Chapter (Classes) - 9 (Session)

01 May 2018 23:08

Creating and Using a Class

Creating the Dog Class

The parentheses in the class definition are empty because we're creating this class from scratch.

The __init__()

The __init__() method is a special method Python runs automatically whenever we create a new instance based on the Dog class.

This method has two leading underscores and two trailing underscores, a convention that helps prevent Python's default method names from conflicting with your method names.

The self parameter is required in the method definition, **and it must come first before the other parameters**.

It must be included in the definition because when Python calls this __init__() method later (to create an instance of Dog), **the method call will automatically pass the self argument**. **Every method call associated with a class automatically passes self, which is a reference to the instance itself**; it gives the individual instance access to the attributes and methods in the class.

When we make an instance of Dog, Python will call the __init__() method from the Dog class. We'll pass Dog() a name and an age as arguments; **self is passed automatically, so we don't need to pass it**.

Any variable prefixed with self is available to every method in the class, and we'll also be able to access these variables through any instance created from the class. self.name = name takes the value stored in the parameter name and stores it in 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 like this are called **attributes**.

```
Class Animal():
    def __init__(self,color):
        self.color=color
        print('I am Animal from__init__,mycolor=',self.color)

    def makeNoise(self):
        print('catmakenoise=',self.color)

animal=Animal('red')
animal.makeNoise()
```

Making an Instance from a Class

Accessing attributes

Calling Methods

□ Creating multiple instances

TRY IT YOURSELF

9-1. Restaurant: Make a class called `Restaurant`. The `__init__()` method for `Restaurant` should store two attributes: a `restaurant_name` and a `cuisine_type`. Make a method called `describe_restaurant()` that prints these two pieces of information, and a method called `open_restaurant()` that prints a message indicating that the restaurant is open.

Make an instance called `restaurant` from your class. Print the two attributes individually, and then call both methods.

9-2. Three Restaurants: Start with your class from Exercise 9-1. Create three different instances from the class, and call `describe_restaurant()` for each instance.

9-3. Users: Make a class called `User`. Create two attributes called `first_name` and `last_name`, and then create several other attributes that are typically stored in a user profile. Make a method called `describe_user()` that prints a summary of the user's information. Make another method called `greet_user()` that prints a personalized greeting to the user.

Create several instances representing different users, and call both methods for each user.

□ Setting a Default Value for an Attribute

```
class Animal():
    def __init__(self, color):
        self.color = color
        self.age = 10
        print('I am Animal form __init__, my color = ', self.color)

    def makeNoise(self):
        print('cat make noise=', self.color)
        print('self.age=', self.age)
        # print('color=', color)
        # print('age=', age)


animal = Animal('red')
animal.makeNoise()
print('animal.color = ', animal.color)
print('animal.age = ', animal.age)
```

□ Modifying Attribute Values

Modifying an Attribute's Value Directly
Modifying an Attribute's Value Through a Method

□ Inheritance

You don't always have to start from scratch when writing a class. If the class you're writing is a specialized version of another class you wrote, you can use inheritance.

When one class inherits from another, it automatically takes on all 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 inherits every attribute and method from its parent class but is also free to define new attributes and methods of its own.

The super() function is a special function that helps Python make connections between the parent and child class. This line tells Python to call the __init__() method from ElectricCar's parent class, which gives an ElectricCar instance all the attributes of its parent class. The name super comes from a convention of calling the parent class a superclass and the child class a subclass.

```
class Animal():
    def __init__(self,color):
        self.color = color
        print('i am Animal init method')

    def sound(self):
        print('i am Animal')

class Dog(Animal):
    def __init__(self,age, breed, color):
        super().__init__(color)
        print('i am Dog init method')
        print('self=',self)
        self.age = age
        self.breed = breed

    def bark(self):
        print('i can bark')
        print('self.breed=',self.breed)
        print('self.age=',self.age)
        print('self.color=',self.color)

d = Dog(5,'dogs','red')
d.bark()
print('d.color=',d.color)
d.sound()
```

□ Defining Attributes and Methods for the Child Class

```
class Animal():
    def __init__(self,color):
        self.color = color
        print('i am Animal init method')

    def sound(self):
        print('i am Animal')
```

```
class Dog(Animal):
    def __init__(self,age, breed, color):
        super().__init__(color)
        print('i am Dog init method')
        print('self=',self)
        self.age = age
        self.breed = breed

    def bark(self):
        print('i can bark')
        print('self.breed=',self.breed)
        print('self.age=',self.age)
        print('self.color=',self.color)

d = Dog(5,'dogs','red')
d.bark()
print('d.color=',d.color)
d.sound()
```

□ Overriding Methods from the Parent Class

```
class Animal():
    def __init__(self,color):
        self.color = color

    def sound(self):
        print('Animal sound...')

class Dog(Animal):
    def __init__(self,age, breed, color):
        super().__init__(color)

    def sound(self):
        print('Dog sound...')

d = Dog(5,'dogs','red')
d.sound()
```

□ Instances as Attributes

□ Importing Classes

□ Importing a Single Class

```
from car import Car
```

□ Storing Multiple Classes in a Module

□ Importing Multiple Classes from a Module

```
from car import Car, ElectricCar
```

□ Importing an Entire Module

```
import car
my_beetle = car.Car('volkswagen', 'beetle', 2016)
```

- **Importing All Classes from a Module**

```
from module_name import *
```

The Python Standard Library

The Python standard library is a set of modules included with every Python installation.

Let's look at one class, **OrderedDict**, from the module **collections**.

Instances of the OrderedDict class behave almost exactly like dictionaries except they keep track of the order in which key-value pairs are added

Notice there are no curly brackets; the call to OrderedDict() creates an empty ordered dictionary
for us and stores it in favorite_languages. We then add each name and language to favorite_languages one at a time w. Now when we loop through favorite_languages at x, we know we'll always get responses back in the order they were added: