第八章：面向对象与类

Table of Contents

[1. 序言 2](#_Toc52897011)

[1.1 章节简介 2](#_Toc52897012)

[2. 类和实例 3](#_Toc52897013)

[2.1 数据的封装 4](#_Toc52897014)

[3. 访问限制 5](#_Toc52897015)

[4. 继承和多态 8](#_Toc52897016)

[4.1 继承 8](#_Toc52897017)

[4.2 多态 10](#_Toc52897018)

[4.3 Python中的鸭子类型(扩展) 12](#_Toc52897019)

[5. 获取对象信息 13](#_Toc52897020)

[5.1 type() 13](#_Toc52897021)

[5.2 isinstance() 14](#_Toc52897022)

[5.3 dir() 14](#_Toc52897023)

[5.4 获取属性的方法 15](#_Toc52897024)

[6. 实例属性和类属性 16](#_Toc52897025)

[7. 使用\_\_slots\_\_ 16](#_Toc52897026)

[8. 使用@property 19](#_Toc52897027)

[9. 多重集成 21](#_Toc52897028)

[10. 定制类 23](#_Toc52897029)

[10.1 \_\_str\_\_() 23](#_Toc52897030)

[10.2 \_\_iter\_\_ () 24](#_Toc52897031)

[10.3 \_\_getitem\_\_ () 25](#_Toc52897032)

[10.4 \_\_call\_\_ () 26](#_Toc52897033)

[11. 枚举类 28](#_Toc52897034)

# 序言

## 章节简介

Python从设计之初就已经是一门面向对象的语言，正因为如此，在Python中创建一个类和对象是很容易的。本章节我们将详细介绍Python的面向对象编程。如果你以前没有接触过面向对象的编程语言，那你可能需要先了解一些面向对象语言的一些基本特征，在头脑里头形成一个基本的面向对象的概念，这样有助于你更容易的学习Python的面向对象编程。面向对象的语言遵循的编程模式不同于结构化编程语言，比如 C 和 COBOL。结构化编程范例是高度面向数据的：您拥有数据结构，然后程序指令处理该数据。面向对象的语言（比如Python、Java等 语言）将数据和程序指令组合到对象中。

对象是一个自成一体的实体，它仅包含属性和行为，不含任何其他内容。不需要拥有包含字段（属性）的数据结构并将该结构传递到处理它（行为）的所有程序逻辑，在面向对象的语言中，数据和程序逻辑组合在一起。本章会介绍Python面向编程的众多细节，课程目标为：

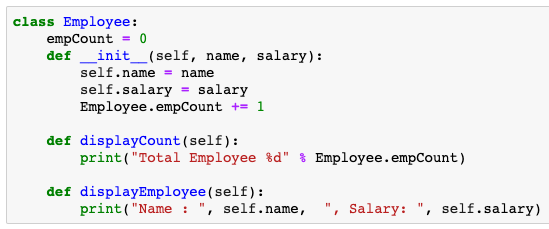
* 类和实例
* 访问限制
* 继承和多态
* 获取对象信息
* 实例属性和类属性
* 使用\_\_slots\_\_
* 使用@property
* 多重集成
* 定制类
* 枚举类

# 类和实例

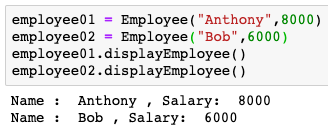
面向对象编程里最重要的就是类和对象的概念。类是抽象的模板，实例是根据类创建出来的一个个具体的"对象"。例如：动物是一大类，猫、狗、鱼等都是特定的对象，即实例。类名一般使用大驼峰来进行命名，尽量见名知意。python采用多继承的机制，一个类可以同时继承多个父类(基类、超类)，继承的父类有先后顺序，写在类名后面的括号中。Python中，object是Python3中所有类的基类中，可以省略类名后面的括号，表示隐式继承。Python中使用class关键字来定义类，基本结构如下。使用class语句来创建一个新类，class之后为类的名称并以冒号结尾。类由类成员，方法，数据属性组成:



以下是一个简单的 Python 类的例子, empCount 变量是一个类变量，它的作用是用来记录我们一共生成了多少个Employee对象，它的值将在这个类的所有实例之间共享。你可以在内部类或外部类使用 Employee.empCount 访问。第一种方法\_\_init\_\_()方法是一种特殊的方法，被称为类的构造函数或初始化方法，当创建了这个类的实例时就会调用该方法。self 代表类的实例，self 在定义类的方法时是必须有的，虽然在调用时不必传入相应的参数:



注意到\_\_init\_\_方法的第一个参数永远是self，表示创建的实例本身，因此，在\_\_init\_\_方法内部，就可以把各种属性绑定到self，因为self就指向创建的实例本身。有了\_\_init\_\_方法，在创建实例的时候，就不能传入空的参数了，必须传入与\_\_init\_\_方法匹配的参数，但self不需要传，Python解释器自己会把实例变量传进去：



在上面的代码中，我们创建了2个Employee类的对象实例，并把第一个对象赋给变量employee01，第二个对象赋值给变量employee02。通过调用\_\_init\_\_函数，我们将2这个对象的name属性和salary属性分别赋值为(”Anthony”, 8000)和(“Bob”, 6000)。当我们第二三行代码调用对象的displayEmployee()方法时，该方法分别打印两个对象的属性值。

## 数据的封装

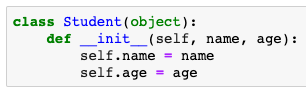
面向对象编程的一个重要特点就是数据封装。在上面的Employee类中，每个实例就拥有各自的name和salary这些数据。我们可以通过函数来访问这些数据，比如打印一个Employee的工资。既然Employee实例本身就拥有这些数据，要访问这些数据，就没有必要从外面的函数去访问，可以直接在Student类的内部定义访问数据的函数，这样，就把“数据”给封装起来了。这些封装数据的函数是和Employee类本身是关联起来的，我们称之为类的方法。要定义一个方法，除了第一个参数是self外，其他和普通函数一样。要调用一个方法，只需要在实例变量上直接调用，除了self不用传递，其他参数正常传入：



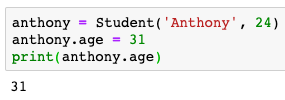
这样一来，我们从外部看Employee类，就只需要知道，创建实例需要给出name和salary，而如何输出这些信息，都是在Employee类的内部定义的，这些数据和逻辑被“封装”起来了，调用很容易，但却不用知道内部实现的细节。这就是类的封装。

