第八章　类和对象

8.1.1 什么是面向对象编程

面向对象编程(Object Oriented Programming)，简称OOP，是一种程序设计思想，是以建立模型体现出来的抽象思维过程和面向对象的方法。模型是用来反映现实世界中事物特征的，是对事物特征和变化规律的抽象，是更普遍、更集中、更深刻地描述客体的特征。**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。**

8.1.2面向对象术语简介

面向对象常用的术语如下：

类：是创建对象的代码段，描述了对象的特征、属性、要实现的功能以及采用的方法等。

属性：描述了对象的静态特征。

方法：描述了对象的动态动作。

对象：对象是类的一个实例，就是模拟真实事件，把数据和代码都集合到一起，即属性、方法的集合。

实例：就是类的实体。

8.1.2面向对象术语简介

实例化：创建类的一个实例的过程。

封装：把对象的属性、方法、事件集中到一个统一的类中，并对调用者屏蔽其中的细节。

继承：一个类共享另一个类的数据结构和方法的机制称为继承。起始类称为基类、超类、父类，而继承类称为派生类、子类。继承类是对被继承类的扩展。

多态：一个同样的函数对于不同的对象可以具有不同的实现，就称为多态。

接口：定义了方法、属性的结构，为其成员提供规约，不提供实现。不能直接从接口创建对象，必须首先创建一个类来实现接口所定义的内容。

8.1.2面向对象术语简介

重载（overload）：一个方法可以具有许多不同的接口，但方法的名称是相同的。

事件：事件是由某个外部行为所引发的对象方法。

重写(overwrite)：在派生类中，对基类某个方法的程序代码进行重新编写，使其实现不同的功能，我们把这个过程称为重写。

构造函数：是创建对象所调用的特殊方法。

析构函数：是释放对象时所调用的特殊方法。

第八章　类和对象

8.2.1 类的定义

类就是对象的属性和方法的封装，静态的特征称为属性，动态的动作称为方法。

类通常的语法格式如下：

```
class ClassName:
    #类属性，属于类，但可在类的各实例之间共享，也就是静态变量（类变量）
    [类属性定义体]
    #方法
    __init__(self,...) :  # 构造方法
        [实例属性定义]  # 实例属性仅属于本实例，也就是实例变量
    [其他方法定义体]
```

8.2.1 类的定义

类的定义以关键字class开始，类名后面紧跟冒号“:”。例如：

```
class Clock:  # Python的类名约定以大写字母开头
    # 定义方法
    def __init__(self, hour=0, minute=0, second=0):  # 构造方法
        self.hour = hour                    # 定义实例属性并初始化
        self.minute = minute
        self.second = second

    def displayTime(self):  # 显示时间
        print('{0}:{1}:{2}'.format(self.hour, self.minute, self.second))

    def setTime(self, hour, minute, second):  # 设置时间
        self.hour = hour
        self.minute = minute
        self.second = second
```

在定义类属性的时候一般用名词，定义类方法的时候一般用动词，类名约定以大写字母开头，函数约定以小写字母开头。

8.2.2 类的使用

类定义好之后，就可将类实例化为对象。类实例化对象的语法格式如下：

对象名 = 类名（属性初始化的参数列表）

实例化对象的操作符是：等号“=”，在类实例化对象的时候，类名后面要添加一个括号“(属性初始化的参数列表)”，如果没有参数值或有默认参数值，则直接用"()"。

类实例化示例：

```
>>> clock = Clock()
```

上例将类Clock实例化为对象clock。

对象成员（属性和方法）的访问则通过成员运算符“.”来完成。例如：

```
>>> clock.displayTime()
0:0:0
```

在定义类的模块内访问该类

```
#  demo_08_01_01.py
class Clock:  # Python的类名约定以大写字母开头
    # 定义方法
    ...

def main():
    clock = Clock()  # 类Clock实例化为clock，即创建时钟对象
    clock.displayTime()  # 调用对象的方法
    clock.setTime(15, 40, 58)
    clock.displayTime()
    clock.hour = 16  # 访问公有属性
    clock.displayTime()

    clock2 = Clock(1, 2, 3)  # 类Clock实例化为clock2，即创建时钟对象
    clock2.displayTime()  # 调用对象的方法
"""
如果本python文件直接作为脚本执行，即作为主程序运行，则__name__的值为'__main__';
如果作为模块运行，则__name__的值为本模块名。
"""
# print(__name__)
if __name__ == "__main__":
    main()
```

```
0:0:0
15:40:58
16:40:58
1:2:3
```

