



武汉大学  
WUHAN UNIVERSITY

# 数据结构与算法

Python 面向对象编程

---

张晓平

武汉大学数学与统计学院

# Table of contents

1. 类和实例
2. 访问限制
3. 继承与多态
4. 获取对象信息
5. 实例属性与类属性

面向对象编程 (Object Oriented Programming, OOP), 是一种程序设计思想。OOP 把对象作为程序的基本单元, 一个对象包含了数据和操作数据的函数。

面向对象编程 (Object Oriented Programming, OOP), 是一种程序设计思想。OOP 把对象作为程序的基本单元, 一个对象包含了数据和操作数据的函数。

面向过程编程 (Procedure Oriented Programming, POP) 把程序视为一系列**命令的集合**, 即一组函数的顺序执行。为了简化程序设计, POP 把函数继续切分为子函数, 即把大块函数通过切割成小块函数来降低系统的复杂度。

面向对象编程 (Object Oriented Programming, OOP), 是一种程序设计思想。OOP 把对象作为程序的基本单元, 一个对象包含了数据和操作数据的函数。

面向过程编程 (Procedure Oriented Programming, POP) 把程序视为一系列**命令的集合**, 即一组函数的顺序执行。为了简化程序设计, POP 把函数继续切分为子函数, 即把大块函数通过切割成小块函数来降低系统的复杂度。

而 OOP 把程序视为一组**对象的集合**, 而每个对象都可以接收其他对象发过来的消息, 并处理这些消息, 计算机程序的执行就是一系列消息在各个对象之间传递。

在 Python 中，**万物皆对象**。可以自定义数据类型，这就是 OOP 中类 (Class) 的概念。

以 " 处理学生成绩 " 举例，我们来说明 POP 与 OOP 在程序流程上的不同之处。

以 " 处理学生成绩 " 举例，我们来说明 POP 与 OOP 在程序流程上的不同之处。

为表示一个学生的成绩，POP 可用一个 dict 来表示：

```
std1 = {'name': 'Michael', 'score': 98}
std2 = {'name': 'Bob', 'score': 81}
```



以 " 处理学生成绩 " 举例，我们来说明 POP 与 OOP 在程序流程上的不同之处。

为表示一个学生的成绩，POP 可用一个 dict 来表示：

```
std1 = {'name': 'Michael', 'score': 98}
std2 = {'name': 'Bob', 'score': 81}
```

而处理学生成绩可通过函数来实现，比如打印学生的成绩：

```
def print_score(std):
    print(f"{std['name']}: {std['score']}")
print_score(std1)
print_score(std2)
```

```
Michael: 98
Bob: 81
```

若采用 OOP，首先考虑的不是程序的执行流程，而是去定义一个 Student 类型，其实例（即 Student 对象）拥有name和score这两个属性 (Property)。

若采用 OOP，首先考虑的不是程序的执行流程，而是去定义一个 Student 类型，其实例（即 Student 对象）拥有 name 和 score 这两个属性 (Property)。

若要打印一个学生的成绩，先创建出这个学生对应的对象，然后给对象发送一个 print\_score 消息，让对象自己把自己的数据打印出来。

若采用 OOP，首先考虑的不是程序的执行流程，而是去定义一个 Student 类型，其实例（即 Student 对象）拥有 name 和 score 这两个属性 (Property)。

若要打印一个学生的成绩，先创建出这个学生对应的对象，然后给对象发送一个 print\_score 消息，让对象自己把自己的数据打印出来。

```
class Student(object):  
    def __init__(self, name, score):  
        self.name = name  
        self.score = score  
    def print_score(self):  
        print(f'{self.name}: {self.score}')
```

若采用 OOP，首先考虑的不是程序的执行流程，而是去定义一个 Student 类型，其实例（即 Student 对象）拥有 name 和 score 这两个属性 (Property)。

若要打印一个学生的成绩，先创建出这个学生对应的对象，然后给对象发送一个 print\_score 消息，让对象自己把自己的数据打印出来。

```
class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score
    def print_score(self):
        print(f'{self.name}: {self.score}')
```

给对象发送消息实际上就是调用对象对应的关联函数，称之为对象的方法 (Method)。

## 类的使用

```
bart = Student('Bart Simpson', 59)
lisa = Student('Lisa Simpson', 87)
bart.print_score()
lisa.print_score()
```

```
Bart Simpson: 59
Lisa Simpson: 87
```