# 访问限制

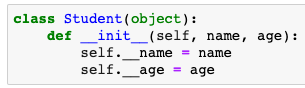
我们先来看一个类的定义：



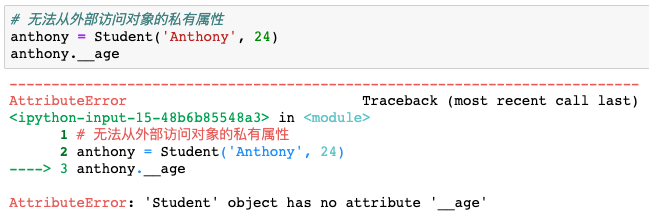
对于由这个类生成的对象，外部代码可以自由地修改实例的属性数据。



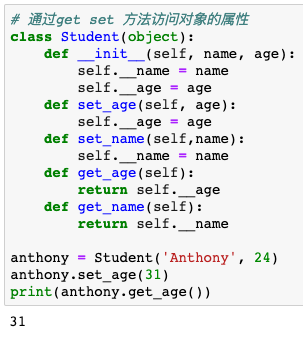
若想让内部数据不被外部访问，可以在属性名称前加两个下划线\_\_，这样就变成了一个私有变量，内部可以访问，对象的外部代码不能访问。



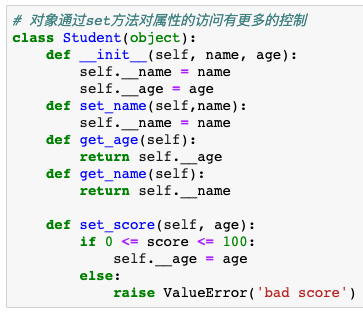
对于这个改动后的类，如果外部想访问生成的对象的\_\_name和\_\_age属性时，代码就会报错。通过这样的改动，外部代码由于不能任意修改对象的属性和状态，这种访问机制的限制使得代码更加健壮：



那对于这种私有属性，如果我想访问对象的私有属性应该如何操作呢？答案是在类定义中为某个属性增加get方法和set方法，如下示例所示：



如果可以直接通过对象加属性名的方式访问属性，为什么还要通过get和set方法这么麻烦的操作对象呢？比如在set方法中，我们可以做参数的检查，如果外部函数给属性赋值时输入和错误的值，对象自己可以进行判断，对自己的属性有更多的掌控，从而增强代码健壮性：



在Python的使用习惯里，变量名类似\_\_xxx\_\_这种的，以双下划线开头双下划线结尾的，我们认为特殊变量，特殊变量是可以直接访问的，不是private变量，所以，不能用\_\_name\_\_、\_\_score\_\_这样的变量名。我们在写代码时候，如果看到以一个下划线开头的实例变量名，比如\_name，这样的对象的属性外部是可以访问的。但是按照约定俗成的习惯，看到这样的变量时，意思就是“虽然我可以被访问，但是，请把我视为私有变量，不要随意访问”。简单来说就是，外部代码不要访问对象里以双下划线或者一个下划线开头命名的属性。

# 继承和多态

## 继承

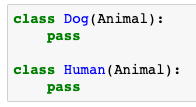
在面向对象编程里，不只是Python很多其他高级语言的类都有继承和多态的特性。所谓的继承，是一种创建新类的机制，目的是专门使用或者修改现有类的行为。原始的类(被继承的类)叫做父类，基类或超类(base class, super class)。新生成的类就叫做子类或派生类。通过继承父类，新创建的子类将继承父类定义的所有属性。同时，子类可以重新定义任何这些属性并添加自己的新属性。

在class语句中使用逗号分隔的父类名称列表来指定继承。如果没有写明父类，默认所有的类都继承自ojbect，就像前面的例子，object是所有Python类的根类，提供了一些常见方法(如\_\_str\_\_()，它可以创建在打印中使用的字符串)的默认实现。

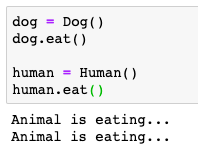
来看一个例子，我们先来定义一个Animal类作为父类，这个类里有一个eat()方法。



我们可以再定义两个类，Human和Dog类，二者因为都是动物，都有吃的功能，都继承了父类Animal。那么这里Human和Dog就是Aminal的子类，Animal就是Human和Dog类的父类。

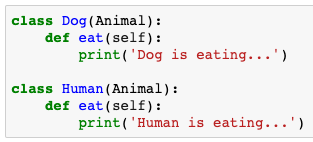


类继承机制最大的好处就是父类里全部的属性和函数，子类全部都可以拿来使用。Human本身有吃饭的功能，Dog也有吃饭的功能，我们避免了在每一个子类里都重复的定义一个eat()方法，而只要继承实现了eat()方法的父类Animal，那么这两个子类就自动的有了eat()这个方法。

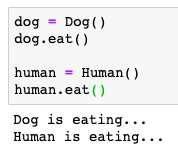


## 多态

我们继续对两个子类改进，虽然Human和Dog都会吃饭，但是吃的动作还是有很大区别，我们对eat()做些改动，让他们分别表示这个子类特有的行为。先将两个子类改进如下：



再次生成两个子类的对象，看看结果：

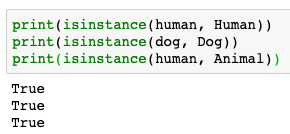


当父类和子类都有同一个方法时，我们说子类的方法会覆盖父类的方法的实现。在代码运行的时候，总是会调用子类的eat()。这也是类的第二个特征：多态。

我们继续介绍，加深对多态的理解，其实我们在定义一个类时，定义的是一个新的数据类型，这个新定义的类是一种类型，就像Python的一些基本数据类型int，str，list一样。



还记得我们可以用isinstance()方法判断一个变量的类型？这里我们分别检验。前两行代码分别检验human变量是否Human类型，dog变量是否Dog类型，结果都返回True没有问题。注意第三行，human变量是否为Animal类型也返回True!这就是继承的特点，虽然human变量是一个Human对象，但由于Human本身也继承了父类Animal，human本身有Animal的所有特性，所以这里human变量也是一个Animal类型。在继承关系中，如果一个变量的数据类型是某个子类，那它的数据类型也可以被看做是这个子类的父类：



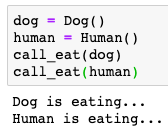
但反过来说缺不正确，我们可以生成一个对象，它的类型是Animal，但我们不能说这个对象的类型也是Human。



我们再看一个例子，可以更好地理解多态这个特性，我们定义一个方法，方法的参数是Animal类型的对象，函数里会调用参数的eat()方法：



我们调用几次这个方法看看结果：



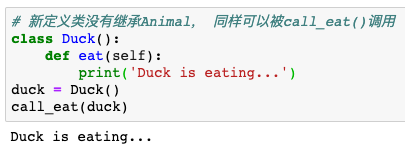
不同的实例的eat方法返回不同值，这就是多态的好处。当我们需要传入Dog、Human时，我们只需要接收Animal类型就可以了，因为Dog、Human都是Animal类型，在方法内部，不管是哪种类型的实例，我们统统按照按照Animal类型进行操作。因为Animal类型有eat()方法，因此，传入的任意类型，只要是Animal类或者子类，就会自动调用实际类型的run()方法，这就是多态的意思.而具体调用的run()方法是作用在Animal、Dog还是Human对象上，由运行时该对象的确切类型决定。这就是多态的好处：调用方只管调用，不管细节，而当我们新增一种Animal的子类时，只要确保run()方法编写正确，不用管原来的代码是如何调用的。

