第八章 类和对象

- 本章要点：
  - ✓ 面向对象概念
  - ✓ 类对象和实例对象
  - ✓ 属性
  - ✓ 方法
  - ✓ 继承
  - ✓ 对象的特殊方法
  - ✓ 对象的引用、浅拷贝和深拷贝

面向对象概念

- 对象的定义
  - 某种事物的抽象（功能）
  - 抽象原则包括数据抽象和过程抽象两个方面
    - 数据抽象-定义对象属性；过程抽象-定义对象操作
- 封装
  - 把客观事物抽象并封装成对象
- 继承
  - 允许使用现有类的功能，并在无需重新改写原来的类的情况下，对这些功能进行扩展
- 多态性：对象可以表示多个类型的能力

类和对象

- 类与对象的关系~车型设计和具体的车

```
class 类名:
    类体
```

- 类的声明
  - 【例8.1】定义类Person1（Person1.py）

```
class Person1:  #定义类Person1
    pass      #类体为空语句
#测试代码
p1 = Person1()    #创建和使用类对象
print(Person1, type(Person1), id(Person1))
print(p1, type(p1), id(p1))
```

程序运行结果如下。

```
<class '__main__.Person1'> <class 'type'> 1247819394248
<__main__.Person1 object at 0x00000122880690F0> <class '__main__.Person1'> 1247822647536
```

- 对象的创建和使用

```
anObject = 类名(参数列表)
anObject.对象函数 或 anObject.对象属性
```

- 【例8.2】实例对象的创建和使用示例

```
>>> c1 = complex(1, 2)
>>> c1.conjugate()      #输出：(1-2j)
(1-2j)
>>> c1.real             #输出：1.0
1.0
```

属性（1）

- 类中定义的成员变量
- 实例属性
  - 通过self.变量名定义的属性
  - 【例8.3】定义类Person2。定义成员变量（域）

```
class Person2:                #定义类Person2
    def __init__(self, name,age): #__init__方法
        self.name = name     #初始化self.name，即成员变量name（域）
        self.age = age       #初始化self.age，即成员变量age（域）
    def say_hi(self):        #定义类Person2的函数say_hi()
        print('您好, 我叫', self.name) #在实例方法中通过self.name读取成员变量name（域）
#测试代码
p1 = Person2('张三',25)      #创建对象
p1. say_hi ()               #调用对象的方法
print(p1.age)               #通过p1.age（obj1.变量名）读取成员变量age（域）
```

程序运行结果如下

```
您好, 我叫 张三
25
```

属性（2）

- 类属性：类本身的变量
  - 【例8.4】定义类Person3。定义类域和类方法

```
class Person3:
    count = 0        #定义属性count，表示计数
    name = "Person"  #定义属性name，表示名称
#测试代码
Person3.count += 1    #通过类名访问，将计数加1
print(Person3.count)   #类名访问，读取并显示类属性
print(Person3.name)   #类名访问，读取并显示类属性
p1 = Person3()        #创建实例对象1
p2 = Person3()        #创建实例对象2
print((p1.name, p2.name))  #通过实例对象访问，读取成员变量的值
Person3.name = "雇员"    #通过类名访问，设置类属性值
print((p1.name, p2.name))  #读取成员变量的值
p1.name = "员工"   #通过实例对象访问，设置实例对象成员变量的值
print((p1.name, p2.name))  #读取成员变量的值
```

程序运行结果如下

```
1
Person
('Person', 'Person')
('雇员', '雇员')
('员工', '雇员')
```

属性（3）

- 私有属性和公有属性
  - 两个下划线开头，但是不以两个下划线结束的属性是私有的（private），其他为公共的（public）
  - 【例8.5】私有属性示例（private.py）

```
class A:
    __name = 'class A'   #私有类属性
    def get_name():
        print(A.__name) #在类方法中访问私有类属性
#测试代码
A.get_name()
A.__name          #导致错误，不能直接访问私有类属性
```

程序运行结果如下：

```
class A
Traceback (most recent call last):
  File "C:\Pythonpa\ch09\private.py", line 7, in <module>
    A.__name                #导致错误，不能直接访问私有类属性
AttributeError: type object 'A' has no attribute '__name'
```

属性（4）

- @property装饰器
  - 【例8.6】property装饰器示例1

```
class Person11:
    def __init__(self, name):
        self.__name = name
    @property
    def name(self):
        return self.__name
#测试代码
p = Person11('王五')
print(p.name)
```

