第五讲
类与特殊变量
Lecture 5
Classes and Special Variables

明玉瑞 Yurui Ming

yrming@gmail.com

声明
Disclaimer

- 本讲义在准备过程中由于时间所限，所用材料来源并未规范标示引用来源。所引材料仅用于教学所用，作者无意侵犯原著者之知识版权，所引材料之知识版权均归原著者所有；若原著者介意之，请联系作者更正及删除。

  The time limit during the preparation of these slides incurs the situation that not all the sources of the used materials (texts or images) are properly referenced or clearly manifested. However, all materials in these slides are solely for teaching and the author is with no intention to infringe the copyright bestowed on the original authors or manufacturers. All credits go to corresponding IP holders. Please address the author for any concern for remedy including deletion.

类
Classes

- 面向对象编程是编写软件最有效的方法之一。在面向对象编程中，类代表了对现实世界事物和情况的抽象，我们基于这些类创建对象。当编写一个类时，通常定义了整个对象类别具有的一般行为。在应用中，会使用从类中实例化的对象。

  Object-oriented programming is one of the most effective approaches to writing software. In object-oriented programming classes represent the abstractions of real-world things and situations, and one create objects based on these classes. When write a class, we define the general behaviour that a whole category of objects can have. For application, we mainly work with instances of a class, aka, instantiate an object from a class.

- 在Python中，类由关键字 class 创建。

  In Python, Classes are created by keyword class.

类
Classes

- 我们以一个狗的类实例代码来讲解Python中类相关的知识。

  We dissect the knowledge of classes in Python exemplified by a dog class.

```
❶ class Dog:
❷     """A simple attempt to model a dog."""

❸     def __init__(self, name, age):
          """Initialize name and age attributes."""
❹         self.name = name
          self.age = age

❺     def sit(self):
          """Simulate a dog sitting in response to a command."""
          print(f"{self.name} is now sitting.")

      def roll_over(self):
          """Simulate rolling over in response to a command."""
          print(f"{self.name} rolled over!")
```

类
Classes

- 上面的代码创建了一个简单的狗的类，它代表对狗的概念的抽象。对大多数狗来说，它们可以有名字和年龄，同时，狗也有一些行为，如坐着和翻滚。这些信息（名字和年龄）和这些行为（坐着和翻滚）将被狗的类包含，因为它们对大多数狗来说都很常见。这个示例展示了如何在Python中编写一个代表狗的类。当编写完狗的类后，可以使用它来创建单独的实例，每个实例代表一只特定的狗。

  The code example above illustrate how to create a simple class, Dog, that represents an abstraction to dogs. It is not about one dog in particular, but traits shared by any dog. For most dogs, well, they can have a name and age. Most dogs can have some common behaviours such as sitting and rolling over. Those pieces of information (name and age) and those behaviours (sit and roll over) will go in the Dog class because they’re common to most dogs. This example class shows how to make a class representing dogs in Python. After the class is written, it can be used to make individual instances, each of which represents one specific dog.

类
Classes

- 通常将作为类的一部分的函数称为成员方法，因此之前学到的关于函数的知识也适用于方法；我们稍后会讲调用方法的方式。注意，在我们的类的，有一个特殊的 __init__() 方法。每当我们基于 Dog 类创建新实例时，Python 都会自动调用该方法，用于做一些成员变量的初始化工作。此方法有两个前导下划线和两个尾随下划线，这种约定有助于避免Python默认的方法名称与用户自定义的方法名称发生冲突。因此要确保在 __init__() 的每一侧使用两个下划线。如果命名与这种形式不符，则在使用类时不会自动调用该方法，可能会导致难以识别的错误 。

  Generally, a function that’s part of a class is also called a member method or member function. Everything learned previously about functions applies to methods as well; the only practical difference is the way they called invoked, which will be covered later. Pay attention here to the special __init__() method. It gets run by Python automatically whenever a new instance based on the Dog class is created. This method has two leading underscores and two trailing underscores, a convention that helps prevent Python’s default method names from conflicting with user-defined method names. Make sure to use two underscores on each side of __init__(). For any mis-matched form, the method won’t be called automatically and can result in errors that are difficult to identify.

类
Classes

- 注意__init__方法中定义的两个变量都有前缀self，这些变量称为成员变量。以self为前缀的变量可用于类中的各个方法，并且可以通过从该类创建的任何实例访问这些变量。语句self.name = name 获取与参数名称关联的值并将其赋值给变量，然后将其关联到正在创建的实例上。对于语句self.age = age 也是同样的过程。有时，也称通过实例来访问的变量称为实例的属性。

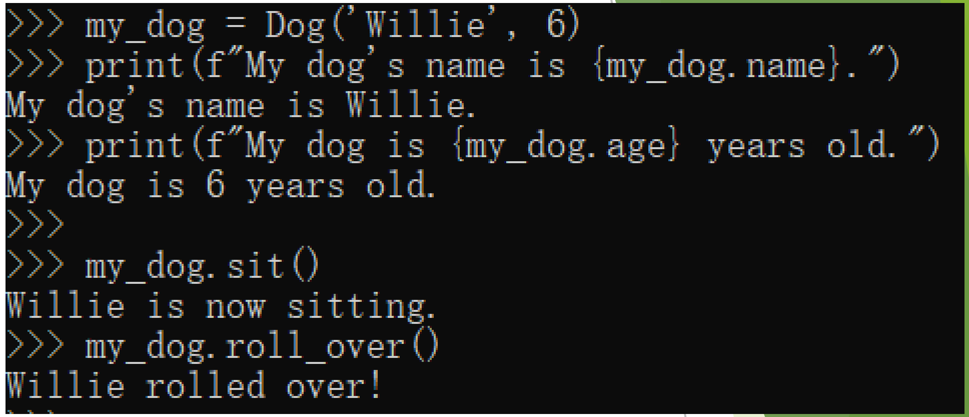
  The two variables defined in __init__ method each have the prefix self, and these variable is sometimes call member variables. Any variable prefixed with self is available to every method in the class and is also accessible through any instance created from the class. The line self.name = name takes the value associated with the parameter name and assigns it to the variable name, which is then attached to the instance being created. The same process happens with self.age = age. Variables that are accessible through instances are also called attributes..