在定义类的模块所在包（ch8）的另一个模块中访问该类

```
#  demo_08_01_02.py
import demo_08_01_01


def main():
    clock = demo_08_01_01.Clock()  # 类Clock实例化为clock，即创建时钟对象
    clock.displayTime()  # 调用对象的方法
    clock.setTime(15, 40, 58)
    clock.displayTime()
    clock.hour = 16  # 访问公有属性
    clock.displayTime()

    clock2 = demo_08_01_01.Clock(1, 2, 3)  # 创建时钟对象clo
    clock2.displayTime()

main()
```

```
0:0:0
15:40:58
16:40:58
1:2:3
```

如果要在定义类的模块所在包（ch8）之外访问该类，则先将该包拷贝到目录在C:\\Program Files\\Python38\\lib\\site-packages，这样导入包及其模块时import就能搜索到它。

8.2.3 类的构造方法、析构方法及专有方法

Python类的方法可以分为两大类：普通方法与内置方法。

- 普通方法包括实例方法、静态方法和类方法。
- 内置方法是Python中一类特殊的方法，也称为魔法方法，其名称是由Python内置的，不可以自己随意起名。
- 与普通方法最大的区别是定义方法时：方法名的命名上，内置方法命名都是以__开始和结束，例如：__init__, __del__, __str__, __eq__, __ne__, __gt__, __ge__, __lt__, __le__等。
- 普通方法需要通过类的实例对象根据方法名调用，而内置方法是在特定情况下由系统自动执行。

8.2.3 类的构造方法、析构方法及专有方法

（1）类的构造方法是：**__init__(self)**。

只要实例化一个对象的时候，这个方法就会在对象被创建的时候自动调用。实例化对象的时候是可以传入参数的，这些参数会自动传入__init__(self, param1, param2, …)方法中，我们可以通过**重写**这个方法来自定义**对象的初始化**操作。

如在Clock类中添加构造方法：

```
def __init__(self, hour=0, minute=0, second=0):  # 构造方法
    self.hour = hour                # 定义实例属性并初始化
    self.minute = minute
    self.second = second
```

8.2.3 类的构造方法、析构方法及专有方法

（2）类的析构方法是：**__del__(self)**。

一个对象被销毁的时候，这个方法就会被自动调用。

例如在Clock类中添加析构方法：

```
def __del__(self):  # 析构方法
    print("调用析构方法")
```

```
#  demo_08_01_05.py
class Clock:  # Python的类名约定以大写字母开头
      # 定义方法
     def __del__(self):  # 析构方法
           print("调用了析构方法")
      ...

def main():
      clock = Clock()  # 类Clock实例化为clock
      clock.displayTime()  # 调用对象的方法
      clock.setTime(15, 40, 58)
      clock.displayTime()
      clock.hour = 16  # 访问公有属性
      clock.displayTime()
      del clock   # 删除对象clock

      clock2 = Clock(1, 2, 3)  # 类Clock实例化为clock2
      clock2.displayTime()  # 调用对象的方法

if __name__ == "__main__":
      main()
```

```
0:0:0
15:40:58
16:40:58
调用了析构方法
1:2:3
调用了析构方法
```

- **del 一个变量且其引用的对象不再被其他变量引用后，系统自动回收时析构函数被自动调用；**
- **程序结束，对象销毁时也会自动调用其析构函数。**

8.2.3 类的构造方法、析构方法及专有方法

(3) __str__(self)方法

重写该方法，将对象转化为恰当的字符串，例如print(clock)，clock将自动调用__str__()方法返回字符串格式的信息，否则默认输出信息类似于：

<__main__.Clock object at 0x00000226145EA400>

例如在Clock类中重写__str__()方法：

```
def __str__(self):  # 重写该方法，以便将对象转化为恰当的字符串
    return '现在时刻为{0}:{1}:{2}'.format(self.hour, self.minute, self.second)
```

```
#  demo_08_01_06.py
class Clock:  # Python的类名约定以大写字母开头
      # 定义方法
     def __str__(self):  # 重写该方法，以便将对象转化为恰当的字符串
          return '现在时刻为{0}:{1}:{2}'.format(self.hour, self.minute, self.second)
      ...

def main():
      clock = Clock()  # 创建时钟对象
      print(clock) # 等价于 print(str(clock)) ， 自动调用__str__()方法
      clock.setTime(15, 40, 58)
      print(clock)
      clock2 = Clock(1, 2, 3)  # 创建时钟对象
      print(clock2)

if __name__ == "__main__":
      main()
```

现在时刻为0:0:0
现在时刻为15:40:58
现在时刻为1:2:3

8.2.3 类的构造方法、析构方法及专有方法

（4）__repr__(self)方法

重写该方法，以规定本类对象的序列（如列表、元组、集合）输出时，各元素如何转换为恰当的字符串。例如print([clock1,clock2])时，列表中各Clock对象将自动调用__repr__()方法返回字符串表示, 得到字符串列表：

['现在时刻为0:0:0', '现在时刻为0:0:0'] .

