# 课程介绍

* 1. 迭代器和生成器
* 2. 自定义属性操作方法

1. 迭代器和生成器
   1. 什么是迭代器

很多语言都有迭代器的概念，可以简单的理解迭代器就是一个工具对象，它的特点是：

* 提供了一个方法，可以不断获取对象里的数据。
* 用户不需要关心内部细节，只需要通过这个特定接口获取数据。
* 当没有数据时，再次调用时，会得到一个停止标记。

比如在python中的列表和字典容器，要想获得他里面的数据？一个是通过[]的语法获取其中一个，另外一个就是借助for…in的语法。其实for…in就是一个隐式迭代器的使用方法。

比如设计一个迭代器，可以把初始化的数据以平方值反馈回来。例如用户向迭代器里初始化1，2，3这3个数字，但用户访问时，会得到1，4，9。

* 1. python迭代器的使用方法
     1. 如何使用迭代器
* 在python中，提供了iter()方法，他会返回一个迭代器；
* 在迭代器中会定义一个叫做\_\_next\_\_方法，他提供获取数据的接口；
* 在python3中，提供了一个next()的内置函数，专门调用迭代器的\_\_next\_\_方法；
* 一旦没有数据后，再次访问时，会收到StopIteration的异常
  + 1. 如何判断一个对象是否可以迭代

可以使用isinstance() 判断一个对象是否是Iterable 对象：

from collections import Iterable

isinstance([], Iterable) # True

isinstance(100, Iterable) # False

* 1. 注意事项

for...in实际是python语言内部实现了iter和next的调用，为使用者隐藏了迭代过程的细节。

* 1. 什么是生成器

利用迭代器，我们可以不断的获取数据。在实际开发中，我们有时候希望数据一开始不存放在内存上，而且在遵循某种规律的情况下，用的时候再给用户。

在python中为我们提供生成器(generator)。他不仅具备迭代器的能力，而且他是使用时再分配，而不是事先分配空间的。

在实际开发时，在需要迭代特性和用时获取特性时，建议使用生成器来实现，他的语法简单清晰。

* + 1. 列表生成器

要创建一个生成器，有很多种方法。第一种方法很简单，只要把一个列表生成式的[ ] 改成( )。

L = [ x\*2 for x in range(5)]

G = ( x\*2 for x in range(5))

next(G) # 0

next(G) # 2

next(G) # 4

next(G) # 6

next(G) # 8

next(G) # StopIteration

这种方式，我们称之为列表生成器，可以看的出来，他也是迭代器，只是书写起来会非常清晰。

* + 1. 函数生成器

generator非常强大。如果推算的算法比较复杂，用列表生成式的逻辑无法实现的时候，还可以用函数来实现。

他的语法形式非常简单，就是将普通函数的return替换为yield。

yield这里的意思不是返回，而是抛出后面的变量值，然后保存当前函数的状态，等待next的操作。

可以看出来，当yiled在函数中使用后，这个函数已经升级为了迭代器。

我们用函数生成器的方式改写上面的例子：

def power2(n):

cnt = 1

while cnt <= n:

yield cnt \* 2

* 1. 总结

迭代器：

获取容器中的数据，只要实现了\_\_next\_\_方法，就称之为迭代器。

生成器：

动态获取数据的工具。本质也是一个迭代器。

1. 对象的属性
   1. 动态添加性

在程序运行过程中，给类或对象添加属性，是动态语言的一个特性。

* + 1. 给对象添加属性

def ismale(person):

return person.sex == “男”

class Person(object):

def \_\_init\_\_(self, name = None, age = None):

self.name = name

self.age = age

P = Person("小明", "24")

# P.sex = “男”

print(“{} 是男性吗？{}”.format(P.name, ismale(P)))

可以看出来，ismale函数需要判断一个对象是否有sex属性，同时这个属性的值是不是“男”字符。

Person类天然不带性别，但是可以在执行过程中，给新产生的对象添加(绑定)sex属性。

* + 1. 给类添加属性

给类添加属性和给对象添加属性，思路和方法都是一样的。

不过这里要注意的是，在对象访问属性时，如果这个属性在对象空间里找不到，那么会以此查找类空间，类空间也没有的时候，继续找他的父类空间，形成一种变量查找作用域链的概念。

P1 = Person("小丽", "25")

P1.sex

AttributeError: Person instance has no attribute 'sex'