类
Classes

- Dog类还定义了另外两个方法：sit() 和 roll_over() 。因为这些方法不需要额外的参数来运行，因此定义时只有一个参数，即self。这些方法可以通过创建实例来访问。当前定义中，sit() 和 roll_over() 并没有做太多的事情，只是打印一条消息，报告狗正在坐着或翻身的状态。但是这个概念可以扩展到现实情况：如果这个类是实际计算机游戏的一部分，这些方法可以包含让动画狗坐下和翻身的代码。如果编写此类用于控制机器人，则这些方法将指导导致机器狗坐下和翻身的运动。

  The Dog class has two other methods defined: sit() and roll_over(). Because these methods don’t need additional information to run, one parameter, namely, self is supplied in definition. The instances created later can access to these methods. For now, sit() and roll_over() don’t do much. They simply print a message saying the dog is sitting or rolling over. But the concept can be extended to realistic situations: if this class were part of an actual computer game, these methods would contain code to make an animated dog sit and roll over. If this class was written to control a robot, these methods would direct movements that cause a robotic dog to sit and roll over.

类
Classes



```
>>> my_dog = Dog('Willie', 6)
>>> print(f"My dog's name is {my_dog.name}.")
My dog's name is Willie.
>>> print(f"My dog is {my_dog.age} years old.")
My dog is 6 years old.
>>>
>>> my_dog.sit()
Willie is now sitting.
>>> my_dog.roll_over()
Willie rolled over!
```

- 在如下的示例代码中，我们用在上一个示例中刚刚编写的Dog类创建了一个名为“Willie”且年龄为6岁的狗的实例。正如前面所讲，调用类名会触发__init__()方法被调用并返回创建的实例，该实例分配给了变量my_dog。注意这里的命名约定：通常像 Dog 这样的大写名称指代一个类，而像 my_dog 这样的小写名称指代从类创建的单个实例。

  In the example below, a dog whose name is 'Willie' and whose age is 6 is instantiated from the Dog class wrote in the previous example. Write the class name in an functioning invoking way will trigger the __init__() method to be called and then an instance representing this particular dog is returned. That instance is assigned to the variable my_dog. The naming convention is helpful here: a capitalized name like Dog refers to a class, and a lowercase name like my_dog refers to a single instance created from a class.

类
Classes

- 正如示例程序中访问my_dog的名字属性的方式那样，可以使用点表示法访问实例的属性。同样，从类创建实例后，可以使用实例名加点再加方法名的表示法来，调用类中定义的任何方法。另外，可以根据需要从类中创建任意数量的实例 。

  The dot notation can be used to access the attributes of an instance, as done in the example to access the value of my_dog’s attribute name. In the same manner, after an instance is created from the class Dog, dot notation can be used to call any method defined in the class. Further, one can create as many instances from a class as needed.

```
my_dog = Dog('Willie', 6)
your_dog = Dog('Lucy', 3)
```

类
Classes

- 使用类的一个好处是编写类时，不必总是从头开始做所有的工作。如果正在编写的类是业已编写的另一个类的特殊情况或专用版本，则可以使用继承来简化工作。当一个类从另一个类继承时，它可以使用原先类的属性和方法。原类称为父类，新类称为子类。子类可以继承其父类的任何或所有属性和方法，但也可以自由定义自己的新属性和方法。

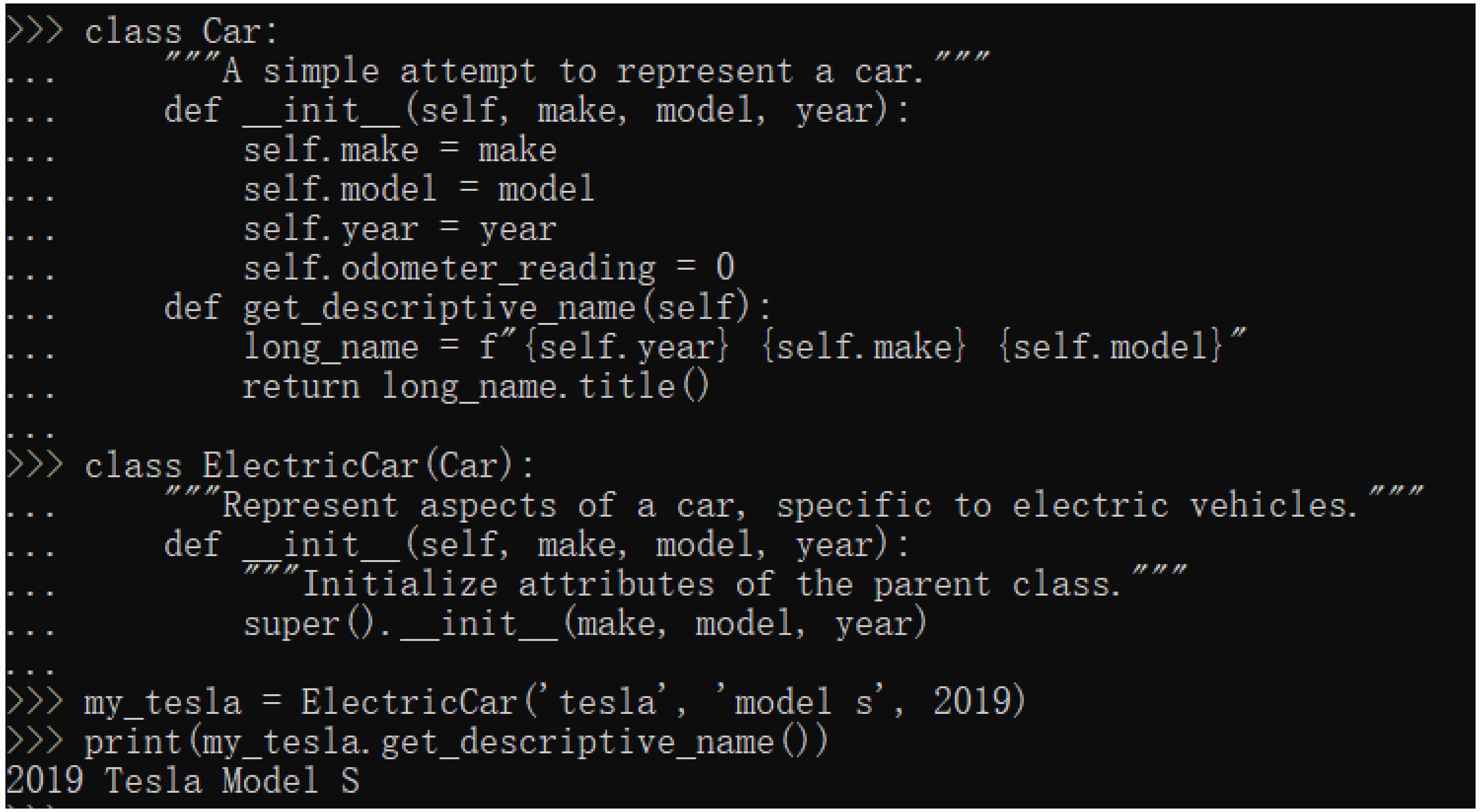
  An advantage for adopting class is that it is not always necessary to start from scratch when writing a class. If the class to be implemented happens to be a specialized version of another class, one can use inheritance. When one class inherits from another, it takes on the attributes and methods of the first class. The original class is called the parent class, and the new class is the child class. The child class can inherit any or all of the attributes and methods of its parent class, but it’s also free to define new attributes and methods of its own.

类
Classes

- 虽然在Python中，属性可以在任何方法中添加，但一般都会集中在__init__方法中创建并作初始化。因此，子类的任务之一是初始化在父类的__init__()方法中定义的属性，以保证它们在子类中可用。当基于现有类编写新类时，通常通过super关键字调用父类的__init__()方法。

  Although for attributes they can be dynamically added to the instance in any method inside a class definition, however, they tend to be created and initialized in the __init__ method. Hence, one task for the child class is to initialize any attributes that were defined in the parent __init__() method so as to make them available in the child class. When writing a new class based on an existing class, the keyword super is used to call the __init__() method of the parent class from the child class.

类
Classes

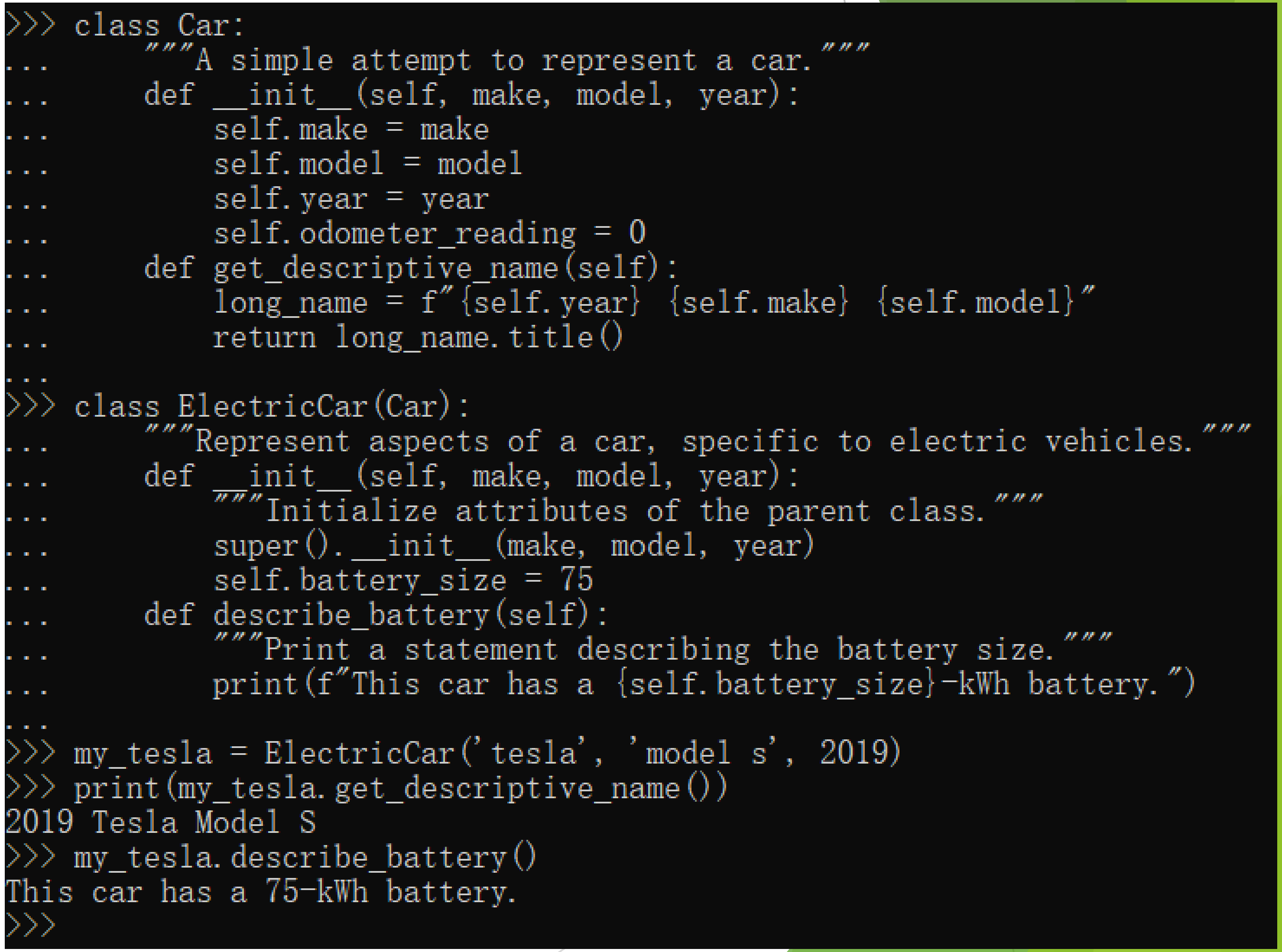


```
>>> class Car:
...     """A simple attempt to represent a car."""
...     def __init__(self, make, model, year):
...         self.make = make
...         self.model = model
...         self.year = year
...         self.odometer_reading = 0
...     def get_descriptive_name(self):
...         long_name = f"{self.year} {self.make} {self.model}"
...         return long_name.title()
...
>>> class ElectricCar(Car):
...     """Represent aspects of a car, specific to electric vehicles."""
...     def __init__(self, make, model, year):
...         """Initialize attributes of the parent class."""
...         super().__init__(make, model, year)
...
>>> my_tesla = ElectricCar('tesla', 'model s', 2019)
>>> print(my_tesla.get_descriptive_name())
2019 Tesla Model S
```

类
Classes

- 有时候，需要在子类中添加必要的新的属性和方法来区分子类和父类。下面示例中，我们添加一个特定于电动汽车的属性（例如电池）和一个报告该属性的方法：

  Now and then, we need to add new attributes and methods necessary to differentiate the child class from the parent class. In the following example, an attribute that’s specific to electric cars (a battery, for example) and a method to report on this attribute are added:

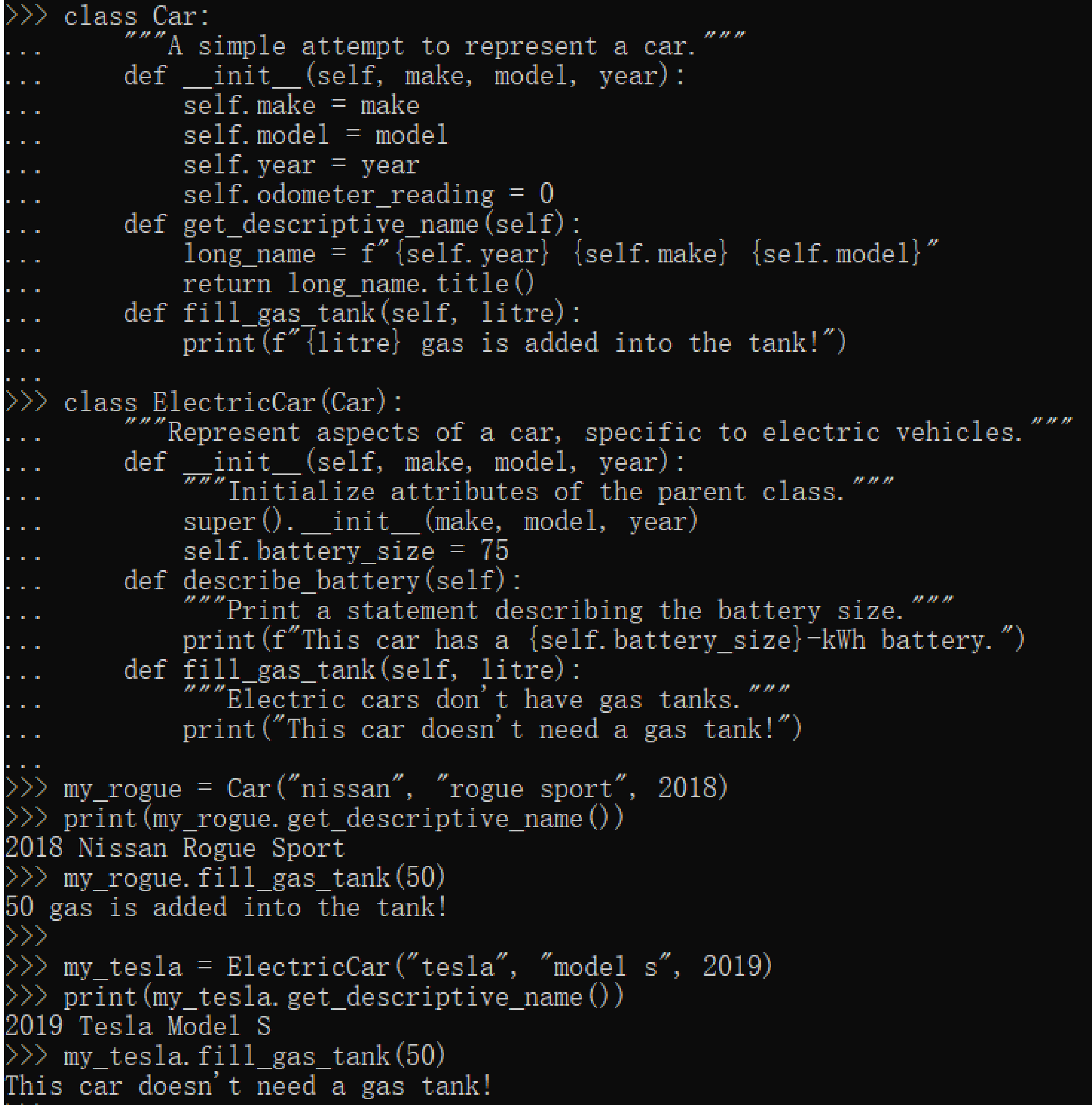


```
>>> class Car:
...     """A simple attempt to represent a car."""
...     def __init__(self, make, model, year):
...         self.make = make
...         self.model = model
...         self.year = year
...         self.odometer_reading = 0
...     def get_descriptive_name(self):
...         long_name = f"{self.year} {self.make} {self.model}"
...         return long_name.title()
...
>>> class ElectricCar(Car):
...     """Represent aspects of a car, specific to electric vehicles."""
...     def __init__(self, make, model, year):
...         """Initialize attributes of the parent class."""
...         super().__init__(make, model, year)
...         self.battery_size = 75
...     def describe_battery(self):
...         """Print a statement describing the battery size."""
...         print(f"This car has a {self.battery_size}-kWh battery.")
...
>>> my_tesla = ElectricCar('tesla', 'model s', 2019)
>>> print(my_tesla.get_descriptive_name())
2019 Tesla Model S
>>> my_tesla.describe_battery()
This car has a 75-kWh battery.
>>>
```

类
Classes

- 有时候，父类中定义的方法可能不使用于子类。为此，可以通过在子类中定义与父类中同名的方法来覆盖父类中的方法。假设Car类有一个名为fill_gas_tank()的方法。这种方法对纯电动汽车没有意义，因此可能需要覆盖此方法：

  Sometimes, a method from the parent class doesn’t fit the child class. To this, one can define a method in the child class with the same name as the method in the parent class in order to override it in the parent class. Say the class Car had a method called fill_gas_tank(). This method is meaningless for an all-electric vehicle, so it is needed to override this method:



```
>>> class Car:
...     """A simple attempt to represent a car."""
...     def __init__(self, make, model, year):
...         self.make = make
...         self.model = model
...         self.year = year
...         self.odometer_reading = 0
...     def get_descriptive_name(self):
...         long_name = f"{self.year} {self.make} {self.model}"
...         return long_name.title()
...     def fill_gas_tank(self, litre):
...         print(f"{litre} gas is added into the tank!")
...
>>> class ElectricCar(Car):
...     """Represent aspects of a car, specific to electric vehicles."""
...     def __init__(self, make, model, year):
...         """Initialize attributes of the parent class."""
...         super().__init__(make, model, year)
...         self.battery_size = 75
...     def describe_battery(self):
...         """Print a statement describing the battery size."""
...         print(f"This car has a {self.battery_size}-kWh battery.")
...     def fill_gas_tank(self, litre):
...         """Electric cars don't have gas tanks."""
...         print("This car doesn't need a gas tank!")
...
>>> my_rogue = Car("nissan", "rogue sport", 2018)
>>> print(my_rogue.get_descriptive_name())
2018 Nissan Rogue Sport
>>> my_rogue.fill_gas_tank(50)
50 gas is added into the tank!
>>>
>>> my_tesla = ElectricCar("tesla", "model s", 2019)
>>> print(my_tesla.get_descriptive_name())
2019 Tesla Model S
>>> my_tesla.fill_gas_tank(50)
This car doesn't need a gas tank!
```

类
Classes

- 虽然编译型语言与脚本语言都有类的概念，即用户自定义类型的概念，但对类型的维护与使用，一般不同。在编译型语言中，类型主要用来指导编译器对所要处理的数据根据类型进行组织访问，指导链接器根据签名链接中间文件生成可执行文件。在最后的文件中，用户自定义的类型概念一般得不到体现。而脚本语言不同，则需要一定的机制维护类型信息，脚本代码解释执行的过程中要随时参考这个类型信息。在Python中，类是作为一个Python解释器的一个内部对象来管理的，因此说当说类对象与实例对象时，读者不必疑惑。

  The maintenance and usage of user-defined types (UDT) such class (if it is supported) can be different between compiled languages and script languages. In compiled language, UDT is used to guide the compiler for organizing and processing the data, to instruct linkers to stitch intermediate files into executable according to signatures. UDT generally loses its trace in the final executable. But for script language such as Python, it requires mechanism to maintain the UDT, since UDT is frequently referenced to interpret and execute the script as long as in proper scopes. Classes are actually implemented as internal objects in Python interpreter, so next time, when people say class object or instance object, the users should not feel confused.