OOP 的设计思想来源于自然界，因为在自然界中，类 (class) 和实例 (instance) 的概念非常自然。

OOP 的设计思想来源于自然界，因为在自然界中，类 (class) 和实例 (instance) 的概念非常自然。

- class 是一种抽象概念，比如我们定义的Student类，是指学生这个概念；
- 而实例 (instance) 则是一个个具体的Student，比如，bart和lisa是两个具体的Student。



OO 的设计思想来源于自然界，因为在自然界中，类 (class) 和实例 (instance) 的概念非常自然。

- class 是一种抽象概念，比如我们定义的Student类，是指学生这个概念；
- 而实例 (instance) 则是一个个具体的Student，比如，bart和lisa是两个具体的Student。

因此，OO 的设计思想是抽象出 class，根据 class 创建 instance。

OO 的设计思想来源于自然界，因为在自然界中，类 (class) 和实例 (instance) 的概念非常自然。

- class 是一种抽象概念，比如我们定义的Student类，是指学生这个概念；
- 而实例 (instance) 则是一个个具体的Student，比如，bart和lisa是两个具体的Student。

因此，OO 的设计思想是抽象出 class，根据 class 创建 instance。

OO 的抽象程度高于函数，因为一个 Class 既包含数据，又包含操作数据的方法。

# 类和实例

OOP 最重要的概念就是类 (Class) 和实例 (Instance).

OOO 最重要的概念就是类 (Class) 和实例 (Instance).

## 注

- 类是抽象的模板，比如 Student 类，
- 实例是根据类创建出来的一个个具体的“对象”，每个对象都拥有相同的方法，但各自的数据可能不同。

仍以 Student 类为例，在 Python 中，使用关键字 `class` 定义类：

```
class Student(object):  
    pass
```

仍以 Student 类为例，在 Python 中，使用关键字 `class` 定义类：

```
class Student(object):  
    pass
```

`class` 后紧跟类名，即 Student，再紧跟 `(object)`，表示该类是从哪个类继承下来的。

仍以 Student 类为例，在 Python 中，使用关键字 `class` 定义类：

```
class Student(object):  
    pass
```

`class` 后紧跟类名，即 Student，再紧跟 `(object)`，表示该类是从哪个类继承下来的。

## 注

- 类名通常以大写字母开头
- 如果没有合适的继承类，就使用 `object` 类，它是所有类最终都会继承的类。



## 类和实例

定义好了 Student 类，就可以根据它来创建出其实例。

# 类和实例

定义好了 Student 类，就可以根据它来创建出其实例。

## 例：通过类名+()来创建实例

```
bart = Student()
lisa = Student()
print(Student)
print(bart)
print(lisa)
```

```
<class '__main__.Student'>
<__main__.Student object at 0x7fb24f350630>
<__main__.Student object at 0x7fb24f3505f8>
```

# 类和实例

定义好了 Student 类，就可以根据它来创建出其实例。

## 例：通过类名+()来创建实例

```
bart = Student()
lisa = Student()
print(Student)
print(bart)
print(lisa)
```

```
<class '__main__.Student'>
<__main__.Student object at 0x7fb24f350630>
<__main__.Student object at 0x7fb24f3505f8>
```

Student 本身是一个类，而bart指向的是一个 Student 的实例，0x7fb24f350630是其内存地址，每个 object 的地址都不一样。

可以自由地给一个实例变量绑定属性.

可以自由地给一个实例变量绑定属性.

**例: 给实例bart绑定一个name属性**

```
bart.name = 'Bart Simpson'  
print(bart.name)
```

```
Bart Simpson
```

## 类和实例

类起到模板的作用，可在创建实例时把我们认为一些必须绑定的属性强制填写进去。

## 类和实例

类起到模板的作用，可在创建实例时把我们认为一些必须绑定的属性强制填写进去。

可通过定义一个名为`__init__`的特殊方法，在创建实例时把`name`，`score`等属性绑上去：

```
class Student(object):  
    def __init__(self, name, score):  
        self.name = name  
        self.score = score
```

