methods, which are the magic methods that every class in Python has. So if we type `m.`, we can also see these magic methods for this instance of mammal class.

Another useful built-in function is as follows,

[22]:
```
issubclass(Mammal, Animal)
```

[22]: True

[23]:
```
issubclass(Animal, object)
```

[23]: True

[24]:
```
issubclass(Mammal, object)
```

[24]: True

1.13 Method Overriding

[ ]:
```
class Animal:
    def __init__(self):
        self.age = 1

    def eat(self):
        print("eat")


class Mammal(Animal):
    def walk(self):
        print("walk")
```

In the above code, the animal class has the constructor where we initialize the age attribute to 1. What if you want to add the constructor to the mammal class, and initialize its weight? Let's do that and see what happens.

[25]:
```
class Animal:
    def __init__(self):
        self.age = 1

    def eat(self):
        print("eat")


class Mammal(Animal):
    def __init__(self):
        self.weight = 2

    def walk(self):
        print("walk")
```

[27]:
```
m = Mammal()
print(m.weight)
print(m.age)
```

```
2
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-27-eab6fc7c251b> in <module>
      1 m = Mammal()
      2 print(m.weight)
----> 3 print(m.age)

AttributeError: 'Mammal' object has no attribute 'age'
```

The reason is because the constructor in the animal class was not executed. In other words, the constructor that was defined in the mammal class replaced the constructor in the base class. This is what we call **method overriding**. So we are overriding or replacing a method in the base class. What if we still want to execute the constructor in the animal class and initialize the age of an animal?

The way is to use the built-in **super** function in the constructor of the mammal class to get access to the super or base class. So we have an access to the animal class and we can call any of the methods in this class.

[28]:
```
class Animal:
    def __init__(self):
        print("Animal Constructor")
        self.age = 1

    def eat(self):
        print("eat")
```

```
class Mammal(Animal):
    def __init__(self):
        super().__init__() # by '.' now we can call any of the methods in this␣
 ↪class
        print("Mammal Constructor")
        self.weight = 2

    def walk(self):
        print("walk")


m = Mammal()
print(m.weight)
print(m.age)
```

```
Animal Constructor
Mammal Constructor
2
1
```

As you can see, first the constructor of the animal class was called, then the constructor of the mammal class was called. We can also change the order of these method calls.

[29]:
```
class Animal:
    def __init__(self):
        print("Animal Constructor")
        self.age = 1

    def eat(self):
        print("eat")


class Mammal(Animal):
    def __init__(self):
        print("Mammal Constructor")
        self.weight = 2
        super().__init__()

    def walk(self):
        print("walk")


m = Mammal()
print(m.weight)
print(m.age)
```

```
Mammal Constructor
Animal Constructor
2
1
```

So to recap, method overriding means replacing or extending a method defined in the base class. **Here, by `super().__init__()` we call the constructor of the base class**. If we didn't have this line, the constructor of the mammal class would completely replace the `__init__` method in the animal class.

1.14 Multi-level Inheritance

[30]:
```python
class Animal:
    def eat(self):
        print("eat")


class Bird(Animal):
    def fly(self):
        print("fly")
```

Inheritance is a good thing, which prevents code duplication and allows us to reuse code. In the above case, we are reusing the eat method in the base class. However, too much of a good thing is a bad thing. Because too much inheritance between classes can increase complexity and introduce various kinds of issues. Let's show an example.

[ ]:
```python
class Animal:
    def eat(self):
        print("eat")


class Bird(Animal):
    def fly(self):
        print("fly")


class Chicken(Bird):
    pass # pass is an empty statement that doesn't do anything, just making␣
 ↪Python interpreter happy
```

There is a problem in the above code. The chicken class inherits all the methods in the bird class, but a chicken cannot fly. So this is an example of **inheritance abuse**. Another example of inheritance abuse is around the concept of employees. Some people think the following relationship holds,
'Employee - Person - LivingCreature - Thing'.
This is what we call **multi-level inheritance**, which can significantly increase the complexity of your software. If you want to use inheritance, limit it in one or two levels.

1.15 Multiple Inheritance

In Python, a class can have multiple base classes. Here, we have the Manager that has two base classes. This is what we call **multiple inheritance**. Similar to multi-level inheritance, there is a source of issues. If you don't use it properly, you're going to introduce all sorts of bugs in your programs.

[32]:
```
class Employee:
    def greet(self):
        print("Employee Greet")


class Person:
    def greet(self):
        print("Person Greet")


class Manager(Employee, Person):
    pass
```

In the two above base classes, we have same method with different print messages. Now let's see what happens if you creat an object of the manager class and call the greet method.

[33]:
```
manager = Manager()
manager.greet()
```

```
Employee Greet
```

This is because we added the employee class first in `class Manager(Employee, Person)`. When Python interpreter executes the greet method, first it looks at the manager class to see if it has a method called greet. If it doesn't, it will look at its first base class. So on, so forth.

Why is there an issue? If we move the person class in front of the employee class, we get a different output.

[34]:
```
class Manager(Person, Employee):
    pass


manager = Manager()
manager.greet()
```

```
Person Greet
```

Multiple inheritance is not always a bad thing. It's bad if you don't use it properly. If the base classes are small classes and they have nothing in common, and you want to inherit their features in a separate class, that's perfectly fine to use multiple inheritance. Things start to get complicated when these classes have things in common, like the greet method here. Let's show a good example of multiple inheritance.

[ ]:
```
class Flyer:
    def fly(self):
        pass


class Swimmer:
    def swim(self):
        pass


class FlyingFish(Flyer, Swimmer):
    pass
```

Mar 11, 2021, Yu Sun

1.16 A Good Example of Inheritance

Let's imagine we want to model the concept of a stream of data. We can read a stream of data from a file, from a network, or from the memory. All these streams have a few things in common. We can open them, close them and read data from them. But how we read data from a stream is dependent upon the type of the stream, because reading data from a file is different from reading it from a network. Let's start by defining a base class called Stream, in which we'll define methods like open and close.

[1]:
```
class InvalidOperationError(Exception):
    pass


class Stream:
    def __init__(self):
        self.opened = False

    def open(self):
        if self.opened:
            raise InvalidOperationError("Stream is already open.")
        self.opened = True

    def close(self):
        if not self.opened:
            raise InvalidOperationError("Stream is already closed.")
        self.opened = False


class FileStream(Stream):
    def read(self):
        print("Reading data from a file.")
```

```
class NetworkStream(Stream):
    def read(self):
        print("Reading data from a network.")
```

We need a flag to know if the stream is open or not. So we can define a constructor and set the 'opened' flag initially to False. When we call open method, we can set this flag to True. What if we try to open a stream that is already open? This is an invalid operation. So we want to raise an exception. What kind of exception? We don't want to do a value error, because here we're not dealing with a value. In this example, we can create a custom exception called InvalidOperationError. We don't have this in Python. So, on the top, let's create another class InvalidOperationError. We want to derive this from the base exception class in Python. So everytime you want to create a custom exception, you should write your class from the exception class. Here we don't need any code, so we simply add a pass statement. Back to our open method, we can now raise this InvalidOperationError with a friendly meseage "Stream is already open."

Similarly, we can define the close method. So, open and close methods are the common features that we need in every stream. Now let's go ahead and implement the file stream by inheritance. Finally, a network stream class is constructed by inheritance.

This is a good example of inheritance. First of all, we don't have multi-level inheritance. Also, we don't have multiple inheritance. Our subclasses don't have multiple parents.

1.17 Abstract Base Classes

We are going to continue with the example from the last lecture. There are a couple of issues with this implementaion. The first issue is that we can create a stream object and call the open method. Why is this an issue? Because this stream class is an abstract concept. What does it mean to open a stream? We shouldn't be able to directly create an instance of the stream class. We should always subclass it and then create an instance of the subclass. **So we only create this stream class as a base class to provide some code that we're going to reuse across different kinds of streams.** Now the second issue. If you look at the implementaion of the file stream and network stream classes, you can see both these classes have a read method. If tomorrow we decide to create a different kind of stream, we should remember to implement this read method, call it exactly read. If you call it read line or read str or read whatever, it's not going to be consistent with the other kinds of streams we have here. In other words, currently there is no way to enforce a common interface across different kinds of streams. This is more of a convention we have used here. **It would be nice to have a common contract or common interface across these different types of streams.** So how can we solve these problems? That's when we use an **abstract base class**.

An abstract base class is like a half baked cookie. Its purpose is to provide some common code to its derivatives. So here we want to make this stream class an abstract base class. To do that, we go on the top, and from abc module, which stands for abstract base class, we import the ABC class and abstract method function, which we are going to use as a decorator. The name of the module is all lowercase, and the name of the class is all uppercase.

