Python Programming
Lecture 10 Object-Oriented Programming

10.1 OOP Basics

- Object-oriented programming (面向对象编程) is one of the most effective approaches to writing software.
- Python has been an object-oriented language (面向对象语言) since it existed. Because of this, creating and using classes and objects are downright easy.
- Understanding object-oriented programming will help you see the world as a programmer does. It'll help you really know your code, not just what's happening line by line, but also the bigger concepts behind it.

- Creating and Using a Class (类)

```
class Dog():
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def sit(self):
        print(f"{self.name} is now sitting.")

    def roll_over(self):
        print(f"{self.name} rolled over!")
```

- By convention, capitalized names（首字母大写）refer to classes in Python.
- Methods (方法) are functions that belong to a class.
- Attributes (属性) are variables that belong to an object or class.
- The __init__() method is a special method Python runs automatically whenever we create a new instance (实例) based on the class.

class

Dog

instance

instance

instance

- Making an Instance from a Class

```python
class Dog():
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def sit(self):
        print(f"{self.name} is now sitting.")

    def roll_over(self):
        print(f"{self.name} rolled over!")
```

```python
my_dog = Dog('Willie', 6)

print(my_dog.name)
print(my_dog.age)
```

```
Willie
6
```

```python
my_dog.sit()
my_dog.roll_over()
```

```
Willie is now sitting.
Willie rolled over!
```

```python
your_dog = Dog('Lucy', 3)
```

- Working with Classes and Instances

```python
class 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 read_odometer(self):
        print(f"This car has\
            {self.odometer_reading}\
             miles on it.")
```

```python
my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

```
2016 Audi A4
This car has 0 miles on it.
```

- Modifying Attribute Values

```
1  class Car():
2      def __init__(self, make, model, year):
3          self.make = make
4          self.model = model
5          self.year = year
6          self.odometer_reading = 0
7
8      def read_odometer(self):
9          print(f"This car has\
10             {self.odometer_reading}\
11              miles on it.")
```

- Modifying an Attribute's Value Directly

```
1  my_new_car.odometer_reading = 23
2  my_new_car.read_odometer()
```

```
2016 Audi A4
This car has 23 miles on it.
```

- Modifying Attribute Values

```
1  class Car():
2      def __init__(self, make, model, year):
3          self.make = make
4          self.model = model
5          self.year = year
6          self.odometer_reading = 0
7
8      def read_odometer(self):
9          print(f"This car has\
10             {self.odometer_reading}\
11              miles on it.")
12
13     def update_odometer(self, mileage):
14         self.odometer_reading = mileage
```

- Modifying an Attribute's Value Through a Method

```
1  my_new_car = Car('audi', 'a4', 2016)
2  my_new_car.update_odometer(23)
3  my_new_car.read_odometer()
```

```
This car has 23 miles on it.
```

```
1  my_new_car.update_odometer(30)
2  my_new_car.read_odometer()
```

```
This car has 30 miles on it.
```

```
class Car():
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def read_odometer(self):
        print(f"This car has\
            {self.odometer_reading}\
             miles on it.")

    def update_odometer(self, mileage):
        self.odometer_reading = mileage

    def increment_odometer(self, miles):
        self.odometer_reading += miles
```

- Modifying an Attribute's Value Through a Method