类
Classes

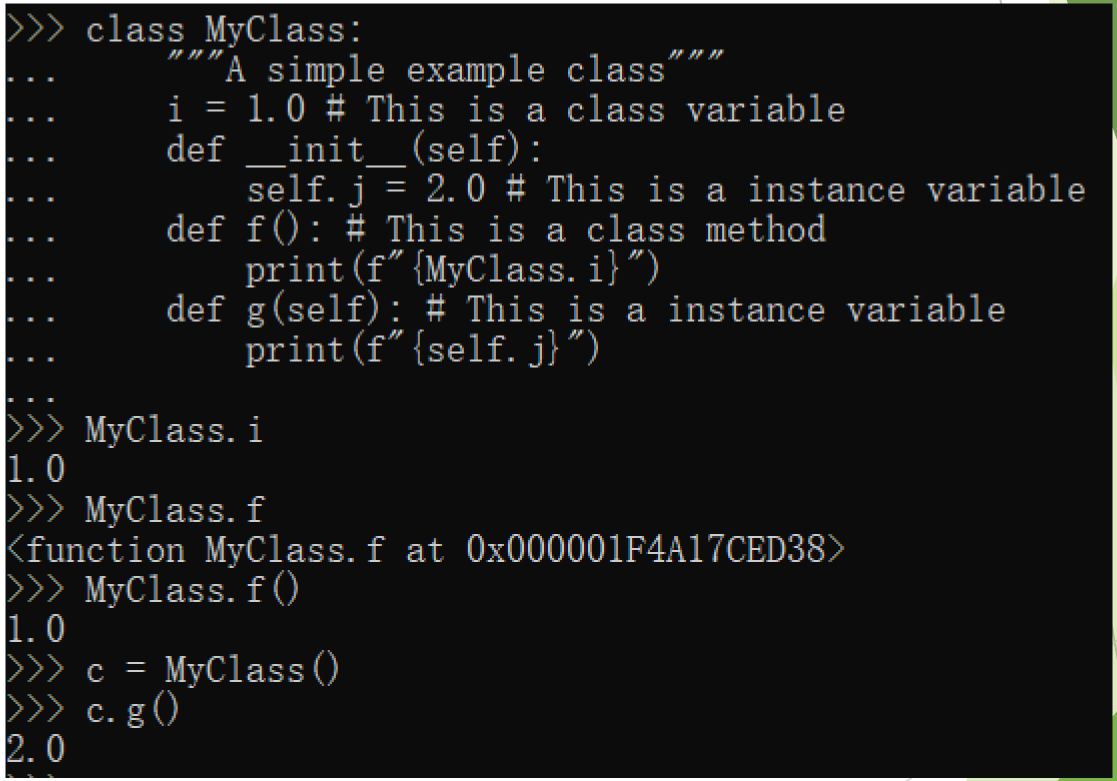
- 由于类本身被实现成内部对象，则我们不难推出，在Python中，类除了支持实例化之外，还支持属性引用。在C++中，我们有类变量与成员变量（或者实例变量）的概念，也有类函数与成员函数的概念。相同的概念在Python中也有，只不过由于内部实现为类，则对类变量的访问，与对实例变量的访问没有差别，而在C++中，对类变量的访问实际上是一种限定访问而已（用双冒号而不是点号）。

  Since the class is implemented as internal object in Python, so besides supporting instantiation, the class also support attribute reference. In C++, we have class variables and member variables (or instance variables), as well as class functions and member functions (instance functions or methods). However, the counterparts in Python behave a little different in Python than in C++. In Python, to access class variables and methods make no difference to accessing instance variables and methods. In C++ we use the dot syntax to access instance variables or method, while we use double colons to access class variables and methods. Actually, this is a sort of qualified assessment.

类
Classes

- Python中，类变量的定义位于任何成员函数之外，类函数的定义与通常函数没有什么区别。与成员函数不同之处在于，其不需要self关键字。

  In Python, definition of any class variable is done at any place out of the member functions, and definition of any class method makes no difference with general functions. The distinct between class methods and instance methods is that class methods are without self as the first argument.