否则默认输出信息类似于：

[<__main__.Clock object at 0x0000025DB53A9E50>,

<__main__.Clock object at 0x0000025DB5400C40>]

```
def __repr__(self):  # 本类对象的序列输出时，规定元素如何转换为字符串
    return str(self)  # 返回对象的字符串表示
```

8.2.3 类的构造方法、析构方法及专有方法

（5）为属性添加getter/setter方法

这是一种编程规范，getter/setter方法用于读写对象某个属性值。命名规则：

getXxx()/setXxx(xxx), 其中xxx为属性名

例如在Clock类中为属性hour添加getter/setter方法：

```
def getHour(self):  # getter
    return self.hour

def setHour(self, hour):  # setter
    self.hour = hour
```

```
#  demo_08_01_07.py
class Clock:
    def __init__(self, hour=0, minute=0, second=0):  # 构造方法
        self.hour = hour  # 实例属性,仅属于本实例，也就是实例变量
        self.minute = minute
        self.second = second
    def __str__(self):  # 重写该方法，以便将对象转化为恰当的字符串
        return '现在时刻为{0}:{1}:{2}'.format(self.hour, self.minute, self.second)

    def getHour(self):  # getter
        return self.hour
    def setHour(self, hour):  # setter
        self.hour = hour
def main():
    clock = Clock()  # 类Clock实例化为clock，即创建时钟对象
    print(clock)  # 等价于 print(str(clock)) ，自动调用__str__()方法
    clock.setHour(8)  # 调用setter
    print(clock)

if __name__ == "__main__":
    main()
```

现在时刻为0:0:0
现在时刻为8:0:0

8.2.3 类的构造方法、析构方法及专有方法

（6）为属性添加deleter方法

这是一种编程规范，deleter方法用于动态删除对象某个属性值。命名规则：

delXxx(), 其中xxx为属性名

例如在Clock类中为属性hour添加deleter方法：

```
def delHour(self):  # 删除属性
    del self.hour
```

8.2.4 类的访问权限

大家都知道，在C++/Java中，是通过关键字public、protected、private来表明访问的权限是公有、保护的、私有的。然而**在Python中，默认情况下对象的属性和方法都是公开的，公有的，通过点(.)操作符来访问**。

如下例所示代码：

```
clock = Clock()  # 类Clock实例化为clock，即创建时钟对象
print(clock)  # 等价于 print(str(clock))，自动调用__str__()方法
clock.setHour(8)  # 调用setter
print(clock)
clock.minute = 9  # 访问公有变量（属性）
print(clock)
```

```
现在时刻为0:0:0
现在时刻为8:0:0
现在时刻为8:9:0
```

8.2.4 类的访问权限

为了实现类似于私有变量的特征，Python内部采用了name mangling的技术（名字重整或名字改变），在**变量名或函数名**前加上**两个下划线**（“__”），这个函数或变量就变为**私有**了。

为了访问类中的私有变量，有一个折衷的处理办法，即在类中添加一个访问私有变量公有的getter方法，从而使该变量只读。

如下例所示代码：

```
class Clock:
    def __init__(self, hour=0, minute=0, second=0):  # 构造方法
        self.__hour = hour  # 私有实例属性,仅属于本实例，也就是实例变量
        self.__minute = minute
        self.__second = second

    def __str__(self):  # 重写该方法，以便将对象转化为恰当的字符串
        return '现在时刻为{0}:{1}:{2}'.format(self.__hour, self.__minute,
self.__second)

    def getHour(self):  # getter
        return self.__hour

def main():
    clock = Clock(15, 36, 20)  # 类Clock实例化为clock，即创建时钟对象
    print(clock)  # 等价于 print(str(clock)) ，自动调用__str__()方法
    # print(clock.__hour)  # AttributeError: 'Clock' object has no attribute '__hour'
    print(clock.getHour())

if __name__ == "__main__":
    main()
```

```
现在时刻为15:36:20
15
```