## Python中的鸭子类型(扩展)

这一小节介绍duck typing(鸭子类型)，大家主要目的理解：是一种动态类型的风格。一个对象有效的语义，不是由继承自特定的类或实现特定的接口，而是由当前方法和属性的集合决定。这个概念的名字来源于由James Whitcomb Riley提出的鸭子测试，“鸭子测试”可以这样表述：“当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。”

Python的“file-like object“就是一种鸭子类型。对真正的文件对象，它有一个read()方法，返回其内容。但是，许多对象，只要有read()方法，都被视为“file-like object“。许多函数接收的参数就是“file-like object“，你不一定要传入真正的文件对象，完全可以传入任何实现了read()方法的对象。

在本小节例子里，call\_eat()方法会调用传入对象的eat方法，实际上，传入的参数不必是Animal类型对象或者继承自Animal的子类的对象，只要该变量有eat()方法程序就可以正常运行，而不关心变量的类型(一个对象只要有eat方法，就认为你是”鸭子”)。

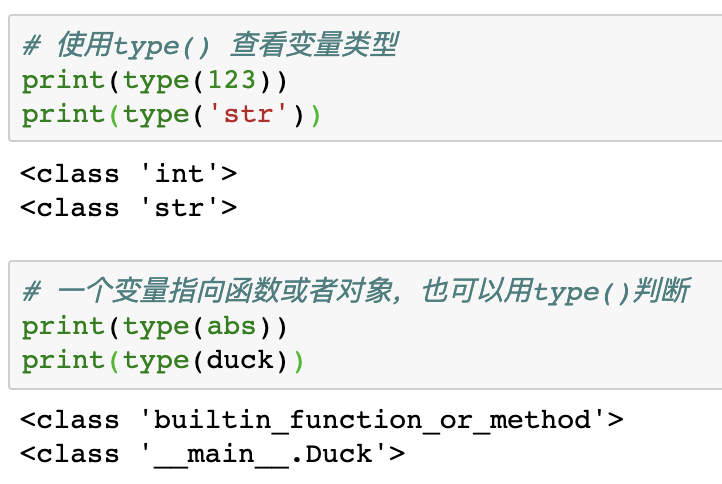


# 获取对象信息

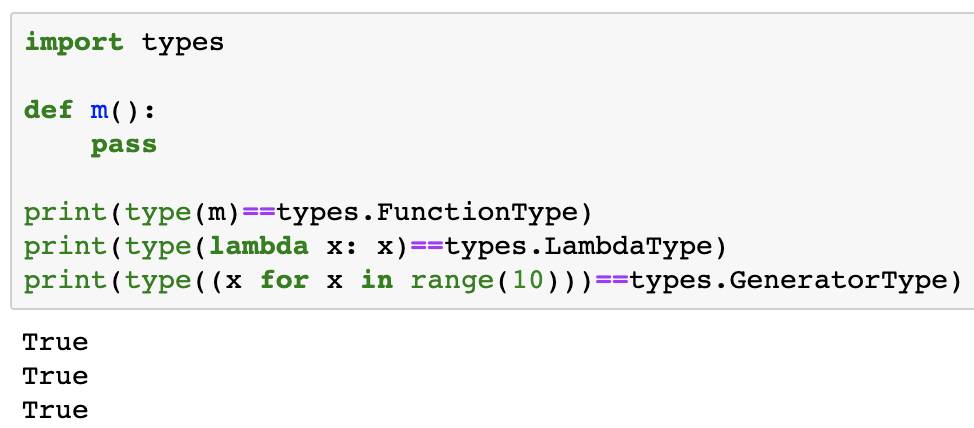
这一小节我们来看看，当我们得到一个对象的引用时，我们有什么方法查看这个对象及它对应的类有什么属性和函数。

## type()

我们可以通过type()函数查看对象的类型，这个对象不仅可以是Python的基础类型，也可以是我们定义的类的对象，还可以是函数。

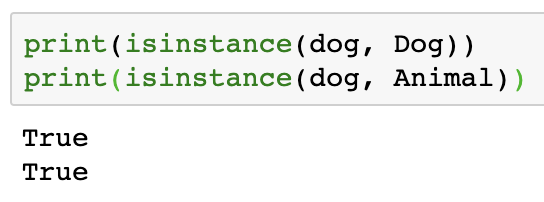


我们也可以判断一个变量是否是函数，通过types定义的常量查看它的类型。



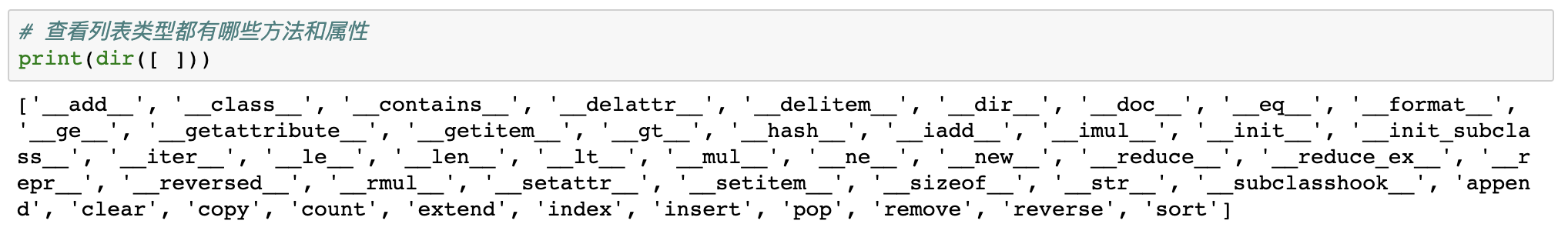
## isinstance()

我们在之前的学习中已经见过，也可以使用instance()函数判断一个变量是否为某各类。比如dog对象生成于类Dog,但因为Dog类同时继承于Animal类，所以dog对象也是Animal类型。



## dir()

带参数时，返回参数的属性、方法列表。如果参数包含方法\_\_dir\_\_()，该方法将被调用。如果参数不包含\_\_dir\_\_()，该方法将最大限度地收集参数信息：



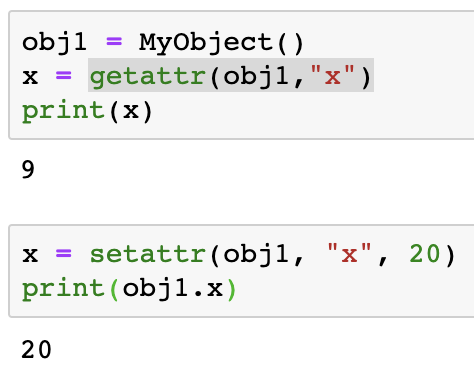
比如在上面的返回里，我们可以看到list对象的所有特殊方法和一般方法，像方法append()我们在之前已经介绍过，追加一个新的元素到list的尾部。