## 类和实例

类起到模板的作用，可在创建实例时把我们认为一些必须绑定的属性强制填写进去。

可通过定义一个名为`__init__`的特殊方法，在创建实例时把`name`，`score`等属性绑上去：

```
class Student(object):  
    def __init__(self, name, score):  
        self.name = name  
        self.score = score
```

### 注

- 特殊方法`__init__`前后分别有两根下划线。
- `__init__()`的第一个参数永远是`self`，表示创建的实例本身。因此，在`__init__()`内部，就可把各种属性绑定到`self`，因为`self`就指向创建的实例本身。



有了`__init__()`，在创建实例时，就不能传入空的参数了，必须传入与`__init__()`相匹配的参数，但`self`不需要传，Python 解释器自己会把实例变量传进去。

有了`__init__()`，在创建实例时，就不能传入空的参数了，必须传入与`__init__()`相匹配的参数，但`self`不需要传，Python 解释器自己会把实例变量传进去。

```
bart = Student('Bart Simpson', 59)
print(bart.name)
print(bart.score)
```

## 类和实例

有了`__init__()`，在创建实例时，就不能传入空的参数了，必须传入与`__init__()`相匹配的参数，但`self`不需要传，Python 解释器自己会把实例变量传进去。

```
bart = Student('Bart Simpson', 59)
print(bart.name)
print(bart.score)
```

```
Bart Simpson
59
```

### 注

和普通函数相比，在类中定义的函数（即方法）只有一点不同，就是第一个参数永远是实例变量`self`，并且调用时不用传递该参数。除此之外，类的方法和普通函数没有什么区别，你仍然可以用默认参数、可变参数、关键字参数和命名关键字参数。

# 类和实例

## 数据封装

OOP 的一个重要特点就是**数据封装**。

OOP 的一个重要特点就是**数据封装**。

对于 Student 类，每个实例拥有各自的数据，如name和score.

OOP 的一个重要特点就是**数据封装**。

对于 Student 类，每个实例拥有各自的数据，如name和score.

**例：可以通过函数来访问这些数据，如打印一个学生的成绩**

```
def print_score(std):  
    print(f"{std.name}: {std.score}")  
print_score(bart)
```

```
Bart Simpson: 59
```



既然 Student 的实例本身就拥有这些数据，要访问它们，就没有必要通过外面的函数去访问，可以直接在 Student 类的内部定义访问数据的函数，这样就把“数据”给封装了。

既然 Student 的实例本身就拥有这些数据，要访问它们，就没有必要通过外面的函数去访问，可以直接在 Student 类的内部定义访问数据的函数，这样就把“数据”给封装了。

封装数据的函数是和 Student 类本身是关联起来的，我们称之为**类的方法**。

## 例：通过类的方法打印学生的成绩

```
class Student(object):  
    def __init__(self, name, score):  
        self.name = name  
        self.score = score  
    def print_score(self):  
        print(f"{self.name}: {self.score}")  
bart = Student('Bart Simpson', 59)  
bart.print_score()
```

## 例：通过类的方法打印学生的成绩

```
class Student(object):  
    def __init__(self, name, score):  
        self.name = name  
        self.score = score  
    def print_score(self):  
        print(f"{self.name}: {self.score}")  
bart = Student('Bart Simpson', 59)  
bart.print_score()
```

```
Bart Simpson: 59
```

## 注

- 从调用方来看 Student 类，只需在创建实例时给定name和score；
- 至于如何打印，会在 Student 类的内部实现。这些数据和逻辑被“封装”起来，调用会变得容易，不用知道内部实现的细节。

封装的另一个好处是可以给 Student 类增加新的方法.

# 数据封装

封装的另一个好处是可以给 Student 类增加新的方法.

## 例: 增加get\_grade方法

```
class Student(object):
    def get_grade(self):
        if self.score >= 90:
            return 'A'
        elif self.score >= 60:
            return 'B'
        else:
            return 'C'

bart = Student('Bart Simpson', 59)
print(f'{bart.name}: {bart.get_grade()}')
```

封装的另一个好处是可以给 Student 类增加新的方法.