```
my_new_car = Car('audi', 'a4', 2016)
my_new_car.update_odometer(23)
my_new_car.read_odometer()
```

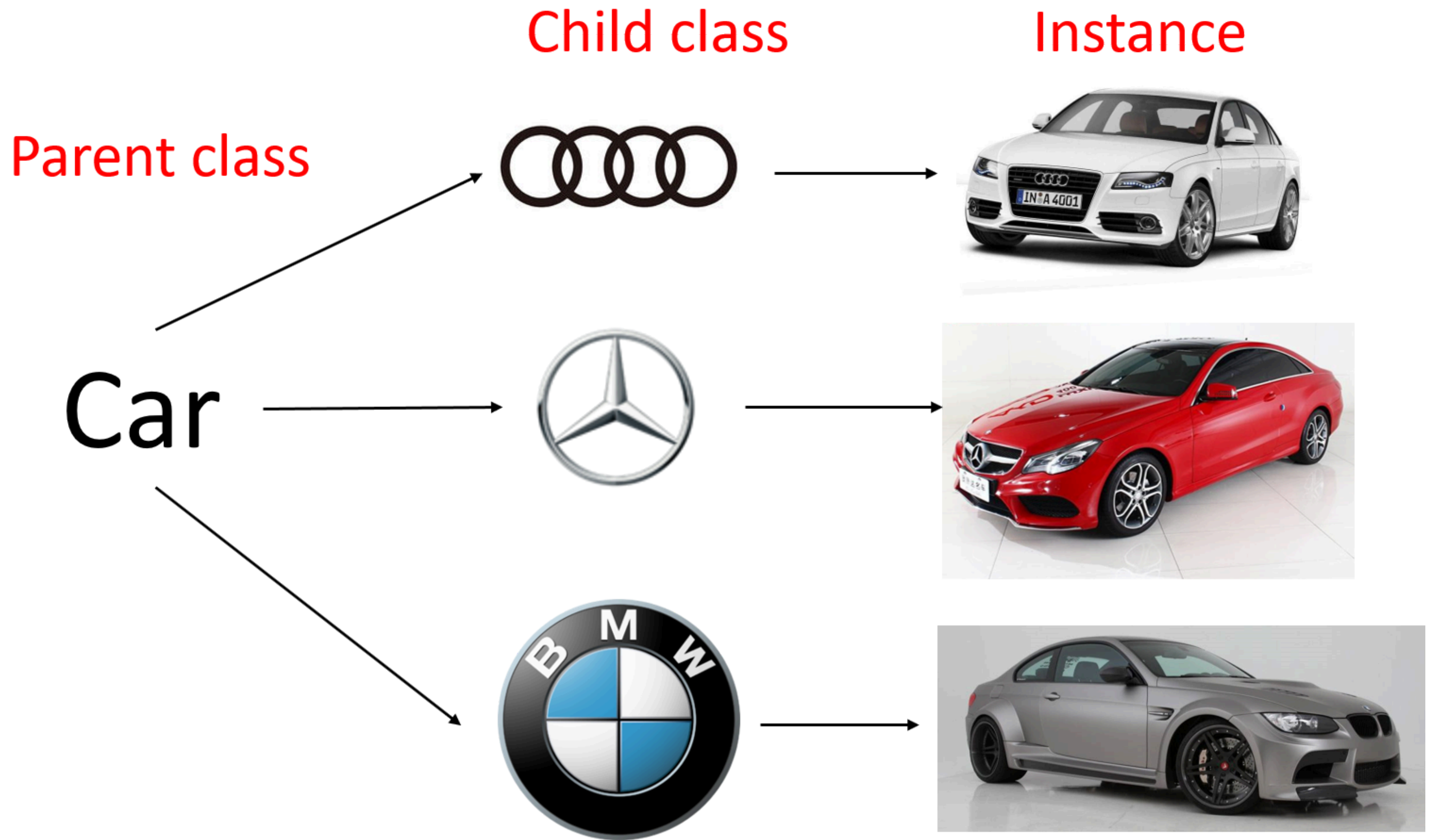
```
This car has 23 miles on it.
```

```
my_new_car.increment_odometer(100)
my_new_car.read_odometer()
```

```
This car has 123 miles on it.
```

10.2 Inheritance, Polymorphism

- 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 (继承).
- 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 (父类), and the new class is the child class (子类). 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.


Child class
Instance
Parent class
Car
M
B
W

Inheritance (继承)

```python
class 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()
```

```python
class ElectricCar(Car):
    def __init__(self, make, model, year):
        super().__init__(make, model, year)
```

```python
my_tesla = ElectricCar('tesla', 'model s', 2016)
print(my_tesla.get_descriptive_name())
```

```
2016 Tesla Model S
```

Polymorphism (多态)

- Defining attributes and methods for the child class

```
1  class Car():
2      def __init__(self, make, model, year):
3          self.make = make
4          self.model = model
5          self.year = year
6          self.odometer_reading = 0
7
8      def get_descriptive_name(self):
9          long_name=f"{self.year} {self.make}\
10                  {self.model}"
11         return long_name.title()
```

```
1  class ElectricCar(Car):
2      def __init__(self, make, model, year):
3          super().__init__(make, model, year)
4          self.battery_size = 70
5
6      def describe_battery(self):
7          print(f"{self.battery_size}-kWh battery.")
```

```
1  my_tesla = ElectricCar('tesla', 'model s', 2016)
2  print(my_tesla.get_descriptive_name())
3  my_tesla.describe_battery()
```

```
2016 Tesla Model S
70-kWh battery.
```

- Overriding (覆盖) methods from the parent class

```
1 class Car():
2     def __init__(self, make, model, year):
3         self.make = make
4         self.model = model
5         self.year = year
6         self.odometer_reading = 0
7
8     def get_descriptive_name(self):
9         long_name=f"{self.year} {self.make}\
10                 {self.model}"
11         return long_name.title()
```

```
1 class ElectricCar(Car):
2     def __init__(self, make, model, year):
3         super().__init__(make, model, year)
4         self.battery_size = 70
5
6     def describe_battery(self):
7         print(f"{self.battery_size}-kWh battery.")
8
9     def get_descriptive_name(self):
10        long_name=f"{self.year} {self.make}"
11        return long_name.title()
```

- Open/closed principle (开放封闭原则)
- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

- We learned to use super() for inheritance. There are two other ways to do it.

```python
class Person:
    def __init__(self,name,sex):
        self.name = name
        self.sex = sex

    def print_title(self):
        if self.sex == "male":
            print("man")
        elif self.sex == "female":
            print("woman")

class Child(Person):
    pass
```

```python
May = Child("May","female")
Peter = Person("Peter","male")

print(May.name,May.sex,Peter.name,Peter.sex)
May.print_title()
Peter.print_title()
```

```
May female Peter male
woman
man
```

```python
class Person:
    def __init__(self,name,sex):
        self.name = name
        self.sex = sex

    def print_title(self):
        if self.sex == "male":
            print("man")
        elif self.sex == "female":
            print("woman")

class Child(Person):
    def __init__(self,name,sex):
        Person.__init__(self,name,sex)
```

```python
May = Child("May","female")
Peter = Person("Peter","male")

print(May.name,May.sex,Peter.name,Peter.sex)
May.print_title()
Peter.print_title()
```

```
May female Peter male
woman
man
```

- Overwrite completely

```
class Person:
    def __init__(self,name,sex):
        self.name = name
        self.sex = sex

class Child(Person):
    def __init__(self,name,sex,mother):
        self.name = name
        self.sex = sex
        self.mother = mother

May = Child("May","female","April")
print(May.name,May.sex,May.mother)
```

```
May female April
```

- Overwrite partially

```
class Person:
    def __init__(self,name,sex):
        self.name = name
        self.sex = sex

class Child(Person):
    def __init__(self,name,sex,mother):
        Person.__init__(self,name,sex)
        #super().__init__(name,sex)
        self.mother = mother

May = Child("May","female","April")
print(May.name,May.sex,May.mother)
```

```
May female April
```

- If you just need complete inheritance of attributes, then all three ways are equivalent.

```python
1 class Person:
2     pass
3 class Child(Person):
4     pass
5 May = Child()
6 Peter = Person()
```

```python
1 print(isinstance(May,Child))          # True
2 print(isinstance(May,Person))         # True
3 print(isinstance(Peter,Child))        # False
4 print(isinstance(Peter,Person))       # True
5 print(issubclass(Child,Person))       # True
```

10.3 Special Methods, Import Class

- The precise definition of dir() is that it lists the methods and attributes of a Python object.