[3]:
```
from abc import ABC, abstractmethod

class InvalidOperationError(Exception):
    pass


class Stream(ABC):
    def __init__(self):
        self.opened = False

    def open(self):
        if self.opened:
            raise InvalidOperationError("Stream is already open.")
        self.opened = True

    def close(self):
        if not self.opened:
            raise InvalidOperationError("Stream is already closed.")
        self.opened = False

    @abstractmethod
    def read(self):
        pass


class FileStream(Stream):
    def read(self):
        print("Reading data from a file.")


class NetworkStream(Stream):
    def read(self):
        print("Reading data from a network.")
```

Now, to make this stream class abstract, we should derive it from the ABC class. The second step is to define the common interface for all streams. We want all streams to have a read method, and potentially a write method in the future. So, in the stream class, we define a read method. This method has no implementaion. Now we need to decorate this method with abstract method decorator. With these two simple steps, we fix both the problems that we have talked about. Now, try to run the program as follows,

[4]:
```
stream = Stream()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-d2384bce04d7> in <module>
----> 1 stream = Stream()
```

```
TypeError: Can't instantiate abstract class Stream with abstract methods read
```

Basically when a class has an abstract method, it's considered as an abstract class, and we cannot instantiate it, which means we cannot create an instance of it. So we get the above TypeError.

Now let's look at the second problem. Let's say tomorrow we are going to create a new kind of stram, called memory stream. We create it from the Stream class. For now let's just pass and create a memory stream object.

[5]:
```
class MemoryStream(Stream):
    pass


stream = MemoryStream()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-32a04c00193f> in <module>
      3
      4
----> 5 stream = MemoryStream()

TypeError: Can't instantiate abstract class MemoryStream with abstract methods␣
 ↪read
```

What's going on here? In our Stream class, we defined an abstract class method called read. If a class derives from this Stream class, it has to implement this method, otherwise that class would also be considered abstract. So in this example, our new class MemoryStream is also abstract. If you want to make it a concrete class, so we can instantiate it, we'll have to implement the read method. So define read and here we print "Reading data from a memory."

[8]:
```
class MemoryStream(Stream):
    def read(self):
        print("Reading data from a memory.")


stream = MemoryStream()
stream.open()
stream.read()
```

```
Reading data from a memory.
```

Now, this MemoryStream is a concrete class. We can instantiate it here, and it also follows the contract or the interface of the Stream class. So all our streams now have a read method.

1.18 Polymorphism

As another example here I have defined an abstract base class called UIControl. This class has an abstract method called draw and this method had no implementation. So this class only defines the contract or the interface that all its derivatives should follow.

[9]:
```
from abc import ABC, abstractmethod

class UIControl(ABC):
    @abstractmethod
    def draw(self):
        pass


class TextBox(UIControl):
    def draw(self):
        print("TextBox")


class DropDownList(UIControl):
    def draw(self):
        print("DropDownList")


def draw(control):
    control.draw()
```

Then we have two classes derived from UIControl. One is TextBox and the other is DropDownList. Both these classes implement the draw method but print different messages. Now, here we also have a function called draw that takes a UIControl object and calls the draw method on it. So, with this, we can create a drop down list object and pass it to this draw function.

[11]:
```
ddl = DropDownList()
print(isinstance(ddl, DropDownList))
draw(ddl)
```

```
True
DropDownList
```

What if we pass a text box to this function?

[13]:
```
textbox = TextBox()
draw(textbox)
```

```
TextBox
```

What is the point of this? I would change the draw function. Instead of getting a control object, I want it to get a list or a tuple of controls.

[14]:
```
from abc import ABC, abstractmethod

class UIControl(ABC):
    @abstractmethod
    def draw(self):
        pass


class TextBox(UIControl):
    def draw(self):
        print("TextBox")


class DropDownList(UIControl):
    def draw(self):
        print("DropDownList")


def draw(controls):
    for control in controls:
        control.draw()
```

Now we are going to pass a list of two objects. Let's run the program and see what happens.

[15]:
```
ddl = DropDownList()
textbox = TextBox()
draw([ddl, textbox])
```

```
DropDownList
TextBox
```

So using this approach, we can render the user interface of an application. Imagine you have a form with a bunch of text boxes, drop down lists, radio buttons and so on. You could have a list of all these objects, and pass that list to a function like draw. That function would take care of rendering the entire form. What is interesting here, is that our draw function doesn't know what kind of control it's working with. This is determined at run time. It simply iterates over the list of controls and calls the draw method of each control object. This is what we call **polymorphism**. "Poly" means many, "morph" means forms. So **polymorphism means many forms**. In this example, our draw method is taking many different forms, and this is determined at run time.

1.19 Duck Typing

What you saw in the last section was a classic example of polymorphism. To achieve polymorphic behavior, you start by defining the base class. And in this class, we define the common behavior or the common method that we need in its derivatives or children. Then we achieve polymorphic behavior in the draw function. So depending on the type of control object that you're working with at run time, this draw method takes different form. This is how polymorphism works in pretty much all languages that support classes. But because Python is a dynamically type language, we don't necessarily need this UIControl as the base class.