## 例: 增加get\_grade方法

```
class Student(object):
    def get_grade(self):
        if self.score >= 90:
            return 'A'
        elif self.score >= 60:
            return 'B'
        else:
            return 'C'

bart = Student('Bart Simpson', 59)
print(f'{bart.name}: {bart.get_grade()}')
```

```
Bart Simpson: C
```



## 访问限制

## 访问限制

对象有属性和方法，而外部代码可以通过直接调用对象的方法来操作数据，这样就隐藏了内部的复杂逻辑。

但是，从Student类的定义来看，外部代码还是可以自由地修改一个实例的属性：

## 访问限制

对象有属性和方法，而外部代码可以通过直接调用对象的方法来操作数据，这样就隐藏了内部的复杂逻辑。

但是，从Student类的定义来看，外部代码还是可以自由地修改一个实例的属性：

### 例：直接修改name, score属性

```
bart = Student('Bart Simpson', 59)
print(bart.score)
bart.score = 99
print(bart.score)
```

## 访问限制

对象有属性和方法，而外部代码可以通过直接调用对象的方法来操作数据，这样就隐藏了内部的复杂逻辑。

但是，从Student类的定义来看，外部代码还是可以自由地修改一个实例的属性：

### 例：直接修改name, score属性

```
bart = Student('Bart Simpson', 59)
print(bart.score)
bart.score = 99
print(bart.score)
```

```
59
```

```
99
```

如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线\_\_。

### 注

在 Python 中，实例的变量名如果以\_\_开头，就变成了一个私有变量 (private)，只有内部可以访问，外部不能访问。

## 访问限制

```
class Student(object):
    def __init__(self, name, score):
        self.__name = name
        self.__score = score
    def print_score(self):
        print(f"{self.__name}: {self.__score}")
bart = Student('Bart Simpson', 59)
print(bart.__name)
```

## 访问限制

```
class Student(object):
    def __init__(self, name, score):
        self.__name = name
        self.__score = score
    def print_score(self):
        print(f"{self.__name}: {self.__score}")
bart = Student('Bart Simpson', 59)
print(bart.__name)
```

```
Traceback (most recent call last):
  File "src/slide02/code/oop3.py", line 19, in <module>
    print(bart.__name)
AttributeError: 'Student' object has no attribute '__name'
```

## 访问限制

```
class Student(object):
    def __init__(self, name, score):
        self.__name = name
        self.__score = score
    def print_score(self):
        print(f"{self.__name}: {self.__score}")
bart = Student('Bart Simpson', 59)
print(bart.__name)
```

```
Traceback (most recent call last):
  File "src/slide02/code/oop3.py", line 19, in <module>
    print(bart.__name)
AttributeError: 'Student' object has no attribute '__name'
```



此时，外部代码已经无法访问`bart.__name`和`bart.__score`。通过访问限制的保护，就确保了外部代码不能随意修改对象内部的状态，使得代码更加健壮。

问

如何外部代码想获取name和score怎么办？

问

如何外部代码想获取name和score怎么办？

可以给Student类增加get\_name和get\_score这样的方法：

```
class Student(object):  
    def get_name(self):  
        return self.__name  
    def get_score(self):  
        return self.__score
```

问

如果外部代码想修改score怎么办？

问

如果外部代码想修改score怎么办？

可以再给Student类增加set\_score方法：

```
class Student(object):  
    def set_name(self, score):  
        self.__score = score
```

问

直接通过`bart.score = 99`不是也可以修改`score`吗？为什么非要大费周折取定义一个方法呢？

### 问

直接通过 `bart.score = 99` 不是也可以修改 `score` 吗？为什么非要大费周折取定义一个方法呢？

因为在方法中，可做参数检查，避免传入无效的参数：

```
class Student(object):
    def set_name(self, score):
        if 0 <= score <= 100:
            self.__score = score
        else:
            raise ValueError('bad score')
```

### 注意：在 Python 中 `__xxx`, `_xxx`, `__xxx__` 等变量的差别

- 类似于 `__xxx` 的变量是 private 变量，外部代码不能直接访问。
- 类似于 `_xxx` 的变量允许外部代码访问，但按照约定俗成的规定，当你看到这样的变量时，意思就是，“虽然我可以被访问，但请把我视为 private 变量，不要随意访问”。
- 类似于 `__xxx__` 的变量是特殊变量，不是 private 变量，外部代码可直接访问。



问

实例变量 `__xxx` 是不是一定不能从外部访问呢？

问

实例变量 `__xxx` 是不是一定不能从外部访问呢？

其实不然。

问

实例变量`__xxx`是不是一定不能从外部访问呢？

其实不然。

不能直接访问`__name`是因为 Python 解释器对外把`__name`变量改成了`_Student__name`。

问

实例变量 `__xxx` 是不是一定不能从外部访问呢？

其实不然。

不能直接访问 `__name` 是因为 Python 解释器对外把 `__name` 变量改成了 `_Student__name`。因此，你仍然可以通过 `_Student__name` 来访问 `__name`：

```
bart = Student('Bart Simpson', 59)
print(bart._Student__name)
```

```
Bart Simpson
```

问

实例变量`__xxx`是不是一定不能从外部访问呢？

其实不然。

不能直接访问`__name`是因为 Python 解释器对外把`__name`变量改成了`_Student__name`。因此，你仍然可以通过`_Student__name`来访问`__name`：

```
bart = Student('Bart Simpson', 59)
print(bart._Student__name)
```

```
Bart Simpson
```

但是强烈建议你不要这么干，因为不同版本的 Python 解释器可能会把`__name`改成不同的变量名。

### 切记

Python 本身没有任何机制阻止你干任何事情，一切全靠自觉。

## 继承与多态

在 OOP 中定义一个 class 时，可以从某个现有的 class 继承，新的 class 称为子类 (subclass)，而被继承的 class 称为基类、父类或超类 (base class、super class)。



# 继承与多态

例:

定义一个Animal类:

```
class Animal(object):  
    def run(self):  
        print('Animal is running ...')
```

再定义Dog和Cat类时, 便可直接从Animal类继承:

```
class Dog(Animal):  
    pass
```

```
class Cat(Animal):  
    pass
```

# 继承与多态

例:

定义一个Animal类:

```
class Animal(object):  
    def run(self):  
        print('Animal is running ...')
```

再定义Dog和Cat类时, 便可直接从Animal类继承:

```
class Dog(Animal):  
    pass
```

```
class Cat(Animal):  
    pass
```

对于Dog来说, Animal是它的父类; 对于Animal来说, Dog, Cat是它的子类。

继承有什么好处呢？

继承有什么好处呢？最大的好处就是子类获得了父类的全部功能。

## 继承与多态

继承有什么好处呢？最大的好处就是子类获得了父类的全部功能。

由于Animal实现了run(), Dog和Cat作为它的子类，什么事也没干，就自动拥有了run():

```
dog = Dog()
dog.run()
cat = Cat()
cat.run()
```

## 继承与多态

继承有什么好处呢？最大的好处就是子类获得了父类的全部功能。

由于Animal实现了run()，Dog和Cat作为它的子类，什么事也没干，就自动拥有了run()：

```
dog = Dog()
dog.run()
cat = Cat()
cat.run()
```

```
Animal is running ...
Animal is running ...
```

也可以为子类增加一些方法

也可以为子类增加一些方法

例:

```
class Dog(Animal):  
    def eat(self):  
        print('Eating meat ...')  
dog = Dog()  
dog.run()  
dog.eat()
```



也可以为子类增加一些方法

例:

```
class Dog(Animal):  
    def eat(self):  
        print('Eating meat ...')  
dog = Dog()  
dog.run()  
dog.eat()
```

```
Animal is running ...  
Eating meat ...
```

继承的第二个好处是[可以改进代码](#)。

继承的第二个好处是[可以改进代码](#).

前面可以看到，无论是Dog还是Cat，调用run()时，显示的都是Animal **is** running...，符合逻辑的做法是分别显示Dog **is** running...和Cat **is** running....

## 继承与多态

```
class Dog(Animal):  
    def run(self):  
        print('Dog is running ...')  
  
class Cat(Animal):  
    def run(self):  
        print('Cat is running ...')
```

## 继承与多态

```
class Dog(Animal):  
    def run(self):  
        print('Dog is running ...')  
  
class Cat(Animal):  
    def run(self):  
        print('Cat is running ...')
```

```
Dog is running ...  
Cat is running ...
```

## 继承与多态

```
class Dog(Animal):  
    def run(self):  
        print('Dog is running ...')  
  
class Cat(Animal):  
    def run(self):  
        print('Cat is running ...')
```

```
Dog is running ...  
Cat is running ...
```