程序运行结果如下

王五

属性（5）

- 【例8.7】property装饰器示例2

```
class Person12:
    def __init__(self, name):
        self.__name = name
    @property
    def name(self):
        return self.__name
    @name.setter
    def name(self, value):
        self.__name = value
    @name.deleter
    def name(self):
        del self.__name
#测试代码
p = Person12('姚六')
p.name = '王依依'
print(p.name)
```

程序运行结果如下

王依依

方法

方法的声明格式如下：

```
def 方法名(self,[形参列表]):
    函数体
```

方法的调用格式如下：

```
对象.方法名([实参列表])
```

- 【例8.10】实例方法示例。定义类Person4，创建其对象，并调用对象函数

```
class Person4:          #定义类Person4
    def say_hi(self, name): #定义方法say_hi
        self.name = name #把参数name赋值给self.name，即成员变量name（域）
        print('您好, 我叫', self.name)
p4 = Person4()   #创建对象实例
p4.say_hi('Alice')  #调用对象实例的方法
```

程序运行结果如下

```
您好，我叫 Alice
```

静态方法（@staticmethod）

- 声明属于与类的对象实例无关的方法
- 静态方法不对特定实例进行操作，在静态方法中访问对象实例会导致错误
- 静态方法通过装饰器@staticmethod来定义

```
@staticmethod
def 静态方法名([形参列表]):
    函数体
```

- 静态方法一般通过类名来访问，也可以通过对象实例来调用

```
类名.静态方法名([实参列表])
```

【例8.11】静态方法示例（TemperatureConverter.py）

```
class TemperatureConverter:
    @staticmethod
    def c2f(t_c): #摄氏温度到华氏温度的转换
        t_c = float(t_c)
        t_f = (t_c * 9/5) + 32
        return t_f
    @staticmethod
    def f2c(t_f): #华氏温度到摄氏温度的转换
        t_f = float(t_f)
        t_c = (t_f - 32) * 5 /9
        return t_c
#测试代码
print("1. 从摄氏温度到华氏温度.")
print("2. 从华氏温度到摄氏温度.")
choice = int(input("请选择转换方向： "))
if choice == 1:
    t_c = float(input("请输入摄氏温度： "))
    t_f = TemperatureConverter.c2f(t_c)
    print("华氏温度为： {0:.2f}".format(t_f))
elif choice == 2:
    t_f = float(input("请输入华氏温度： "))
    t_c = TemperatureConverter.f2c(t_f)
    print("摄氏温度为： {0:.2f}".format(t_c))
else:
    print("无此选项，只能选择1或2！")
```

- 摄氏温度与华氏温度之间的相互转换

```
1. 从摄氏温度到华氏温度.
2. 从华氏温度到摄氏温度.
请选择转换方向：1
请输入摄氏温度： 30
华氏温度为： 86.00
```

```
1. 从摄氏温度到华氏温度.
2. 从华氏温度到摄氏温度.
请选择转换方向：2
请输入华氏温度： 70
摄氏温度为： 21.11
```

（a）从摄氏到华氏 （b）从华氏到摄氏

类方法（@classmethod）

- 允许声明属于类本身的方法，即类方法
- 类方法不对特定实例进行操作，在类方法中访问对象实例属性会导致错误
- 类方法通过装饰器@classmethod来定义，第一个形式参数必须为类对象本身，通常为cls

```
@classmethod
def 类方法名(cls,[形参列表]):
    函数体
```

- 类方法一般通过类名来访问，也可通过对象实例来调用

```
类名.类方法名([实参列表])
```

【例8.12】类方法示例

```
class Foo:
    classname = "Foo"
    def __init__(self, name):
        self.name = name
    def f1(self): #实例方法
        print(self.name)
    @staticmethod
    def f2():    #静态方法
        print("static")
    @classmethod
    def f3(cls):  #类方法
        print(cls.classname)
#测试代码
f = Foo("李")
f.f1()
Foo.f2()
Foo.f3()
```

程序运行结果

```
李
static
Foo
```

__init__方法（构造函数）和__new__方法

- __init__方法即构造函数（构造方法），用于执行类的实例的初始化工作。创建完对象后调用，初始化当前对象的实例，无返回值
- __new__方法是一个类方法，创建对象时调用，返回当前对象的一个实例，一般无需重载该方法
- 【例8.13】__init__方法示例1（PersonInit.py）

```
class Person5:          #定义类Person5
    def __init__(self, name): #__init__方法
        self.name = name #把参数name赋值给self.name，即成员变量name（域）
    def say_hi(self):    #定义类Person的方法say_hi
        print('您好, 我叫', self.name)
p5 = Person5('Helen')   #创建对象
p5.say_hi()             #调用对象的方法
```