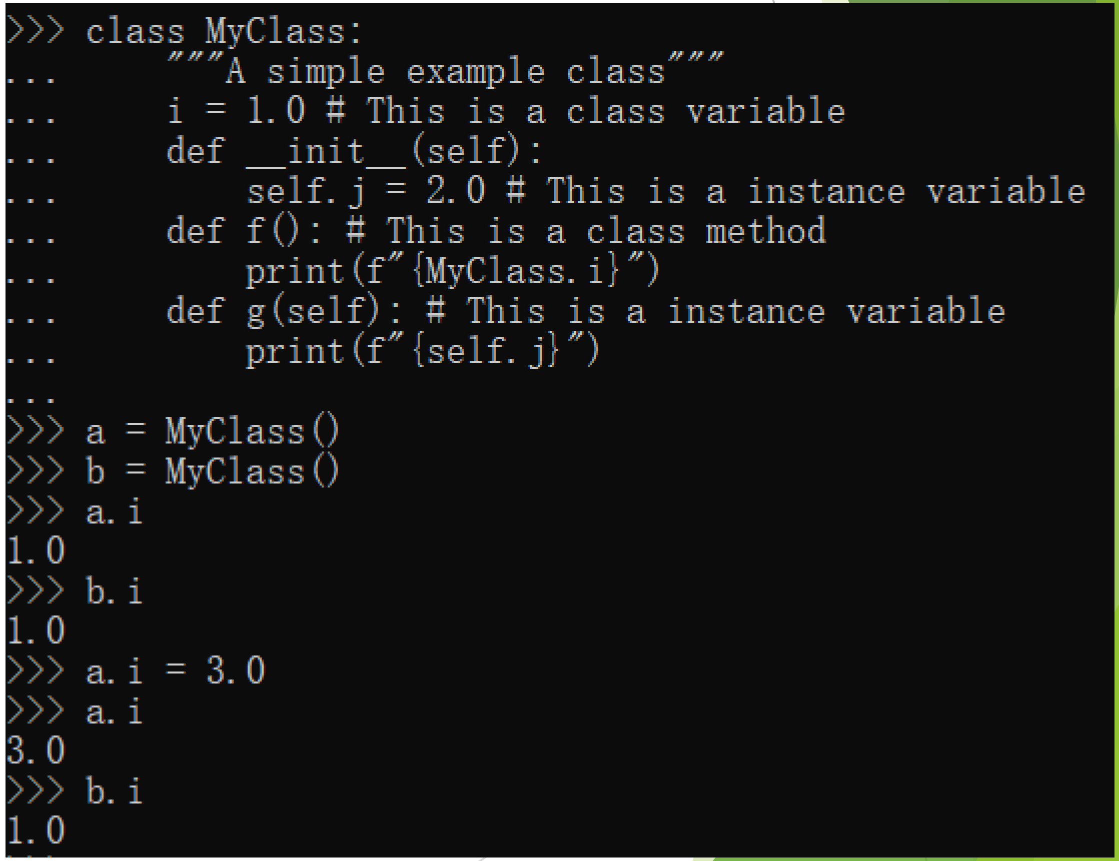


```
>>> class MyClass:
...     """A simple example class"""
...     i = 1.0 # This is a class variable
...     def __init__(self):
...         self.j = 2.0 # This is a instance variable
...     def f(): # This is a class method
...         print(f"{MyClass.i}")
...     def g(self): # This is a instance variable
...         print(f"{self.j}")
...
>>> MyClass.i
1.0
>>> MyClass.f
<function MyClass.f at 0x000001F4A17CED38>
>>> MyClass.f()
1.0
>>> c = MyClass()
>>> c.g()
2.0
```

类
Classes

- 由于Python中类实际上在内部由对象表示，因此，有人也称Python中基于类的继承实际上是基于对象的继承。理解这些对于Python中某些行为的理解比较有帮助。如一个实例修改了类变量，这种修改只对自己可见，对其他实例并不可见。而在C++中，对类变量的修改对所有实例都是可见的。

  Due to the fact that classes are implemented as internal objects in Python, now and then people say the inheritance based on class is essentially inheritance based on objects. Understanding of this helps to anticipate some behaviors in Python. For example, if an instance modified the class variable, such a modification is only effective for the current instance, without affecting any other instances.



```
>>> class MyClass:
...     """A simple example class"""
...     i = 1.0 # This is a class variable
...     def __init__(self):
...         self.j = 2.0 # This is a instance variable
...     def f(): # This is a class method
...         print(f"{MyClass.i}")
...     def g(self): # This is a instance variable
...         print(f"{self.j}")
...
>>> a = MyClass()
>>> b = MyClass()
>>> a.i
1.0
>>> b.i
1.0
>>> a.i = 3.0
>>> a.i
3.0
>>> b.i
1.0
```

特殊变量
Special Variables

- 大家经常在脚本中见到如下语句：

  We usually see the statement in scripts below:

  if __name__ == "__main__":

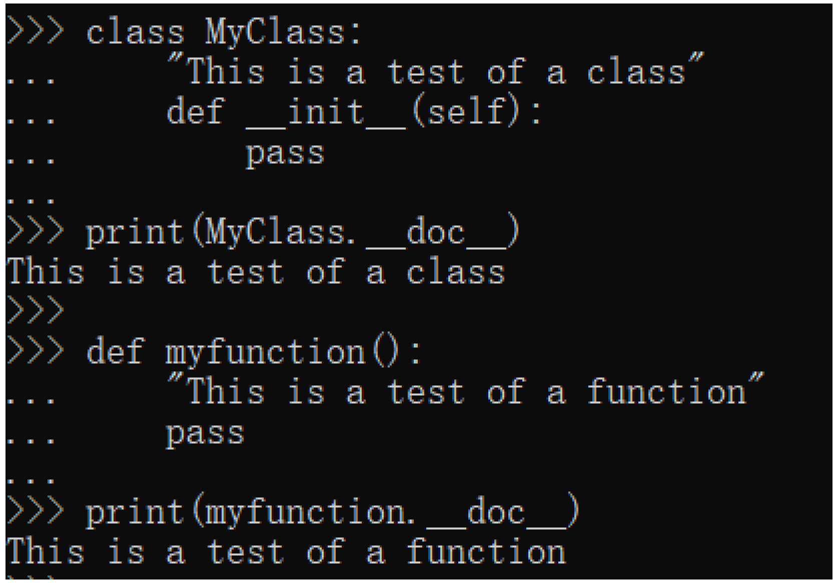
- 特殊变量__name__定义了运行的模块所在的名称空间。当Python实际运行代码时，会用实际的名称作为变量__name__的当前的值。如果在命令行运行下面程序：python example.py，则example.py为入口程序，则由于example本身亦可看作一个模块，则该模块对应的命名空间即为__main__。如果在example中再导入其它模块，则其他模块中的函数所在的名称空间便是所导入的模块的名称。

  The special variable __name__ defines the namespace that a Python module is running in. When Python is run, it will replace __name__ with it's namespace. If one runs python example.py in the terminal, since example.py is both the entry script and a module, then the correspondence namespace is __main__. If other modules are imported into example.py, then functions in other modules are running in the namespace of other modules.

特殊变量
Special Variables

- 特殊变量__doc__指代当前类或函数的说明字符串。说明字符串为类头或函数头的第一行开始的字符串型注释。

  __doc__ will print out the docstring that appears in a class or method. A docstring is a string comment and is the first line after the class or method header.

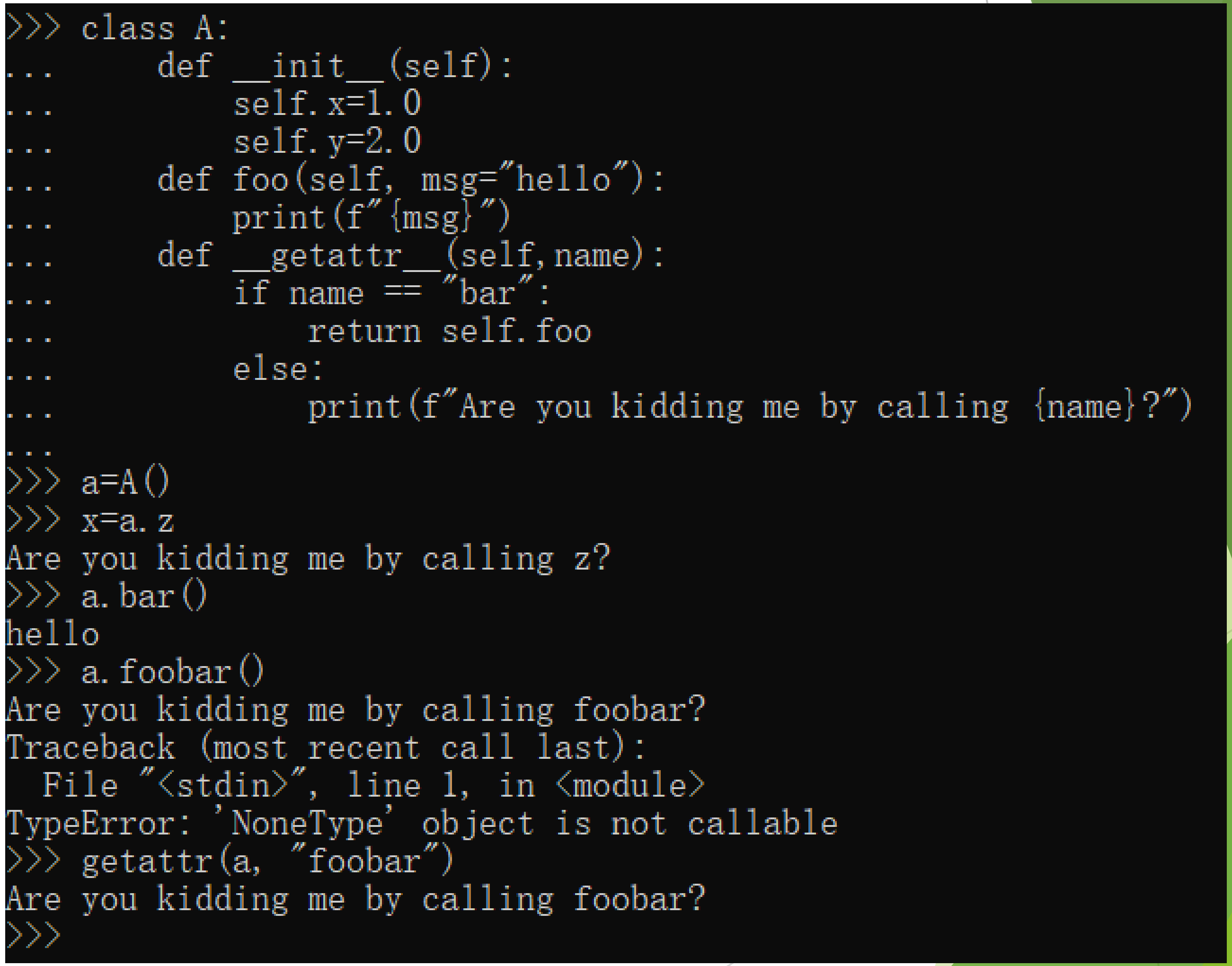


```
>>> class MyClass:
...     "This is a test of a class"
...     def __init__(self):
...         pass
...
>>> print(MyClass.__doc__)
This is a test of a class
>>>
>>> def myfunction():
...     "This is a test of a function"
...     pass
...
>>> print(myfunction.__doc__)
This is a test of a function
```

特殊变量
Special Variables

- 特殊变量__getattr__与__init__类似是类的成员方法，它指导Python在找不到调用的变量或方法时如何反应。__getattr__的参数为索引的变量或方法的名称。

  __getattr__ is a method like __init__ that you can write into your classes that specifies how Python reacts when it can't find a called variable or method. __getattr__ takes the name of what it's looking for as a parameter.

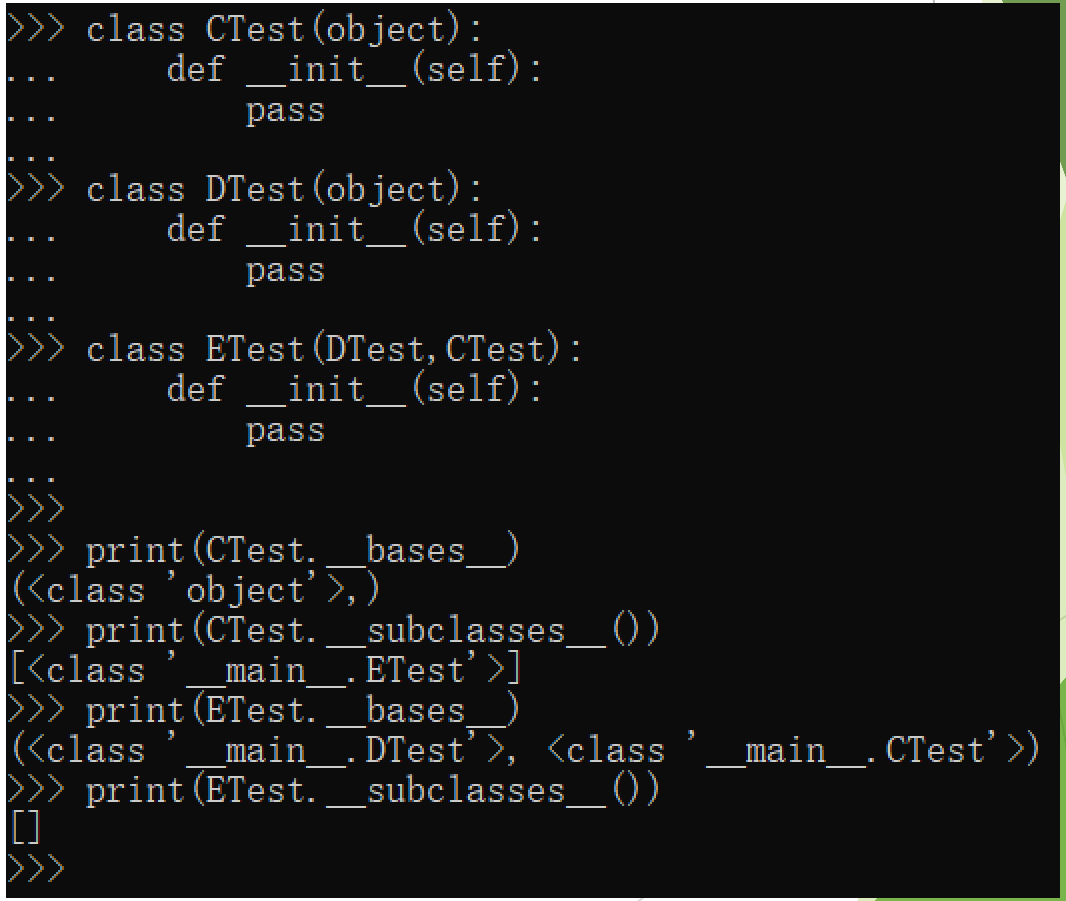


```
>>> class A:
...     def __init__(self):
...         self.x=1.0
...         self.y=2.0
...     def foo(self, msg="hello"):
...         print(f"{msg}")
...     def __getattr__(self,name):
...         if name == "bar":
...             return self.foo
...         else:
...             print(f"Are you kidding me by calling {name}?")
...
>>> a=A()
>>> x=a.z
Are you kidding me by calling z?
>>> a.bar()
hello
>>> a.foobar()
Are you kidding me by calling foobar?
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'NoneType' object is not callable
>>> getattr(a, "foobar")
Are you kidding me by calling foobar?
>>>
```