8.2.4 类的访问权限

实际上，Python把双下划线“__”开头的变量名，改为了：**单下划线“_”类名+双下划线“__”变量名，即【_类名__变量名】**，因此我们可以通过以下的访问方式来访问类的私有变量。例如，在上例中添加一行代码：

```
def main():
    clock = Clock(15, 36, 20)  # 类Clock实例化为clock，即创建时钟对象
    print(clock)  # 等价于 print(str(clock))，自动调用__str__()方法
    # print(clock.__hour)  # AttributeError: 'Clock' object has no attribute '__hour'
    print(clock._Clock__hour)
    print(clock.getHour())
```

现在时刻为15:36:20
15
15

可见，就目前而言，Python的私有机制是伪私有，Python的类是没有权限控制的，变量是可以被外部调用的。

私有属性如何用象名.属性的形式读取?

Python 中提供了内置的 property() 函数，可以实现在不破坏类封装原则的前提下，让开发者依旧使用“对象.属性”的方式操作对象的属性。

property() 函数的基本使用格式如下：

属性名=property(fget=None, fset=None, fdel=None, doc=None)

- fget -- 获取属性值的函数，**对象.属性**触发getter
- fset -- 设置属性值的函数，**对象.属性=value**时触发setter
- fdel -- 删除属性值函数，**del 对象.属性**时触发deleter
- doc -- 属性描述信息。如果给定 doc 参数，其将成为这个属性的 docstring，否则 property 函数就会复制 fget 函数的 docstring（如果有的话）

通过属性访问，鼠标停留在该属性上时，提示信息中将显示其docstring。或者，help(对象)时，也将显示其docstring。

```
class C:
    def __init__(self, x=0):  # 构造方法
        self.__x = x  # 私有实例属性,仅属于本实例，也就是实例变量

    def getX(self):  # getter
        return self.__x

    def setX(self, x):  # setter
        self.__x = x

    def delX(self):  # deleter
        print('删除私有变量__x')
        del self.__x

# 定义property x
x = property(fget=getX, fset=setX, fdel=delX, doc="I'm the 'x' property.")
```

如果 c 是 C 的实例化, c.x 将触发 getter, c.x = value 将触发 setter, del c.x 触发 deleter。

如果给定 doc 参数，其将成为这个属性值的 docstring，否则 property 函数就会复制 fget 函数的 docstring（如果有的话）。

```
c = C(1949)
c2 = C(2020)
print(c.getX())  # 通过getter方法实现
print(c.x)  # 通过property x 访问，鼠标停留在x上，提示信息中将显示x的docstring
print(c2.x)
c.x = 1978  # 通过property x 访问，鼠标停留在x上，提示信息中将显示x的docstring
print(c.x)
print(c2.x)
del c.x  # 通过property x 访问，  鼠标停留在x上，提示信息中将显示x的docstring
help(c)
```

1949
1949
2020
1978
2020
删除私有变量__x

```
Help on C in module __main__ object:

class C(builtins.object)
 |  C(x=0)
 |
 |  Methods defined here:
 |
 |  __init__(self, x=0)
 |      Initialize self.  See help(type(self)) for accurate signature.
.....
 |  x
 |      I'm the 'x' property.
```

将 property 函数用作装饰器可以很方便的创建只读属性。

```
class C:
    def __init__(self, x=0):  # 构造方法
        self.__x = x  # 私有实例属性,仅属于本实例，也就是实例变量

    @property
    def x(self):  # 创建只读属性x, 实际上是一个方法
        """ I'm the 'x' property. """
        return self.__x

c = C(1949)
print(c.x)  # 通过property x 访问，鼠标停留在x上，提示信息中将显示x的docstring
c.x = 1978  # AttributeError: can't set attribute
```

```
Traceback (most recent call last):
  File "C:/Users/ruanz/PycharmProjects/new/ch8/demo_08_01_10.py", line 20, in <module>
    c.x=1978  # Property 'x' cannot be set
AttributeError: can't set attribute
1949
```

```
class C:
    def __init__(self, x=0):  # 构造方法
        self.__x = x  # 私有实例属性,仅属于本实例，也就是实例变量
    @property
    def x(self):  # 创建只读属性，实际上是一个方法
        """ I'm the 'x' property. """
        return self.__x

    @x.setter
    def x(self, x):   # setter
        self.__x = x

    @x.deleter
    def x(self):    # deleter
        print('删除私有属性__x')
        del self.__x

c = C(1949)
print(c.x)
c.x = 1978
print(c.x)
del c.x
```

- property 的 getter,setter 和 deleter 方法同样可以用作装饰器。
- 注意这些额外函数的名字和 property 下的一样，例如这里都是 x。