## 获取属性的方法

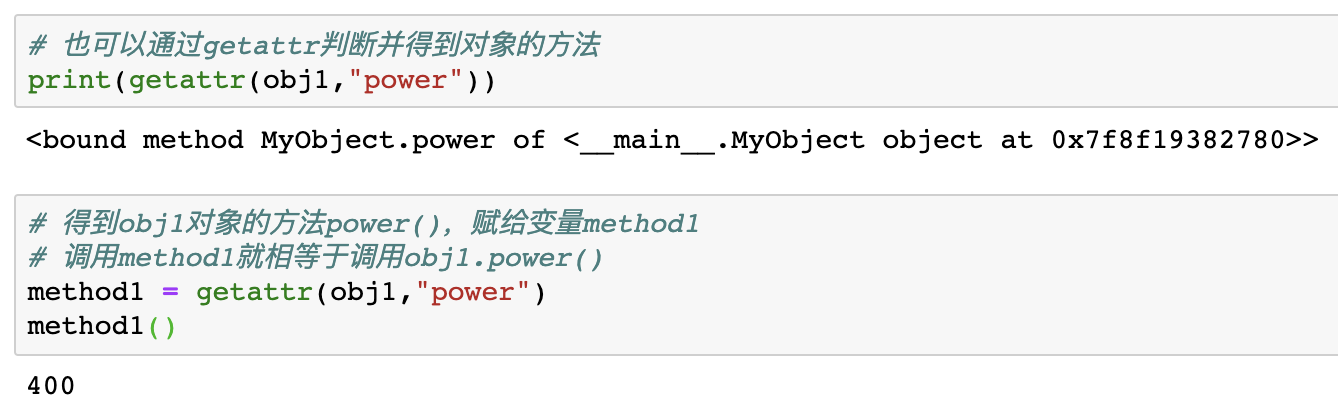
我们还有三个方法：getattr()、setattr()以及hasattr()，我们可以通过这些直接操作一个对象的状态(即针对某个对象直接查看编辑判断是否存在某个属性)：



前面使用到的Student类的对象，有name和age两个属性，我们可以通过getattr和setattr两个方法对属性进行查看或设置。实际使用中注意，只有在不知道对象信息的时候，我们才会去用这种方式查看编辑对象信息，如果很明确的知道一个变量有哪些属性，我们就用之前介绍过的变量的get()和set()方法对变量进行操作。



我们也可以通过getattr，setattr获得对象的函数并进行操作：



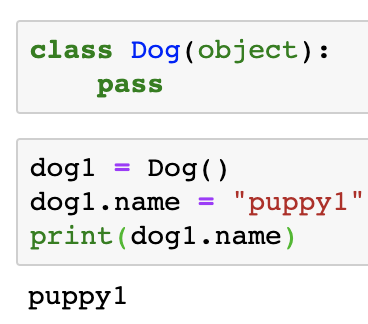
# 实例属性和类属性

我们经常获取并编辑一个对象的属性，如果类本身需要绑定一个属性呢？就是这个属性不属于某个对象而属于这个类本身。可以直接在class中定义属性，这种属性是类属性，归类所有，下面的示例我们定义一个类Student，它有一个name类属性，我们可以通过类访问，同时也可以通过生成的对象去访问这个类属性：

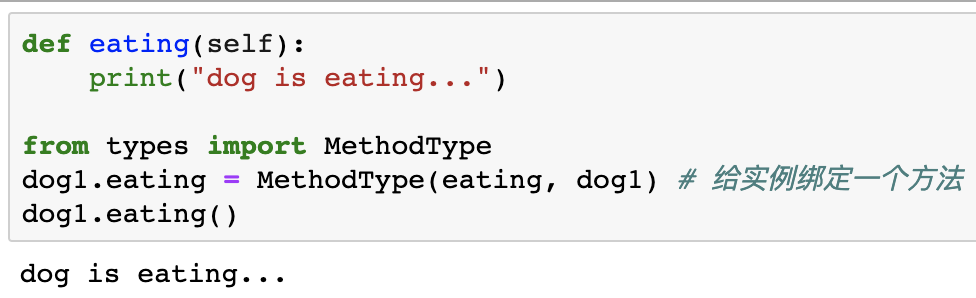


# 使用\_\_slots\_\_

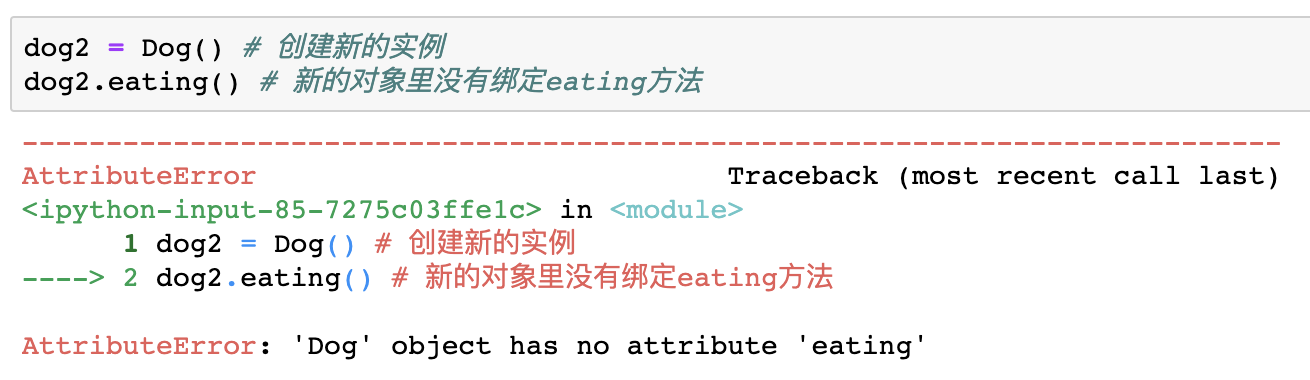
在Python编程中，当我们定义了一个class，创建了一个class的实例后，在程序里我们可以给该实例添加任何属性和方法，这就是动态语言的灵活性。在下面的实例中，我们先定义了一个Dog类,然后在一个Dog实例里为对象添加一个新的属性并访问：