### 注

当子类 and 父类都存在相同的run()时，子类的run()覆盖了父类的run()，运行代码时总是会调用子类的run()。这样，就获得了继承的另一个好处：**多态**。

要理解什么是多态，先要对数据类型作一些说明。

要理解什么是多态，先要对数据类型作一些说明。

定义一个 class，实际上就定义了一种新的数据类型。



要理解什么是多态，先要对数据类型作一些说明。

定义一个 class，实际上就定义了一种新的数据类型。

自定义的数据类型和 Python 内置的数据类型 (如 `str`, `list`, `dict` 等) 没什么两样:

```
a = list()
b = Animal()
c = Dog()
```

## 继承与多态

判断一个变量是否是某个类型可以用 `isinstance()`:

```
print(isinstance(a, list))  
print(isinstance(b, Animal))  
print(isinstance(c, Dog))  
print(isinstance(c, Animal))
```

```
True  
True  
True  
True
```

a, b, c分别是`list`, `Animal`, `Dog`类型, 但c还是`Animal`类型.

在继承关系中，如果一个实例的数据类型是某个子类，则它的数据类型也可看做是其父类，但反过来不行：

```
print(isinstance(b, Dog))
```

```
False
```

要理解多态的好处，再来看一个例子.

# 继承与多态

要理解多态的好处，再来看一个例子.

**例：编写一个函数接受一个Animal类型的变量**

```
def run_twice( animal ):
    animal.run()
    animal.run()
run_twice(Animal())
run_twice(Dog())
run_twice(Cat())
```

要理解多态的好处，再来看一个例子。

**例：编写一个函数接受一个Animal类型的变量**

```
def run_twice(animal):  
    animal.run()  
    animal.run()  
run_twice(Animal())  
run_twice(Dog())  
run_twice(Cat())
```

```
Animal is running ...  
Animal is running ...  
Dog is running ...  
Dog is running ...  
Cat is running ...  
Cat is running ...
```

## 多态的含义

对于一个变量，只需要知道它是Animal类型，无需确切地知道它的子类型，就可以放心地调用run()方法，而具体调用的run()方法是作用在Animal，Dog还是Cat对象上，由运行时该对象的确切类型决定，这就是多态真正的威力。

## 多态的含义

对于一个变量，只需要知道它是Animal类型，无需确切地知道它的子类型，就可以放心地调用run()方法，而具体调用的run()方法是作用在Animal，Dog还是Cat对象上，由运行时该对象的确切类型决定，这就是多态真正的威力。

调用方只管调用，不管细节。当新增一种Animal的子类时，只要确保run()方法编写正确，不用管原来的代码是如何调用的。这就是著名的“开闭”原则。

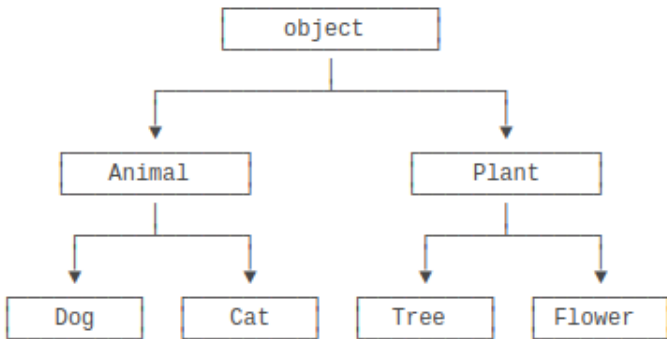


## “开闭”原则

- 对扩展开放：允许新增Animal子类；
- 对修改封闭：不需要修改依赖Animal类型的run\_twice()等函数。

## 继承与多态

继承还可以一级一级地继承下来，就好比从爷爷到爸爸、再到儿子这样的关系。而任何类，最终都可以追溯到根类 object，这些继承关系看上去就像一颗倒着的树。比如如下的继承树：



## 获取对象信息

当我们拿到一个对象的引用时，如何知道这个对象是什么类型、有哪些方法呢？

获取对象信息

使用`type()`

## 使用type()

可使用type()来判断对象类型

```
print( type(123) )
print( type('123') )
print( type(None) )
print( type(abs))
class Animal(object):
    def run(self):
        print('Animal is running ...')
a = Animal()
print( type(a) )
```