```
1  >>> stuff = list()
2  >>> dir(stuff)
3  ['__add__', '__class__', '__contains__', '__delattr__',
4  '__delitem__', '__dir__', '__doc__', '__eq__',
5  '__format__', '__ge__', '__getattribute__', '__getitem__',
6  '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
7  '__iter__', '__le__', '__len__', '__lt__', '__mul__',
8  '__ne__', '__new__', '__reduce__', '__reduce_ex__',
9  '__repr__', '__reversed__', '__rmul__', '__setattr__',
10 '__setitem__', '__sizeof__', '__str__', '__subclasshook__',
11 'append', 'clear', 'copy', 'count', 'extend', 'index',
12 'insert', 'pop', 'remove', 'reverse', 'sort']
```

- Everything in Python is an object. (Python中一切都是对象)

```
1 stuff = []
2 stuff.append('python')
3 stuff.append('chuck')
4 stuff.sort()
5
6 print(stuff[0])
7 print(stuff.__getitem__(0))
8 print(list.__getitem__(stuff,0))
```

```
chuck
chuck
chuck
```

- Actually, the class you create inherits the "object" class.
- Classes can intercept Python Operators

```python
class Dog():
    def __init__(self, name, age):
        self.name = name
        self.age = age


my_dog = Dog('willie', 6)
print(dir(my_dog))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__',
 '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
'age', 'name']
```

- __str__

```python
class Dog():
    def __init__(self, name, age):
        self.name = name
        self.age = age

my_dog = Dog('willie', 6)
print(my_dog)
```

```
<__main__.Dog object at 0x053A3750>
```

```python
class Dog():
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return self.name

my_dog = Dog('willie', 6)
print(my_dog)
```

```
willie
```

```
class Dog():
    "This is about dog class."
    def __init__(self, name, age):
        self.name = name
        self.age = age

print(Dog.__doc__)
print(Dog.__name__)
print(Dog.__module__)
print(Dog.__bases__)
```

```
This is about dog class.
Dog
__main__
(<class 'object'>,)
```

- __doc__, __name__, __module__, __bases__

```
class Dog():
    "This is about dog class."
    def __init__(self, name, age):
        self.name = name
        self.age = age
class Husky(Dog):
    pass

print(Husky.__bases__)
```

```
(<class '__main__.Dog'>,)
```

- Importing a Single Class
- Save your Car class in a python file: car.py

```
1 from car import Car
2
3 my_new_car = Car('audi', 'a4', 2016)
4 print(my_new_car.get_descriptive_name())
5
6 my_new_car.odometer_reading = 23
7 my_new_car.read_odometer()
```

```
2016 Audi A4
This car has 23 miles on it.
```

- Storing Multiple Classes in a Module
- You can store as many classes as you need in a single module, although each class in a module should be related somehow.

```
1 from car import ElectricCar
2
3 my_tesla = ElectricCar('tesla', 'model s', 2016)
4
5 print(my_tesla.get_descriptive_name())
```

```
2016 Tesla Model S
```

- Importing Multiple Classes from a Module

```
1 from car import Car, ElectricCar
2
3 my_beetle = Car('volkswagen', 'beetle', 2016)
4 print(my_beetle.get_descriptive_name())
5
6 my_tesla = ElectricCar('tesla', 'roadster', 2016)
7 print(my_tesla.get_descriptive_name())
```

```
2016 Volkswagen Beetle
2016 Tesla Roadster
```

- Importing a Module into a Module
- car.py

```python
1 class Car():
2 ...
```

- electric_car.py

```python
1 from car import Car
2
3 class Battery():
4 ...
5
6 class ElectricCar(Car):
7 ...
```

- my_cars.py

```python
1 from car import Car
2 from electric_car import ElectricCar
```

Summary of OOP

- Three Features of OOP

  Encapsulation（封装）, Inheritance（继承）, Polymorphism（多态）

- Everything in Python is object.

- Process-oriented programming (面向过程编程)

  Object-oriented Programming (面向对象编程)

Summary

- OOP
  - Reading: Python Crash Course, Chapter 9