由于添加了setter方法，所以属性x不再是只读，而是可读可写了。

```
1978
删除私有属性__x
```

8.2.5 类变量，类方法（class method），静态方法(static method)

（1）类变量

不同于实例变量，**类变量**属于类，而不在各对象中存在，但是类的所有实例都可以共享。

```
class 类名:
    类变量1=value1
    类变量2=value2
    ...
```

访问时类变量，直接使用类名限定即**类名.类变量**，如果对象存在，也可以通过对象名限定即**对象名.类变量**。

例如，在Clock类中定义类变量count，用于统计对象个数。代码如下：

```
class Clock:
    # 定义类属性，属于类，但可在类的各实例之间共享
    count = 0  # Clock对象计数

    # 定义方法以及实例变量
    def __init__(self, hour=0, minute=0, second=0):  # 构造方法
        self.hour = hour  # 实例属性,仅属于本实例，也就是实例变量
        self.minute = minute
        self.second = second
        Clock.count += 1  # 每产生一个Clock对象，计数器就加一

    def __del__(self):  # 析构方法
        Clock.count -= 1  # 当一个Clock对象消亡时，计数器就减一
```

```
def main():
    print('Clock.count=', Clock.count)  # 通过类名限定
    print('-----' * 4)
    clock = Clock()  # 创建时钟对象
    print('Clock.count=', Clock.count)
    print('clock.count=', clock.count)  # 通过对象名限定
    print('-----'*4)
    clock2 = Clock(1, 2, 3)  # 创建时钟对象
    print('Clock.count=', Clock.count)
    print('clock2.count=', clock2.count)
    print('clock.count=', clock.count)

if __name__ == "__main__":
    main()
```

```
Clock.count= 0
----------
Clock.count= 1
clock.count= 1
----------
Clock.count= 2
clock2.count= 2
clock.count= 2
```

Clock.count = clock2.count = clock.count = 2

可见，Clock的两个对象共享了类变量count。

8.2.5 类变量，类方法（class method），静态方法(static method)

（2）实例方法

如果类的一个方法中需要访问实例变量，则该方法就应定义为**实例方法（instance method）**，其第一个参数通常命名为**self**，表示这个实例本身，形如：

```
def func(self,...) : # 定义实例方法
```

self其实也改为任意的合法的名字，例如this、selfX等。

前面例子中定义的__init__(self,...)、__del__(self)、getX(self)、setX(self, x)等都是实例方法，除了构造方法等特殊方法自动调用，其他实例方法都是通过对象名限定访问的，即**对象名.实例方法(...)** 。

8.2.5 类变量，类方法（class method），静态方法(static method)

（3）类方法

如果类的一个方法中不使用实例变量，但需要访问类变量，则该方法就应定义为**类方法（class method）**，类方法必须使用指令**@classmethod**加以装饰，其第一个参数通常命名为**cls**，代表类，语法形如：

```
@classmethod
def func(cls,...) : # 定义类方法
```

cls其实也改为任意的合法的名字，例如clsX等。

类方法的访问与类变量相似，一般类名限定，即**类名.类方法(...)**，如果存在该类的实例，也可以通过对象名限定，即**对象名.类方法(...)** 。

例如，将上例中公有类变量count改为私有，然后为类添加一个读取count的公有方法getCount(cls)，代码如下：

```
class Clock:
    # 定义类属性，属于类，但可在类的各实例之间共享
    __count = 0  # Clock对象计数，私有类变量

    # 定义方法
    def __init__(self, hour=0, minute=0, second=0):  # 构造方法
        self.hour = hour  # 实例属性,仅属于本实例，也就是实例变量
        self.minute = minute
        self.second = second
        Clock.__count += 1  # 每产生一个Clock对象，计数器就加一

    def __del__(self):  # 析构方法
        Clock.__count -= 1  # 当一个Clock对象消亡时，计数器就减一

    @classmethod
    def getCount(cls):    # 定义类方法
        return cls.__count
```

```
def main():
        print('Clock.count=', Clock.getCount())  # 通过类名限定
        print('-----' * 4)
        clock = Clock()  # 创建时钟对象
        print('Clock.count=', Clock.getCount())
        print('clock.count=', clock.getCount())  # 通过对象名限定
        print('-----'*4)
        clock2 = Clock(1, 2, 3)  # 创建时钟对象
        print('Clock.count=', Clock.getCount())
        print('clock2.count=', clock2.getCount())
        print('clock.count=', clock.getCount())

if __name__ == "__main__":
    main()
```

```
Clock.count= 0
----------
Clock.count= 1
clock.count= 1
----------
Clock.count= 2
clock2.count= 2
clock.count= 2
```