## 使用type()

可使用type()来判断对象类型

```
print( type(123) )
print( type('123') )
print( type(None) )
print( type(abs))
class Animal(object):
    def run(self):
        print('Animal is running ...')
a = Animal()
print( type(a) )
```

```
<class 'int'>
<class 'str'>
<class 'NoneType'>
<class 'builtin_function_or_method'>
<class '__main__.Animal'>
```

## 使用type()

type()返回的是class类型. 可用==比较两个变量的类型是否相同:

```
print( type(123) == type(456) )
print( type(123) == int )
print( type('123') == type('abc') )
print( type('123') == str )
print( type('123') == type(123) )
```



## 使用type()

type()返回的是class类型. 可用==比较两个变量的类型是否相同:

```
print( type(123) == type(456) )
print( type(123) == int )
print( type('123') == type('abc') )
print( type('123') == str )
print( type('123') == type(123) )
```

```
True
True
True
True
False
```

## 获取对象信息

使用 `isinstance()`

## 使用 `isinstance()`

对于 class 的继承关系来说，使用 `type()` 就很不方便。我们要判断 class 的类型，可以使用 `isinstance()`。

```
class Animal(object):  
    pass  
class Dog(Animal):  
    pass  
class Husky(Dog):  
    pass  
a = Animal()  
d = Dog()  
h = Husky()
```

## 使用 `isinstance()`

```
print(isinstance(h, Husky))
print(isinstance(h, Dog))
print(isinstance(h, Animal))
print(isinstance(d, Dog) and isinstance(d,
Animal))
print(isinstance(d, Husky))
```

## 使用 `isinstance()`

```
print(isinstance(h, Husky))
print(isinstance(h, Dog))
print(isinstance(h, Animal))
print(isinstance(d, Dog) and isinstance(d,
Animal))
print(isinstance(d, Husky))
```

```
True
True
True
True
False
```

## 使用 `isinstance()`

能用 `type()` 判断的基本类型也可以用 `isinstance()` 判断：

```
print(isinstance('a', str))
print(isinstance(123, int))
print(isinstance(b'a', bytes))
```

## 使用 `isinstance()`

能用 `type()` 判断的基本类型也可以用 `isinstance()` 判断：

```
print(isinstance('a', str))  
print(isinstance(123, int))  
print(isinstance(b'a', bytes))
```

```
True  
True  
True
```

## 使用 `isinstance()`

可使用 `isinstance()` 判断一个变量是否是某些类型中的一种:

```
print(isinstance([1, 2, 3], (list, tuple)))  
print(isinstance({'a':1, 'b':2}, (list, dict)))
```



## 使用 `isinstance()`

可使用 `isinstance()` 判断一个变量是否是某些类型中的一种:

```
print(isinstance([1, 2, 3], (list, tuple)))  
print(isinstance({'a':1, 'b':2}, (list, dict)))
```

```
True
```

```
True
```

获取对象信息

使用`dir()`

## 使用dir()

可使用dir()来获得一个对象的所有属性和方法，并以一个字符串列表返回。

```
print(dir('abc'))
```

## 使用dir()

可使用dir()来获得一个对象的所有属性和方法，并以一个字符串列表返回。

```
print(dir('abc'))
```

```
['__add__', '__class__', '__contains__', ..., 'translate', 'upper', 'zfill']
```

## 使用dir()

类似\_\_xxx\_\_的属性和方法在 Python 中都是有特殊用途的，比如\_\_len\_\_()返回长度。

在 Python 中，当你调用len()试图获取一个对象的长度，实际上，在len()内部，它自动去调用该对象的\_\_len\_\_()。故以下代码等价：

```
len('abc')  
'abc'.__len__()
```

## 使用dir()

自定义类时，如果也想使用`len(myObj)`，就自己写一个`__len__()`：

```
class MyObj(object):  
    def __len__(self):  
        return 100  
myObj = MyObj()  
print(len(myObj))
```

```
100
```

## 使用dir()

再介绍一些其他的特殊方法：

```
class Point(object):
    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)
    def __sub__(self, other):
        return Point(self.x-other.x, self.y-
            other.y)
    def __mul__(self, other):
        return self.x*other.x + self.y*other.y
    def __repr__(self):
        return f'({self.x}, {self.y})'
```