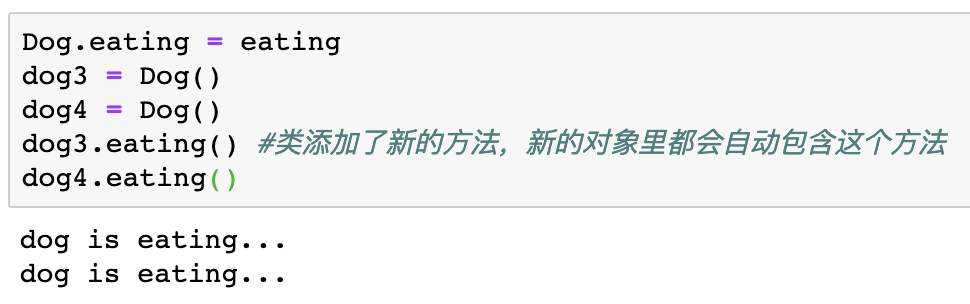
我们还可以动态在生成的对象里添加方法：



我们动态给一个对象添加的属性或方法，在另一个对象里是没有的，当访问另外一个对象的不存在的方法时程序会报错：



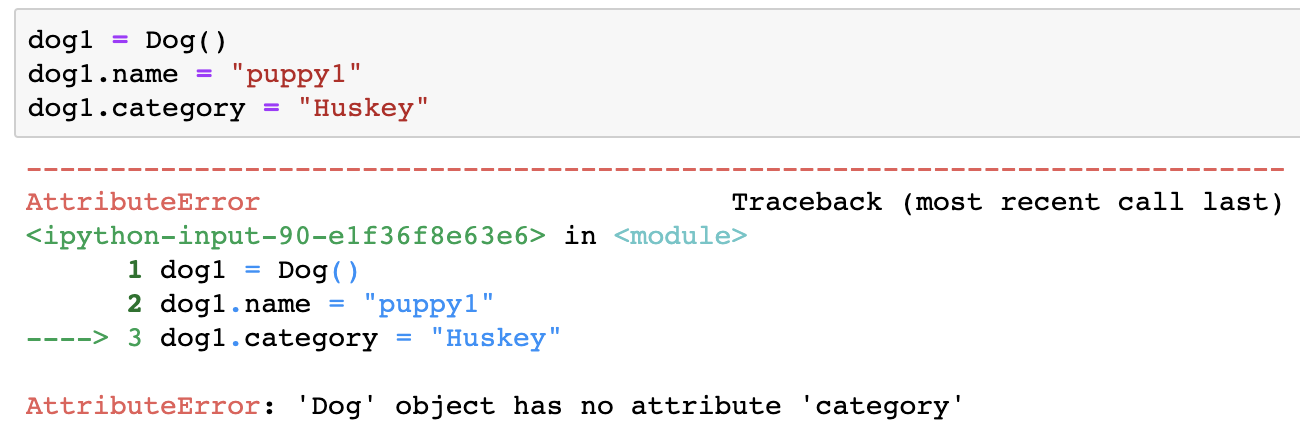
如果我们想在类定义之外给这个类的对象添加额外的方法呢，我们也可以给类动态的添加方法或属性，这样生成的新的对象里，都会自动包含动态添加的属性和方法：



动态添加属性在给程序带来方便的同时，也增加了代码的风险和不可控，如果我们想限制动态的添加，比如只允许对Dog实例添加name和age属性。为了实现这个，我们可以使用Python的\_\_slots\_\_变量，该变量的作用是限制类对象能添加的属性，来看下面的示例，我们为Dog类添加\_\_slots\_\_变量：



我们给新的Dog类的对象添加属性，看看效果。因为我们用\_\_slots\_\_规定了可添加的属性名称，所以当代码为dog1对象添加属性category时，程序会报错：

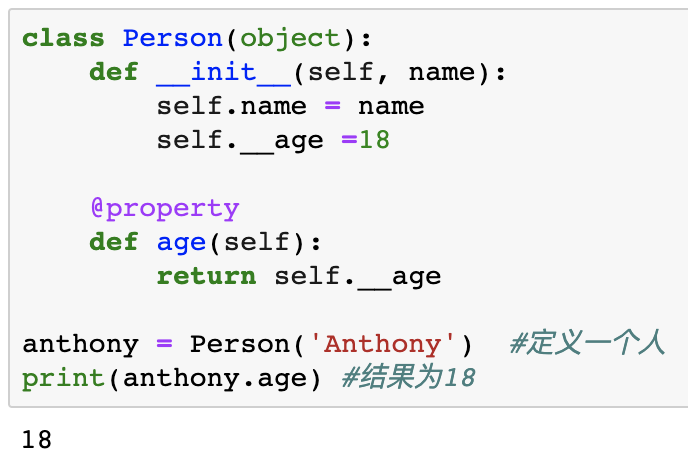


\_\_slots\_\_只会限定当前类的可以添加的属性，对于继承的子类没有限制，如果我们同样要对子类加以限制可以添加的属性名称，就要同样在子类里定义\_\_slots\_\_。在实际使用中，\_\_slots\_\_的作用其实不是当做一种安全的特性来实现。Python的这个特性实际是对内存和执行速度的一种性能优化。使用\_\_slots\_\_的类的实例不再使用字典来存储实例数据。相反，会使用基于数组的更加紧凑的数据结构。在需要大量创建对象的程序里，使用\_\_slots\_\_会显著的减少内存的占用和执行时间。

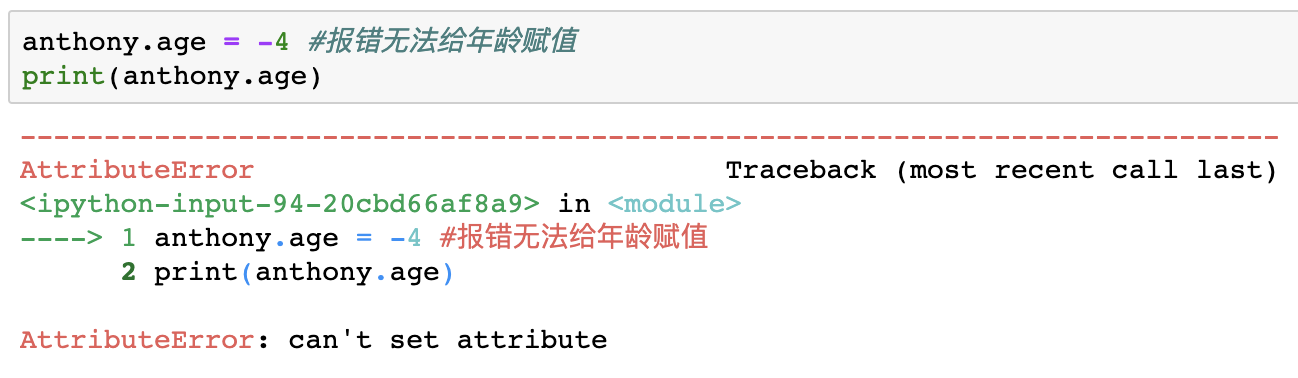
# 使用@property

在我们操作类的对象时，既要保护类的封装特性，又要让开发者可以使用“对象.属性”的方式操作操作类属性，Python提供了@property装饰器(decorator)。通过@property 装饰器，可以直接通过方法名来访问方法，不需要在方法名后添加一对小括号。Python的@property是一个非常好用的内置的装饰器。@property最大的好处就是在类中把一个方法变成属性调用，起到既能检查属性，还能用属性的方式来访问该属性的作用。

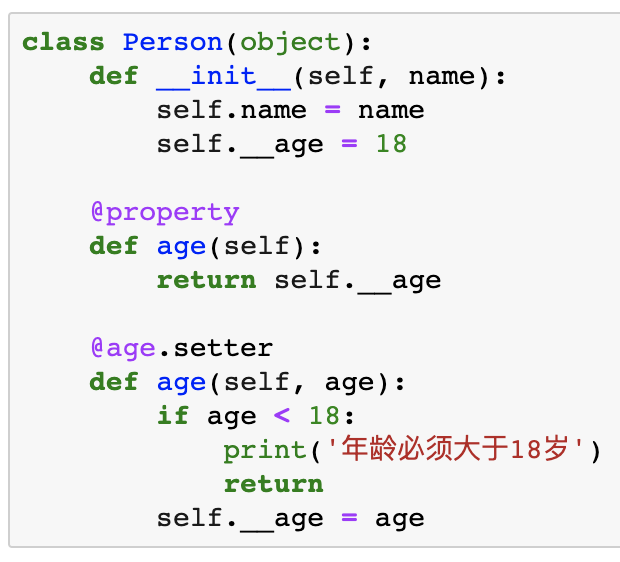
我们来看下@property装饰器的应用。其功能1是可定义只读属性。我们来定义类Person，具有年龄和姓名，要求初始化生成对象时年龄必须等于18。



在python中定义只读属性非@property莫属，如果细心留意大部分源码，都跑不了@property的身影。而定义只读属性也很简单：以需要定义的属性为方法名(上例age属性定义为方法)，其上装饰内置装饰器@property就可以了。当程序企图给对象的age属性赋值时，因为属性只读程序会报错。



@property真正强大的是可以限制属性的定义。我们定义类时，希望其中的属性必须符合实际，但因为在\_\_init\_\_里定义的属性可以随意的修改，导致很难实现。如我想实现Person类，规定每个人(即创建的实例)的年龄必须大于18岁，正常实现的话，则必须将属性age设为只读属性，然后通过方法来赋值。而通过@property，则可以一定程度上简化我们的代码。将set\_age修改为age，并且在上方加入装饰器@age.setter。这就是@property定义可访问属性的语法，即仍旧以属性名为方法名，并在方法名上增加@属性.setter就行了。更改后的Person类定义如下：

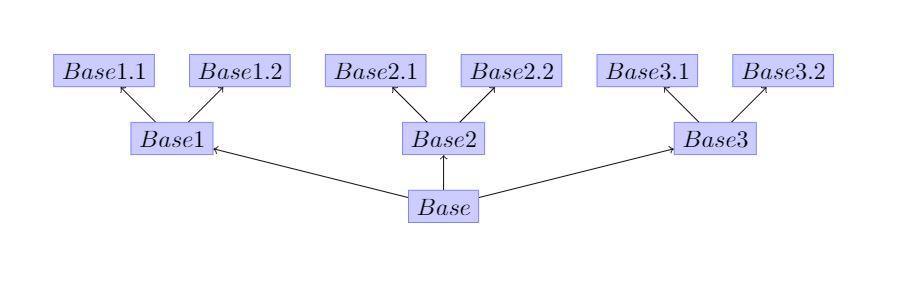


再次访问生成的对象的属性age时，我们就可以使用像a.age = 20这样的代码了，装饰器@age.setter，负责把一个setter方法变成属性赋值。它看起来像对一个属性赋值，实际上它调用的是age()方法，并在方法里对输入的age做了检验。

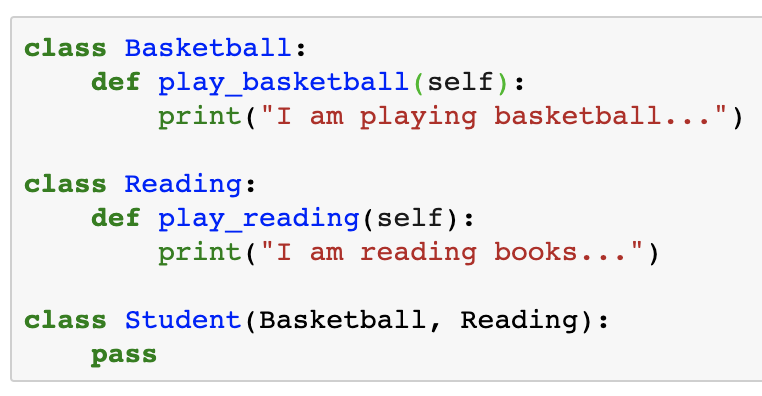


# 多重集成

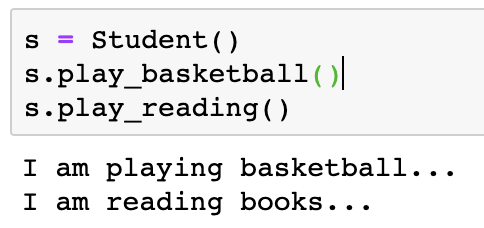
继承是面向对象编程的一个重要的方式，通过继承，子类就可以扩展父类的功能。在Python中是支持多重继承的，也就是我们可以为一个类同时指定多个父类。可以在类名的()后边添加多个类，来实现多重继承。多重继承会使子类同时拥有多个父类，并且会获取到所有父类中的方法。这叫做python的多重继承(Multiple Inheritance)。当然，子类所继承的所有父类同样也能有自己的父类，这样就可以得到一个继承关系机构图如下图所示：



我们来看一个例子，比如有2个类Basketball和Reading。我们希望实现一个类Student，这个S类的对象即会打篮球也会读书，那么我们可以定义一个类Student，让它同时继承Basketball类和Reading类。



当我们生成Student类的对象s时，s会自动同时继承Basketball和Reading两个类的所有属性和方法：



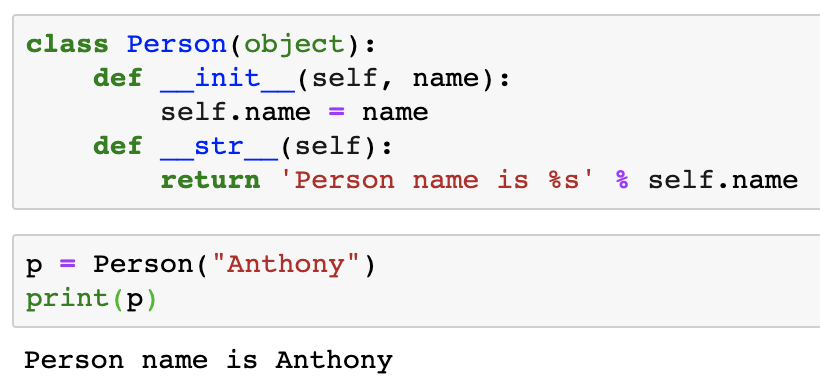
多重继承有诸如钻石继承等问题(多个父类都有同一方法，调用时的有限顺序)，很多编程语言中并没有多重继承的概念，它带来的麻烦也不少，对多重继承适合的使用场景还是要谨慎。