程序运行结果如下

```
您好, 我叫 Helen
```

【例8.14】__init__方法示例2

- 定义类Point，表示平面坐标点

```
class Point:
    def __init__(self, x = 0, y = 0): #构造函数
        self.x = x
        self.y = y
p1 = Point()                    #创建对象
print("p1({0},{1})".format(p1.x, p1.y))
p1 = Point(5, 5)                #创建对象
print("p1({0},{1})".format(p1.x, p1.y))
```

程序运行结果

```
p1(0,0)
p1(5,5)
```

__del__方法（析构函数）

- __del__方法即析构函数（析构方法），用于实现销毁类的实例所需的操作，如释放对象占用的非托管资源（例如：打开的文件、网络连接等）
- 默认情况下，当对象不再被使用时，__del__方法运行，由于Python解释器实现自动垃圾回收，即无法保证这个方法究竟在什么时候运行
- 通过del语句，可以强制销毁一个对象实例，从而保证调用对象实例的__del__方法

【例8.15】__del__方法示例

```
class Person3:
    count = 0              #定义类域count，表示计数
    def __init__(self, name,age): #构造函数
        self.name = name #把参数name赋值给self.name，即成员变量name（域）
        self.age = age   #把参数age赋值给self.age，即成员变量age（域）
        Person3.count += 1  #创建一个实例时，计数加1
    def __del__(self):      #析构函数
        Person3.count -= 1  #销毁一个实例时，计数减1
    def say_hi(self):       #定义类Person3的方法say_hi()
        print('您好, 我叫', self.name)
    def get_count():        #定义类Person3的方法get_count()
        print('总计数为：', Person3.count)
print('总计数为：',Person3.count) #类名访问
p31 = Person3('张三',25)   #创建对象
p31.say_hi()       #调用对象的方法
Person3.get_count()  #通过类名访问
p32 = Person3('李四',28)   #创建对象
p32.say_hi()       #调用对象的方法
Person3.get_count()  #通过类名访问
del p31            #删除对象p31
Person3.get_count()  #通过类名访问
del p32            #删除对象p32
Person3.get_count()  #通过类名访问
```

程序运行结果如下：

```
总计数为：  0
您好, 我叫 张三
总计数为：  1
您好, 我叫 李四
总计数为：  2
总计数为：  1
总计数为：  0
```

私有方法与公有方法

- 两个下划线开头，但不以两个下划线结束的方法是私有的（private），其他为公共的（public）
- 以双下划线开始和结束的方法是Python的专有特殊方法。不能直接访问私有方法，但可以在其他方法中访问

【例8.16】私有方法示例

```
class Book:                    #定义类Book
    def __init__(self, name, author, price):
        self.name = name  #把参数name赋值给self.name，即成员变量name（域）
        self.author = author#把参数author赋值给self.author，即成员变量author（域）
        self.price = price   #把参数price赋值给self.price，即成员变量price（域）
    def __check_name(self):      #定义私有方法，判断name是否为空
        if self.name == '' : return False
        else: return True
    def get_name(self):          #定义类Book的方法get_name
        if self.__check_name():print(self.name,self.author) #调用私有方法
        else:print('No value')
b = Book('Python程序设计教程','江红',59.0)   #创建对象
b.get_name()                   #调用对象的方法
b.__check_name()               #直接调用私有方法，非法
```

```
Python 程序设计教程 江红
Traceback (most recent call last):
  File "C:\Pythonpa\ch09\BookPrivate.py", line 14, in <module>
    b.__check_name()                #直接调用私有方法，非法
AttributeError: 'Book' object has no attribute '__check_name'
```

继承

- 派生类：Python支持多重继承，即一个派生类可以继承多个基类

```
class 派生类名(基类 1,[基类 2,…]):
    类体
```

- 声明派生类时，必须在其构造函数中调用基类的构造函数

```
基类名.__init__(self, 参数列表)
```

基类	派生类
Quadrilateral	Trapezoid、Parallelogram、Rectangle、Square
Shape	Rectangle、Triangle、Circle
Degree	Doctor、Master、Bachelor

【例8.19】派生类示例

- 创建基类Person，包含两个数据成员name和age；创建派生类Student，包含一个数据成员stu_id

```
class Person:                #基类
    def __init__(self, name, age): #构造函数
        self.name = name     #姓名
        self.age = age       #年龄
    def say_hi(self):        #定义基类方法say_hi
        print('您好,                我叫{0},                {1}岁
'.format(self.name,self.age))
class Student(Person):       #派生类
    def __init__(self, name, age, stu_id): #构造函数
        Person.__init__(self, name, age) #调用基类构造函数
        self.stu_id = stu_id    #学号
    def say_hi(self):        #定义派生类方法say_hi
        Person.say_hi(self)    #调用基类方法say_hi
        print('我是学生, 我的学号为：', self.stu_id)
p1 = Person('张王一', 33)          #创建对象
p1.say_hi()
s1 = Student('李姚二', 20, '2018101001') #创建对象
s1.say_hi()
```

程序运行结果如下：

```
您好，我叫张王一，33 岁
您好，我叫李姚二，20 岁
我是学生，我的学号为：  2013101001
```

类成员的继承和重写

- 通过继承，派生类继承基类中除构造方法之外的所有成员
- 如果在派生类中重新定义从基类继承的方法，则派生类中定义的方法覆盖从基类中继承的方法
- 【例8.21】类成员的继承和重写示例

```
class Dimension:  #定义类Dimensions
    def __init__(self, x, y): #构造函数
        self.x = x       #x坐标
        self.y = y       #y坐标
    def area(self):       #基类的方法area()
        pass
class Circle(Dimension):  #定义类Circle（圆）
    def __init__(self, r): #构造函数
        Dimension.__init__(self, r, 0)
    def area(self):       #覆盖基类的方法area()
        return 3.14 * self.x * self.x   #计算圆面积
class Rectangle(Dimension):  #定义类Rectangle（矩形）
    def __init__(self, w, h): #构造函数
        Dimension.__init__(self, w, h)
    def area(self):  #覆盖基类的方法area()
        return self.x * self.y  #计算矩形面积
d1 = Circle(2.0)         #创建对象：圆
d2 = Rectangle(2.0, 4.0)  #创建对象：矩形
print(d1.area(), d2.area())  #计算并打印圆和矩形面积
```

程序运行结果如下

```
12.56 8.0
```

对象的特殊方法

- 包含许多以双下划线开始和结束的方法，称之为特殊方法
- 例如，创建对象实例时自动调用其__init__方法，a<b时，自动调用对象a的__lt__方法
- 表9-2 Python特殊方法

特殊方法	含义
__lt__、__add__等	对应运算符<、+等
__init__、__del__	创建或销毁对象时调用
__len__	对应于内置函数 len()
__setitem__、__getitem__	按索引赋值、取值
__repr__(self)	对应于内置函数 repr()
__str__(self)	对应于内置函数 str()
__bytes__(self)	对应于内置函数 bytes()
__format__(self, format_spec)	对应于内置函数 format()
__bool__(self)	对应于内置函数 bool()
__hash__(self)	对应于内置函数 hash()
__dir__(self)	对应于内置函数 dir()

【例8.22】对象的特殊方法示例

```
class Person:
    def __init__(self, name, age): #特殊方法（构造函数）
        self.name = name
        self.age = age
    def __str__(self):          #特殊方法，输出成员变量
        return '{0}, {1}'.format(self.name,self.age)
#测试代码
p1 = Person('张三', 23)
print(p1)
```

程序运行结果如下：

张三, 23

运算符重载与对象的特殊方法

- Python的运算符实际上是通过调用对象的特殊方法实现的

```
>>> x=12; y=23
>>> x+y                #等价于调用x.__add__(y)。输出： 35
35
>>> x.__add__(y)       #输出： 35
35
```

运算符	特殊方法	含义
<，<=，==， >，>=，!=，	__lt__、__le__、__eq__、 __gt__、__ge__、__ne__	比较运算符
|，^，&	__or__、__ror__、__xor__、__rxor__、__and__、__rand__	按位或、异或、与
|=，^=，&=	__ior__、__ixor__、__iand__	按位复合赋值运算
<<，>>	__lshift__,__rlshift__、__rshift__,__rrshift__	移位运算
<<=，>>=	__ilshift__,__irlshift__、__irshift__,__irrshift__	移位复合赋值运算
+，-	__add__、__radd__、__sub__、__rsub__	加法与减法
+=，-=	__iaddr__、__isub__	加减复合赋值运算
*，/， %，//	__mul__、__rmul__、__truedivv__、__rtruedivv__、 __mod__、__rmod__、__floordiv__、__rfloordiv__	乘法、除法、取余 整数除法
*=，/=， %=，//=	__imul__、__idiv__、__itruedivv__、__imod__、 __ifloordiv__	乘除复合赋值运算
+x，-x	__pos__、__neg__	正负号
~x	__invert__	按位翻转
，=	__pow__、__rpow__、__ipow__	指数运算