8.2.5 类变量，类方法（class method），静态方法(static method)

（4）静态方法

如果类的一个方法中既不使用实例变量，也不访问类变量，则该方法就应定义为**静态方法**（static method），静态方法必须使用指令**@staticmethod**加以装饰，语法形如：

```
@staticmethod
def func(...) : # 定义静态方法
```

静态方法的访问与类方法相似，一般类名限定，即**类名.静态方法(...)**，如果存在该类的实例，也可以通过对象名限定，即**对象名.静态方法(...)** 。

类中用于诸如单位换算、数学计算等与属性无关的方法通常就声明为静态方法。

例如，在类中添加stat方法，用于统计列表或元组的最大值、最小值和平均值。

```
class C:
    def __init__(self, x):  # 构造函数
            self.x = 0

    @staticmethod  # 定义静态方法
      def stat(iterable):  # 参数为列表list或元组tuple！
            """ 用于统计列表或元组的最大值、最小值和平均值 """
       if (not isinstance(iterable, (tuple, list))) or len(iterable) == 0:
          raise Exception("参数应为元素非空的列表list或元组tuple！ ")  # 抛出异常
            for x in iterable:
          if not isinstance(x, (int, float)):  # 检查元素类型是否正确
                        raise Exception("元素类型错误！ ")  # 抛出异常
            return max(iterable), min(iterable), sum(iterable) / len(iterable)

a = [3, 0, 5, 12, 8]
x, y, z = C.stat(a)
print('a=', a)
print('max={},min={},mean={}'.format(x, y, z))
c = C(2)
print(c.stat(a))
```

```
a= [3, 0, 5, 12, 8]
max=12,min=0,mean=5.6
(12, 0, 5.6)
```

8.2.6 实现方法重载（overload）效果

在OOP中，**方法重载（overload）**指的是一个方法可以具有许多不同的接口，但**方法的名称是相同**的。例如，初始化一个Clock对象时，我们希望可以给构造方法传入表示时、分、秒的三个值，也可以传入一个已知的Clock实例，即：

```
c1=Clock(13,58,25)

c2=Clock(c1)
```

因此需要定义两个构造方法，分别含有1个参数和3个参数。

重载方法通过参数个数不同或参数类型不同来区分的。

如果 Python 按类似的方式定义方法，不会报错，只是后面的方法定义会覆盖前面的，达不到重载的效果。

```
class Clock:
    # 定义方法
    def __init__(self):  # 无参构造方法
        self.hour, self.minute, self.second = 0, 0, 0

    def __init__(self, hour, minute, second):  # 构造方法
        self.hour, self.minute, self.second = hour, minute, second

    def __init__(self, clock):  # 构造方法
        if not isinstance(clock, Clock):
            raise Exception('参数类型错误！')
        # 拷贝对象属性
        self.hour = clock.hour
        self.minute = clock.minute
        self.second = clock.second

c1 = Clock()
c2 = Clock(15, 40, 58)
c3 = Clock(c2)
```

Python中，后面的方法定义会覆盖前面的同名方法，这样定义达不到重载的效果。

```
...
    c1 = Clock()
TypeError: __init__() missing 1 required positional argument: 'clock'
```

8.2.6 实现方法重载（overload）效果

但是Python 函数的形参十分灵活，我们可以只定义一个函数来实现相同的功能，其中利用可变参数。

```
class Clock:
def __init__(self, *args):
    if len(args) == 0:
        self.hour, self.minute, self.second = 0, 0, 0
    elif len(args) == 1:
        if not isinstance(args[0], Clock):
            raise Exception('参数类型错误')
        clock = args[0]
        self.hour,  self.minute , self.second = clock.hour, clock.minute, clock.second
    elif len(args) == 3:
        self.hour,  self.minute,  self.second = args[0], args[1], args[2]
    else:
        raise Exception('参数错误！ ')

def __str__(self):  # 重写该方法，以便将对象转化为恰当的字符串
    return '{}:{}:{}'.format(self.hour, self.minute, self.second)
```