## 使用dir()

```
p1 = Point(1, 1)
p2 = Point(1, 2)
print(f'p1 + p2 = {p1 + p2}')
print(f'p1 - p2 = {p1 - p2}')
print(f'p1 * p2 = {p1 * p2}')
```



## 使用dir()

```
p1 = Point(1, 1)
p2 = Point(1, 2)
print(f'p1 + p2 = {p1 + p2}')
print(f'p1 - p2 = {p1 - p2}')
print(f'p1 * p2 = {p1 * p2}')
```

```
p1 + p2 = (2, 3)
p1 - p2 = (0, -1)
p1 * p2 = 3
```

## 使用dir()

仅仅把属性和方法列出来是不够的，配合`getattr()`、`setattr()`以及`hasattr()`，可直接操作一个对象的状态：

```
print(hasattr(p1, 'x'))
print(p1.x)
print(hasattr(p1, 'z'))
setattr(p1, 'z', 19)
print(hasattr(p1, 'z'))
print(getattr(p1, 'z'))
```

## 使用dir()

仅仅把属性和方法列出来是不够的，配合`getattr()`、`setattr()`以及`hasattr()`，可直接操作一个对象的状态：

```
print(hasattr(p1, 'x'))
print(p1.x)
print(hasattr(p1, 'z'))
setattr(p1, 'z', 19)
print(hasattr(p1, 'z'))
print(getattr(p1, 'z'))
```

```
True
1
False
True
19
```

## 使用dir()

如果试图获取不存在的属性，会抛出 `AttributeError` 的错误：

```
# print(getattr(p1, 'u'))
```

## 使用dir()

如果试图获取不存在的属性，会抛出 AttributeError 的错误：

```
# print(getattr(p1, 'u'))
```

```
Traceback (most recent call last):  
  File "src/slide02/code/dir.py", line 36, in <  
    module>  
    print(getattr(p1, 'u'))  
AttributeError: 'Point' object has no attribute  
'u'
```

## 使用dir()

可以传入一个 default 参数，如果属性不存在，就返回默认值：

```
print(getattr(p1, 'u', 404))
```

## 使用 `dir()`

可以传入一个 default 参数，如果属性不存在，就返回默认值：

```
print(getattr(p1, 'u', 404))
```

```
404
```

也可以获得对象的方法：

```
print(hasattr(p1, '__add__'))  
print(getattr(p1, '__add__'))
```



也可以获得对象的方法：

```
print(hasattr(p1, '__add__'))  
print(getattr(p1, '__add__'))
```

```
True  
<bound method Point.__add__ of (1, 1)>
```

### 注

通过内置的一系列函数，我们可以对任意一个 Python 对象进行剖析，拿到其内部的数据。要注意的是，只有在不知道对象信息的时候，我们才会去获取对象信息。如果可以直接写：

```
sum = p.x + p.y
```

就不要写

```
sum = getattr(p, 'x') + getattr(p, 'y')
```

## 实例属性与类属性

## 实例属性与类属性

由于 Python 是动态语言，根据类创建的实例可以任意绑定属性。给实例绑定属性的方法是通过实例变量，或者通过 self 变量：

```
class Student(object):  
    def __init__(self, name):  
        self.name = name  
s = Student('Wang')  
s.score = 90  
print(s.name, s.score)
```

```
Wang 90
```

## 实例属性与类属性

如果Student类本身需要绑定一个属性呢？可以直接在class中定义属性，这种属性是**类属性**，归Student类所有：

```
class Student(object):  
    name = 'Student'
```

## 实例属性与类属性

```
s = Student()
print(s.name)
print(Student.name)
s.name = 'Li'
print(s.name)
print(Student.name)
del s.name
print(s.name)
```

## 实例属性与类属性

```
s = Student()
print(s.name)
print(Student.name)
s.name = 'Li'
print(s.name)
print(Student.name)
del s.name
print(s.name)
```

```
Student
Student
Li
Student
Student
```

## 实例属性与类属性

从上面的例子可以看出，在编写程序的时候，千万不要对实例属性和类属性使用相同的名字，因为相同名称的实例属性将屏蔽掉类属性，但是当你删除实例属性后，再使用相同的名称，访问到的将是类属性。