[ ]:
```python
from abc import ABC, abstractmethod

class TextBox:
    def draw(self):
        print("TextBox")


class DropDownList:
    def draw(self):
        print("DropDownList")


def draw(controls):
    for control in controls:
        control.draw()
```

With this change, we can still achieve polymorphic behavior. As long as the controls object in the draw function is iterable and each individual item has a draw method. This is what we call **duck typing**. So if it walks like a duck, and quacks like a duck, it is a duck. That is how Python works, because Python is a dynamically type language and it doesn't check the type of the objects. It only looks for the existence of certain methods in our objects.

Having that UIControl as an abstract base class is a good practice because it enforces a common interface or a common contract across all its derivatives. With this, we'll make sure that every kind of UIControl will have a draw method.

1.20 Extending Built-in Types

In Python, we can also use inheritance with the built-in types. For example, we can create a class of text, and have it inherit from the built-in string class. With this, our text class will inherit all the features of Python strings, but we can also add additional features to it. For example, we can add the ability to summarize it or duplicate it, and so on. So let's define a method called duplicate.

[19]:
```python
class Text(str):
    def duplicate(self):
        return self + self


text = Text("Python")
print(text.lower())
print(text.duplicate())
```

```
python
PythonPython
```

Here, we are concatenating a string with itself. As another example, we can also extend python lists. So let's define a new class, call it TrackableList, inherited from the built-in list class.

[20]:
```
class TrackableList(list):
    def append(self, object):
        print("Append called")
        super().append(object)


list = TrackableList()
list.append("1")
```

```
Append called
```

Here, we are going to override the append method. So every time we want to append an object to this list. We want to print a message on the terminal, perhaps for logging. Now we want to call the append method of the super class, namely the base class. Pass this object. So here, technically we are extending the append method of the list class. We are not replacing it. Now we can create a list object using TrackableList. Call the append method, and when we run the program, we can see this message on the terminal. So as you see, extending built-in types in Python is really easy.

1.21 Data Classes

As you work on larger program, you may come across classes that don't have any behavior. They don't have any methods. They only have data. Here is an example.

[22]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y


p1 = Point(1, 2)
p2 = Point(1, 2)
print(p1 == p2) # these two objects are stored in different lications in memory
print(id(p1))# this id() built-in function returns the address of the memory␣
 ↪location where an object is stored.
print(id(p2))
```

```
False
2186655008648
2186655009992
```

The above Point class doesn't have any methods. By default, Python compares objects based on where they are stored in memory. So to solve the issue with comparing point objects, we need to come back to the Point class and implement a magic method `__eq__()`.

[23]:
```
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

p1 = Point(1, 2)
p2 = Point(1, 2)
print(p1 == p2) # these two objects are stored in different lications in memory
```

```
True
```

However, writing all this code for data classes is a little bit tedious. If you're dealing with classes that have no behavior, no methods, and only have data, you can use a named tuple instead. Let me show you how that works.

[32]:
```
from collections import namedtuple

Point = namedtuple("Point", ["x", "y"])

p1 = Point(x=1, y=2)
p2 = Point(x=1, y=2)
print(p1 == p2)
```

```
True
```

From the collections module, we should import namedtuple function. We call this function. As a first argument, we specify the name for this new type we want to create. We're going to call that Point. Now as a second argument, we pass an array of attribute names. So this returns a named tuple that we can store. Name it to Point, which represents a type, like a class. So we can call it to create new point objects p1 and p2. However, instead of passing numbers only, we should pass keyword arguments.

The first improvement here is that our code is more clear and less ambiguous. We don't have to wonder what are these values. It's easy to guess 1 and 2 represent x and y here. The second benefit is that we don't have to explicitly implement a magic method to compare two objects.

So if you're working with classes that have only data and no methods, you might want to use a named tuple instead. You will write less code, and these named tuples are better than regular tuples, because here we'll have attributes in this Point object just like the attributes we have in our classes. So if we print `p1.x`, we see 1.

[33]:
```
print(p1.x)
```

```
1
```

Just be aware that these named tuples are immutable, which means once we create them we cannot modify them.

[34]:
```
p1.x = 10
print(p1.x)
```

```
---------------------------------------------------------------------------
```

```
AttributeError                            Traceback (most recent call last)
<ipython-input-34-757510bf33eb> in <module>
----> 1 p1.x = 10
      2 print(p1.x)

AttributeError: can't set attribute
```

So if you really need to modify these values, you need to create a new Point object.

[35]:
```
p1 = Point(x=1, y=2)
p1 = Point(x=10, y=2)
p2 = Point(x=1, y=2)
print(p1 == p2)
```

```
False
```