8.2.6 实现方法重载（overload）效果

但是Python 函数的形参十分灵活，我们可以只定义一个函数来实现相同的功能，其中利用可变参数。

```
c1 = Clock()                # 传入0个参数
c2 = Clock(15, 40, 58)      # 传入3个参数
c3 = Clock(c2)              # 传入1个参数
print(c1)
print(c2)
print(c3)
c2.hour = 19
print(c2)
print(c3)
```

```
0:0:0
15:40:58
15:40:58
19:40:58
15:40:58
```

python是一门动态语言，不需要声明变量类型，函数中可以接受任何类型的参数也就无法根据参数类型来支持重载，python没有必要去考虑参数的类型问题，这些都可以在函数内部判断处理，并无必要去在写一个函数。python 有多种传参方式，默认参数/可变参数/可变关键字参数可以处理函数参数中参数可变的问题。

在函数中也可以使用 functools.singledispatch 装饰器实现重载效果

用 @singledispatch装饰的函数，被称为**泛型函数（generic function）**。
@singledispatch： Single-dispatch generic function decorator.

```
from functools import singledispatch

@singledispatch  # 用 @singledispatch装饰的泛型函数
def func(a):
    print(f'Other: {a}')

@func.register(int)
def abc_(a):   # 函数名可以任意，一般使用单个下划线_，这是最简洁的形式
        print(f'Int: {a}')

@func.register(float)
def _(a):
    print(f'Float: {a}')

func(5)
func(5.0)
func('python')
```

```
Int: 5
Float: 5.0
Other: python
```

- **func 函数被 functools.singledispatch 装饰后，又根据不同的形式参数类型绑定了另外两个函数。**
- **当实际参数类型为整形或者浮点型时，调用绑定的对应的某个函数，否则，调用自身。**
- **仅根据第一个参数的不同类型来区分实际调用的函数**

在函数中也可以使用 functools.singledispatch 装饰器实现重载效果

用 @singledispatch装饰的函数，被称为**泛型函数（generic function）**。
@singledispatch： Single-dispatch generic function decorator.

```
from functools import singledispatch

@singledispatch  # 用 @singledispatch装饰的泛型函数
def func(a,b):
    print(f'Other: {a},{b}')

@func.register(int)
def abc_(a, b):   # 函数名可以任意，一般使用单个下划线_，这是最简洁的形式
        print(f'Int: {a},{b}')

@func.register(float)
def _(a,b):
    print(f'Float: {a},{b}')

func(5, 'China')
func(5.0, 'China')
func('python', 'China')
```

```
Int: 5,China
Float: 5.0,China
Other: python,China
```

- **泛型函数func(...)有多个参数时，仅根据第一个参数的不同类型来区分实际调用的函数**
- **@func.register(type)参数只有一个，表示数据类型**

第八章　类和对象

8.3.1 封装

从形式上看，对象封装了属性就是变量，而方法和函数是独立性很强的模块，封装就是一种信息掩蔽技术，使数据更加安全。

例如，列表(list)是Python的一个序列对象，我们要对列表进行调整，如下所示代码：

```
>>> list1 = ['K','J','L','Q','M']
>>> list1.sort()
>>> list1
['J', 'K', 'L', 'M', 'Q']
```

在上例中，我们调用了排序函数sort()对无序的列表进行了正序排序。

8.3.1 封装

由此可见，Python的列表就是对象，它提供了若干种方法来供我们根据需求调整整个列表，但是我们不知道列表对象里面的方法是如何实现的，也不知道列表对象里面有哪些变量，这就是封装。它封装起来，只给我们需要的方法的名字，然后我们调用这个名字，知道它可以实现就可以了，然而它没有具体告诉我们是怎么实现的。

8.3.2 多态

不同对象对同一方法响应不同的行动就是多态。

```
class Test_X:
    def func(self):
        print("测试X...")

class Test_Y:
    def func(self):
        print("测试Y...")

x = Test_X()
y = Test_Y()
x.func()  # 输出：测试X...
y.func()  # 输出：测试Y...
```

类Test_X()的实例对象为：x，类Test_Y()的实例对象为：y，对象x和对象y分别调用了同名函数func()，分别运行得到了不同的结果：测试X…和测试Y…。由此可见，不同对象调用了同名方法(函数)，实现的结果不一样，这就是多态。

注意：self相当于C++的this指针。由同一个类可以生成无数个对象，这些对象都来源于同一个类的属性和方法，当一个对象的方法被调用的时候，对象会将自身作为第一个参数传给self参数，接收到self参数的时候，Python就知道是哪一个对象在调用方法了。