# 定制类

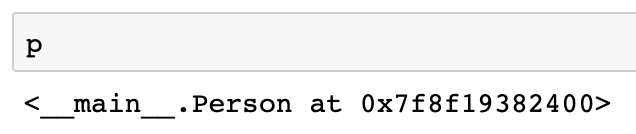
我们在前面的学习中已经看到，\_\_slots\_\_规范了类中只能使用的属性，\_\_init\_\_()方法的实现用于在生成类的对象时调用。实际上Python里还有很多这类特殊用途的变量或方法，我们也把他们叫做类里的特殊方法(special-method)，本小节介绍一些常用的特殊方法的用途和示例，来完成我们对类的定制。

## \_\_str\_\_()

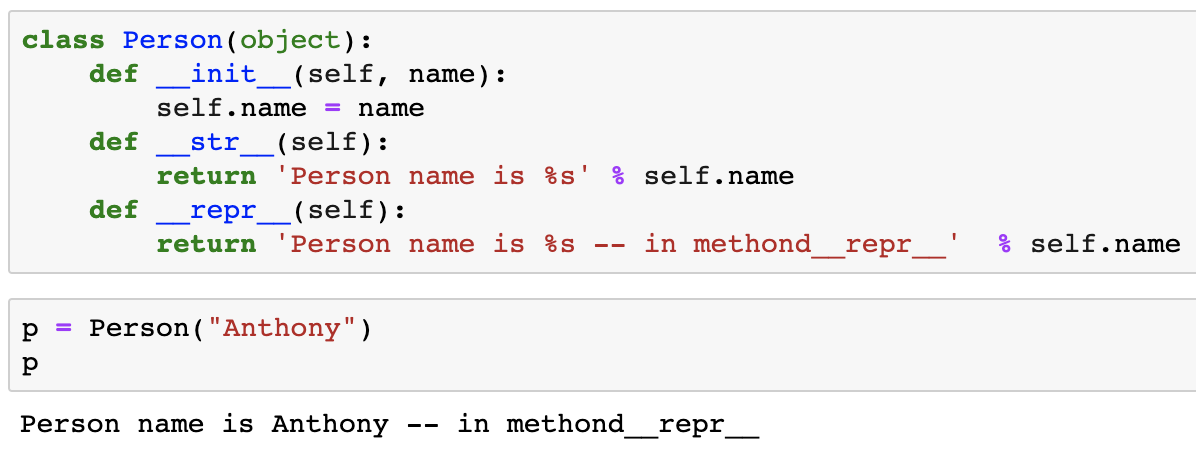
当我们打印一个对象时，如何实现我们想输出的信息呢？实际上print()函数在打印对象时，调用的是对象的\_\_str\_\_()方法，我们只要实现这个方法，就可以输出我们想看到的信息，示例如下：



如果我们不用print()打印对象的信息，而是直接查看变量的返回，则输出的信息仍然看起来有些怪：



实际上直接显示变量调用的不是\_\_str\_\_()，而是\_\_repr\_\_()，两者的区别是\_\_str\_\_()返回用户看到的字符串，而\_\_repr\_\_()返回程序开发者看到的字符串，也就是说，\_\_repr\_\_()是为调试服务的。我们可以再实现\_\_repr\_\_()方法返回我们想看到的内容。

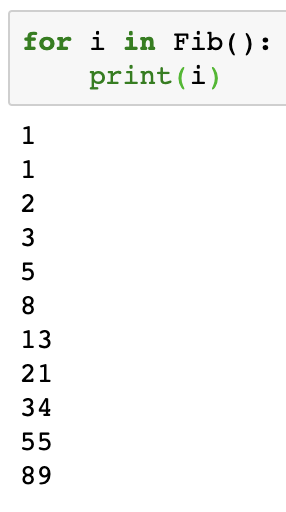


## \_\_iter\_\_ ()

如果我们想定义一个类，它可以在for循环中使用遍历元素，就像list和tuple。前面的学习我们知道这个类的行为是一个迭代器。那么在类定义中，我们只要实现一个\_\_iter\_\_()和获取下一个元素的\_\_next\_\_()，就可以让这个类满足迭代器的特性而应用于循环。示例如下，我们定义了一个Fib()类用以返回斐波那契数列：

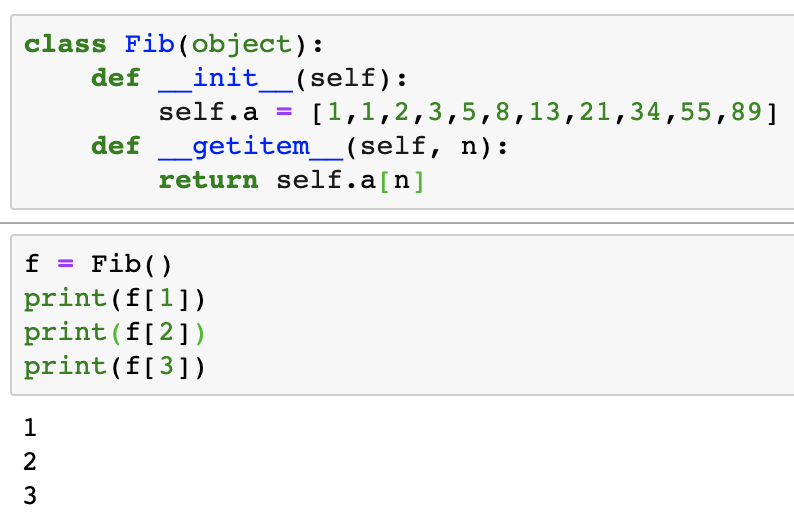


然后我们将Fib对象应用于for循环，返回斐波那契数列：



## \_\_getitem\_\_ ()

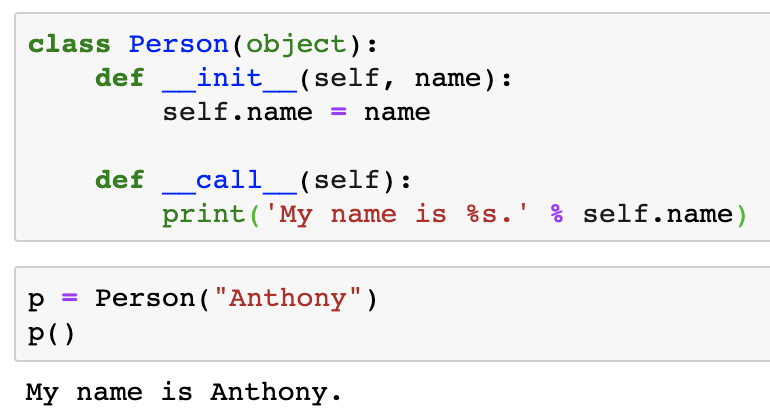
为Fib类实现\_\_iter\_\_()方法，我们就可以像迭代器一样在for循环中遍历元素，如果我们想像通过下标访问list或tuple集合那样访问对象时，此时我们就要实现\_\_getitem\_\_()方法。下面的示例中，类实现了\_\_getitem\_\_()方法，根据索引返回指定的数列中的数字，然后我们就可以通过索引访问对象里的指定元素。也就是在Python中调用f[n]就相当于调用f.\_\_getitem\_\_(n):