类
Classes

- 特殊变量__class__返回实例所代表的类名称，特殊变量__bases包含一个类继承自的所有类作为元组，特殊__subclasses__方法，而不是变量，返回列表中类的所有子类。

  __class__ returns the class an instance is, __bases__ is a variable that contains as a tuple all the classes that a class inherits from, and __subclasses__, instead of a variable, this method returns all the subclasses of a class in a list.

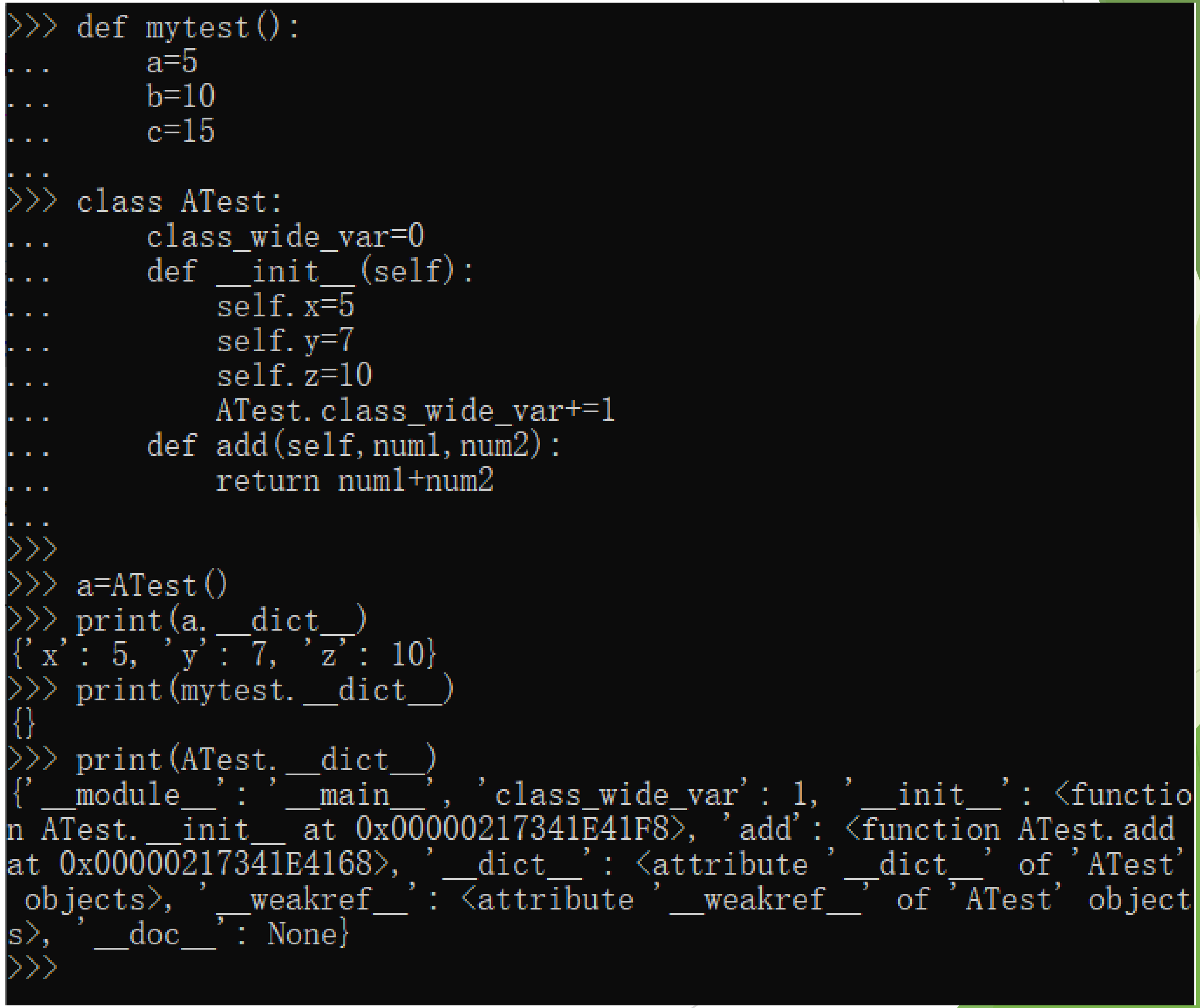


```
>>> class CTest(object):
...     def __init__(self):
...         pass
...
>>> class DTest(object):
...     def __init__(self):
...         pass
...
>>> class ETest(DTest,CTest):
...     def __init__(self):
...         pass
...
>>>
>>> print(CTest.__bases__)
(<class 'object'>,)
>>> print(CTest.__subclasses__())
[<class '__main__.ETest'>]
>>> print(ETest.__bases__)
(<class '__main__.DTest'>, <class '__main__.CTest'>)
>>> print(ETest.__subclasses__())
[]
>>>
```

类
Classes

- 特殊变量__dict__以字典的形式返回类实例的所有属性。

  __dict__ will return, as a dictionary, all attributes of a class instance.



```
>>> def mytest():
...     a=5
...     b=10
...     c=15
...
>>> class ATest:
...     class_wide_var=0
...     def __init__(self):
...         self.x=5
...         self.y=7
...         self.z=10
...         ATest.class_wide_var+=1
...     def add(self,num1,num2):
...         return num1+num2
...
>>>
>>> a=ATest()
>>> print(a.__dict__)
{'x': 5, 'y': 7, 'z': 10}
>>> print(mytest.__dict__)
{}
>>> print(ATest.__dict__)
{'__module__': '__main__', 'class_wide_var': 1, '__init__': <functio
n ATest.__init__ at 0x00000217341E41F8>, 'add': <function ATest.add
at 0x00000217341E4168>, '__dict__': <attribute '__dict__' of 'ATest'
 objects>, '__weakref__': <attribute '__weakref__' of 'ATest' object
s>, '__doc__': None}
>>>
```