8.3.3 继承

继承是子类自动共享父类的数据和方法的机制。

语法格式如下：

```
Class ClassName(BaseClassName):

        ……
```

ClassName：是子类的名称，第一个字母必须大写。

BaseClassName：是父类的名称。

被继承的类我们称为基类、父类或超类，而**继承者我们称为子类**。子类可以继承父类的任何属性和方法。

我们以列表对象为例，如下所示程序代码：

```
class Test_list(list): # 从父类list类继承
    def show(self):  # 新增方法
        print(self)

    def __str__(self):  # 重写父类list的str方法，输出元素间隔改为两个空格
        temp = "['" + self[0]
        for x in self[1:]:
            temp += "',  '" + x
        temp += "']"
        return temp

list1 = Test_list()
list1.append('O')  # 调用从父类继承的方法
print(list1)  # 自动调用重写的str()方法
list1.append('MT')
print(list1)
list1.sort()  # 调用从父类继承的方法
print(list1)
list1.show()  # 调用新增的方法
```

- **子类Test_list继承了父类list的append(), sort()等方法和属性**
- **新增了show()方法**
- **重写了父类的__str__()方法**

```
['O']
['O',  'MT']
['MT',  'O']
['MT',  'O']
```

8.3.3 继承

在使用类继承机制的时候，有几点需要注意：

(1)如果子类中定义与父类同名的方法或属性，则会自动覆盖父类对应的方法或属性。

(2)子类重写了父类的方法就会把父类的同名方法覆盖，如果被重写的子类同名的方法里面没有引入父类同名的方法，实例化对象要调用父类的同名方法的时候，程序就会报错。

要解决上述问题，就要在子类里面重写父类同名方法的时候，先引入父类的同名方法。要实现这个继承的目的，有2种技术可采用：一是调用未绑定的父类方法，二是使用super函数。

8.3.3 继承

我们先看看“调用未绑定的父类方法”的技术，它的语法格式如下：

Parent.func(self)

语法各项解释如下：

Parent——父类的名称；

func——子类要重写的父类的同名方法名称；

self——子类的实例对象，注意这里不是父类的实例对象。

8.3.3 继承

接下来，我们看看“使用super函数”的技术，super函数可以自动找到基类的方法和传入self参数，它的语法格式如下：

super().func([parameter])

super()——super函数，代表父类；

func——子类要重写的父类的同名方法名称；

parameter——可选参数，如果参数是self可以省略。

使用super函数的方便之处在于不用写任何基类的名称，直接写重写的方法就可以了，这样Python会自动到基类去寻找，尤其是在多重继承中，或者子类有多个祖先类的时候，super函数会自动到多种层级关系里面去寻找同名的方法。

8.3.4 多重继承

可以同时继承多个父类的属性和方法，称为多重继承。语法格式如下：

```
Class ClassName(Base1, Base2, Base3):

    ……
```

ClassName ——子类的名字；

Base1, Base2, Base3——基类1的名字，基类2的名字，基类3的名字；有多个基类时，名字依次写入即可。

虽然多重继承的机制可以让子类继承多个基类的属性和方法使用起来很方便，但很容易导致代码混乱，有时候会引起不可预见的Bug，对程序而言不可预见的Bug几乎就是致命的。**因此，当我们不确定必须要使用“多重继承”语法的时候，尽量避免使用它。**

第八章　类和对象

8.4.1 声明类

8.4.2 类的继承和多态

8.4.3 复制对象（注意区分浅拷贝与深拷贝）

第八章　类和对象

我们发现想要设计一门出色的语言，就要从现实世界里面去寻找，学习，并归纳抽象出真理包含到其中。

继承可以把父类的所有功能都直接拿过来，这样就不必从零做起，子类只需要新增自己特有的方法，也可以把父类不适合的方法覆盖重写。

继承可以一级一级地继承下来。而任何类，最终都可以追溯到根类。

有了继承，才能有多态。

第八章　类和对象

习题：

1. 面向对象程序设计的特点分别为？
2. 假设c为类C的对象且包含一个私有数据成员“__name”，那么在类的外部通过对象c直接将其私有数据成员“__name”的值设置为kate的语句可以写作什么？
3. 设计一个学生类Student,具有学号、姓名、性别等私有实例属性，设计构造方法、析构方法，各属性具有相应的getter方法、setter方法，重写__str__方法，学号从1900开始。在程序中，使用list来管理学生对象，并按学生升序排列各对象。
4. 定义一个圆类Circle，具有私有实例属性radius，表示半径，设计构造方法、计算圆面积的方法和计算周长的方法， 编写radius属性的getter方法、setter方法，重写__str__方法。在程序中，使用list来管理圆对象，按半径升序排列各对象，并将半径大于5的圆筛选出来。

[C1, C2, C3 ……]. lambda X: X.radius / X.getRadius()

感谢聆听