【例8.23】运算符重载示例

程序运行结果如下

```
3 4 5 6 7
2 3 4 5 6
8 12 16 20 24
4.0 6.0 8.0 10.0 12.0
5
```

```
class MyList:           #定义类MyList
    def __init__(self, *args): #构造函数
        self.__mylist = []  #初始化私有属性，空列表
        for arg in args:
            self.__mylist.append(arg)
    def __add__(self, n):   #重载运算符"+"，每个元素增加n
        for i in range(0, len(self.__mylist)):
            self.__mylist[i] += n
    def __sub__(self, n):   #重载运算符"-"，每个元素减少n
        for i in range(0, len(self.__mylist)):
            self.__mylist[i] -= n
    def __mul__(self, n):   #重载运算符"*"，每个元素乘以n
        for i in range(0, len(self.__mylist)):
            self.__mylist[i] *= n
    def __truediv__(self, n): #重载运算符"/"，每个元素除以n
        for i in range(0, len(self.__mylist)):
            self.__mylist[i] /= n
    def __len__(self):    #对应于内置函数len()，返回列表长度
        return(len(self.__mylist))
    def __repr__(self):   #对应于内置函数str()，显示列表
        str1 = ''
        for i in range(0, len(self.__mylist)):
            str1 += str(self.__mylist[i]) + ' '
        return str1
#测试代码
m = MyList(1, 2, 3, 4, 5) #创建对象
m + 2; print(repr(m))   #每个元素加2
m - 1; print(repr(m))   #每个元素减1
m * 4; print(repr(m))   #每个元素乘4
m / 2; print(repr(m))   #每个元素除2
print(len(m))       #列表长度
```

@functools.total_ordering装饰器

- 支持大小比较的对象需要实现特殊方法：__eq__、__lt__、__le__、__ge__、__gt__
- 使用functools模块的total_ordering装饰器装饰类，则只需要实现__eq__，以及__lt__、__le__、__ge__、__gt__中的任意一个
- total_ordering装饰器实现其他比较运算，以简化代码量

【例8.24】total_ordering装饰器函数示例

```
import functools
@functools.total_ordering
class Student:
    def __init__(self, firstname, lastname):  #姓和名
        self.firstname = firstname
        self.lastname = lastname
    def __eq__(self, other):     #判断姓名是否一致
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))
    def __lt__(self, other):     #self姓名<other姓名
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))
#测试代码
if __name__ == '__main__':
    s1 = Student('Mary','Clinton')
    s2 = Student('Mary','Clinton')
    s3 = Student('Charlie','Clinton')
    print(s1==s2)
    print(s1>s3)
```

程序运行结果如下：

True

True

__call__方法和可调用对象（callabe）

- Python类体中可以定义一个特殊的方法：__call__方法
- 定义了__call__方法的对象称之为可调用对象（callabe），即该对象可以像函数一样被调用
- 【例8.25】可调用对象示例

```
class GDistance:      #类：自由落体距离
    def __init__(self, g): #构造函数
        self.g = g
    def __call__(self, t): #自由落体下落距离
        return (self.g*t**2)/2
#测试代码
if __name__ == '__main__':
    e_gdist = GDistance(9.8) #地球上的重力加速度
    for t in range(11):     #自由落体0~10秒的下落距离
        print(format(e_gdist(t), "0.2f"),end=' ') #调用可调用对象e_gdist
```

程序运行结果如下：

0.00 4.90 19.60 44.10 78.40 122.50 176.40 240.10 313.60 396.90 490.00

对象的引用、浅拷贝和深拷贝

- 对象的引用：对象的赋值
- 【例8.26】对象的引用示例。若银行卡采用列表[户主名，[卡种别，金额]]表示，则：

```
>>> acc10=['Charlie', ['credit', 0.0]] #创建列表对象（信用卡账户），变量acc10代表主卡
>>> acc11=acc10                        #变量acc11代表副卡，指向acc10（主卡）的对象
>>> id(acc10),id(acc11)                #二者id相同，输出：
(2739033039112, 2739033039112)
```

- 对象的浅拷贝
  - 对象的赋值引用同一个对象，即不拷贝对象
    - 切片操作。例如，acc11[:]。
    - 对象实例化。例如，list(acc11)。
    - copy模块的copy函数。例如，copy.copy(acc1)。

【例8.27】对象的浅拷贝示例

```
>>> import copy
>>> acc1=['Charlie', ['credit', 0.0]]
>>> acc2=acc1[:]        #使用切片方式拷贝对象
>>> acc3=list(acc1)     #使用对象实例化方法拷贝对象
>>> acc4=copy.copy(acc1) #使用copy.copy函数拷贝对象
>>> id(acc1),id(acc2),id(acc3),id(acc4) #拷贝对象id各不相同
(2739033039240, 2739033040008, 2739035724168, 2739033039880)
>>> acc2[0]='Mary'  #acc2的第一个元素赋值，即户主为'Mary'
>>> acc2[1][1]=-99.9 #acc2的第二个元素的第二个元素赋值，即消费金额99.9
>>> acc1, acc2      #注意，acc2消费金额改变99.9，acc1也随之改变
(['Charlie', ['credit', -99.9]], ['Mary', ['credit', -99.9]])
>>> id(acc1[1]),id(acc2[1]) # acc1[1]和acc2[1]指向同一个对象
(2739033038152, 2739033038152)
```

对象的深拷贝

- 使用copy模块的deepcopy函数，拷贝对象中包含的子对象
- 【例8.28】对象的深拷贝示例

```
>>> import copy
>>> acc1=['Charlie', ['credit', 0.0]]
>>> acc5=copy.deepcopy(acc1) #使用copy.deepcopy函数深拷贝对象
>>> acc5[0]='Clinton' #acc5的第1个元素赋值，即户主为'Clinton'
>>> acc5[1][1]=-19.9  #acc5的第2个元素的第2个元素赋值，即消费金额19.9
>>> acc1,acc5
(['Charlie', ['credit', 0.0]], ['Clinton', ['credit', -19.9]])
>>> id(acc1),id(acc5),id(acc1[1]),id(acc5[1])
(2739033040648, 2739033040264, 2739033040520, 2739033039688)
```

可迭代对象：迭代器和生成器

- 可循环迭代的对象称之为可迭代对象，迭代器和生成器函数是可迭代对象，Python提供了定义迭代器和生成器的协议和方法
- 相对于序列，可迭代对象仅在迭代时产生数据，故可节省内存空间。Python语言提供了若干内置可迭代对象：range、map、filter、enumerate、zip；标准库itertools模块中包含各种迭代器。这些迭代器非常高效，且内存消耗小

直方图（Histogram）

Histogram 类封装直方图（包括数据及基本统计功能）。Histogram 类的设计思路如下：

（1）定义带一个整数参数 n 的构造函数，用于初始化存储数据的列表，列表长度为 n，列表各元素初始值为 0。

（2）定义实例对象方法 addDataPoint(self,i)，用于增加一个数据点。

（3）定义用于计算数据点个数之和、平均值、最大值、最小值的实例对象方法：count()、mean()、max()、min()。

（4）定义用于绘制简单直方图的实例对象方法：draw()。

```
import random
import math
class Stat:
    def __init__(self, n):
        self._data = []
        for i in range(n):
            self._data.append(0)
    def addDataPoint(self, i):
        """增加数据点"""
        self._data[i] += 1
    def count(self):
        """计算数据点个数之和（统计数据点个数）"""
        return sum(self._data)
    def mean(self):
        """计算各数据点个数的平均值"""
        return sum(self._data)/len(self._data)
    def max(self):
        """计算各数据点个数的最大值"""
        return max(self._data)
    def min(self):
        """计算各数据点个数的最小值"""
        return min(self._data)
    def draw(self):
        """绘制简易直方图"""
        for i in self._data:
            print('#'* i)
#测试代码
if __name__ == '__main__':
    #随机生成100个的0到9的数
    st = Stat(10)
    for i in range(100):
        score = random.randrange(0,10)
        st.addDataPoint(math.floor(score))
    print('数据点个数：{}'.format(st.count()))
    print('数据点个数的平均值：{}'.format(st.mean()))
    print('数据点个数的最大值：{}'.format(st.max()))
    print('数据点个数的最小值：{}'.format(st.min()))
    st.draw()      #绘制简易直方图
```

```
数据点个数：100
数据点个数的平均值：10.0
数据点个数的最大值：17
数据点个数的最小值：5
########
###########
#########
########
#######
###############
#################
#####
################
######
```