与之对应的是\_\_setitem\_\_()方法，把对象视作list或dict来对集合赋值。最后，还有一个\_\_delitem\_\_()方法，用于删除某个元素。总之，通过上面的方法，我们自己定义的类表现得和Python自带的list、tuple、dict没什么区别，这完全归功于动态语言的“鸭子类型”，不需要强制继承某个接口。

## \_\_call\_\_ ()

我们之前学习了直接对象名相当于调用对象的\_\_repr\_\_()方法，那么我们如果这样直接调用对象：*对象名()*，实际上它对应的是另一种特殊函数\_\_call\_\_()。我们来看下面示例。\_\_call\_\_()还可以定义参数。让我们可以调用一个实例就像调用一个参数一样。在Python这些足够灵活，只要你实现了对应的特殊函数，就可以像函数一样调用对象：



另外，我们如何在程序中判断一个对象能否被直接调用呢，我们可以使用Callable函数进行判断，实现了\_\_call\_\_()方法的类的对象我们就认为是Callable。方法本身就是被调用的，自然我们认为方法也是callable。示例如下：



所谓的定制，就是不同的方式使用对象时，我们要实现对应的特殊函数，让类的对象满足我们的需求。Python的class允许定义许多定制方法，可以让我们非常方便地生成特定的类。本节介绍的是最常用的几个定制方法，还有很多可定制的方法，请参考Python的官方文档<https://docs.python.org/3/reference/datamodel.html#special-method-names>。

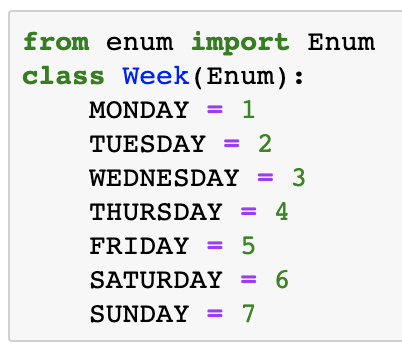
# 枚举类

在我们的代码里，经常要使用常量，如果一些不同值得常量代表的意义相近，这些类型我们就可以用枚举的类进行定义，很多编程语言都有枚举的概念，Python也是。在python中枚举是一种类（Enum,IntEnum），存放在enum模块中。枚举类型可以给一组标签赋予一组特定的值。

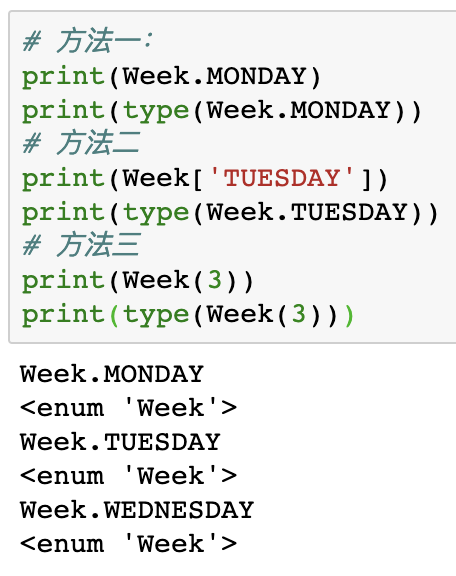
枚举有以下特点：

* 枚举类中不能存在相同的标签名
* 枚举是可迭代的
* 不同的枚举标签可以对应相同的值，但它们都会被视为该值对应第一个标签的别名
* 如果要限制定义枚举时，不能定义相同值的成员。可以使用装饰器@unique【要导入unique模块】
* 枚举成员之间不能进行大小比较，可进行等值和同一性比较
* 枚举成员为单例，不可实例化，不可更改

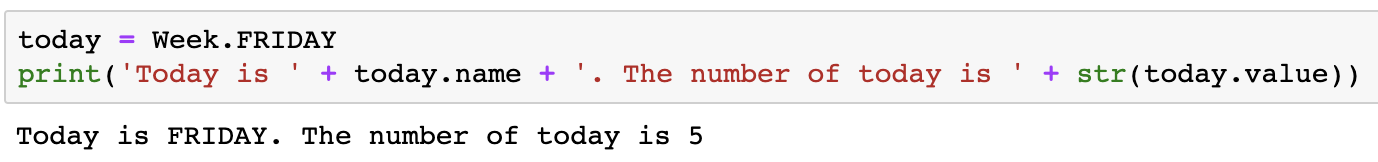
我们首先定义一个枚举类型,在Python中我们通过继承Enum类来实现：



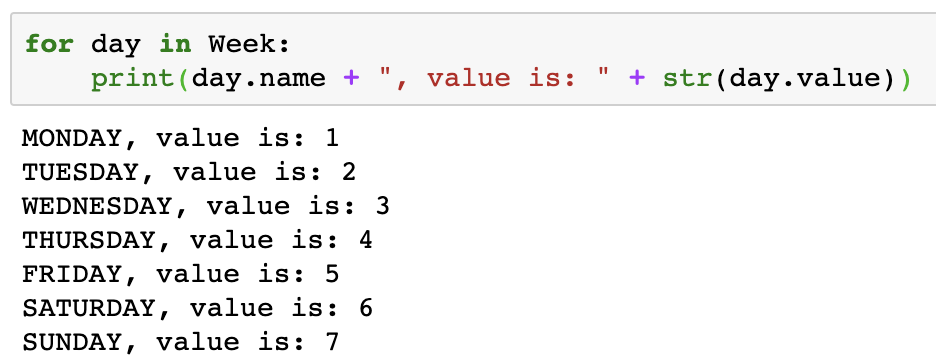
第一个示例，获取在枚举类中定义的枚举成员：



第二个示例，枚举成员中包含标签名和该标签对应的值：



第三个示例，我们来遍历一个枚举对象的值：



第四个示例，限定不含相同值的枚举类@unique，当我们定义的枚举值有重复时，代码会报错：