Person类没有sex属性，对象P1访问时，报错。

Person.sex = None #给类Person添加一个属性

P1 = Person("小丽", "25")

print(P1.sex)

None #如果P1这个实例对象中没有sex属性的话，那么就会访问它的类属性可以看到没有出现异常

* 1. 动态属性的删除

删除的方法:

* del 对象.属性名
* delattr(对象, "属性名")

通过以上例子可以得出一个结论：相对于动态语言，静态语言具有严谨性！所以，玩动态语言的时候，小心动态的坑！

那么怎么避免这种情况呢？请使用\_\_slots\_\_

* 1. \_\_slots\_\_（了解）

如果我们想要限制实例的属性怎么办？比如，只允许对Person实例添加name和age属性。

为了达到限制的目的，Python允许在定义class的时候，定义一个特殊的\_\_slots\_\_变量，来限制该class实例能添加的属性：

class Person(object):

\_\_slots\_\_ = ("name", "age")

P.score = 100

AttributeError: Person instance has no attribute 'score'

【注意】：

使用\_\_slots\_\_要注意，\_\_slots\_\_定义的属性仅对当前类实例起作用，对继承的子类是不起作用的。

* 1. 属性命名规则
     1. name mangling

在python的class中，当对象绑定一个以双下划线命名的变量，称之为私有变量。其实在python中，并没有私有变量的说法，所谓的私有是人为定义的一种变量访问方式而已。

但在python的语法中，利用了名字重整的方法来避免用户直接访问这个变量。

* + 1. 命名规则
* xx: 公有变量
* \_x: 单前置下划线,私有化属性或方法，from somemodule import \*禁止导入,类对象和子类可以访问，但不希望访问到。相当于protected。
* \_\_xx：双前置下划线,避免与子类中的属性命名冲突，无法在外部直接访问(名字重整所以访问不到)
* \_\_xx\_\_:双前后下划线,用户名字空间的魔法对象或属性。例如: \_\_init\_\_, \_\_不要自己发明这样的名字
* xx\_:单后置下划线,用于避免与Python关键词的冲突

1. 自定义属性操作方法
   1. 几个场景

传统意义上，通过赋值操作，让一个属性变量指向了一个内存区域。但在实际开发时，为了更友好的让用户进行访问，我们也希望定义一个属性变量名，但是指向的内容可能是不同的情况下有不同的效果。比如获取一个人的税后工资时，需要根据税率梯度进行计算才能得到结果。

定义了一个属性变量，但在赋值时，需要做一些数据有效性的检查，比如age，就不能为负数。

* 1. property

很显然，在写一个变量，读一个变量时，不再是单纯基本内存数据空间的读写了，而是一个具有一定逻辑的程序进行管理的。

在python中，提供了一个内置的property函数，他可以把读、写、删除的动作进行统一管理，然后映射成一个变量名，供用户访问。

class Person:

def \_\_init\_\_(self):

self.\_\_age = 0

# 读取方式

def get\_age(self):

return self.\_\_age

# 写入方式

def set\_age(self, value):

if value < 0:

raise ValueError('Expected a < 0 number')

self.\_\_age = value

# 删除方式

def del\_age(self):

raise AttributeError("Can't delete attribute")

# 制作一个property从自定义的方法

age = property(get\_age, set\_age, del\_age)

p = Person()

p.age = -10

* + 1. 装饰器写法

上面的写法可以满足用户对一个变量的所有可能操作了。但是写法显得特别麻烦，python提供了一个内置的@property的装饰器，来帮助程序员简化代码。

class Person:

def \_\_init\_\_(self):

self.\_\_age = 0

@property

def age(self):

return self.\_\_age

@age.setter

def age(self, value):

if value < 0:

raise ValueError('Expected a < 0 number')

self.\_\_age = value

@age.deleter

def age(self):

raise AttributeError("Can't delete attribute")

p = Person()

p.age = -10

1. 元类（了解、扩展）
   1. python世界的对象产生规律

对象是从类中生成来的，那么在python中类是从哪里来的那？

实际上，在python中，会有这么几个概念，我们可以类比道家的“道生一、一生二、二生三、三生万物”的道理。

* 道 即是 type
* 一 即是 metaclass(元类，或者叫类生成器)
* 二 即是 class(类，或者叫实例生成器)
* 三 即是 instance(实例)
* 万物 即是 实例的各种属性与方法，我们平常使用python时，调用的就是它们。
  1. 类和对象及属性

二、三和万物是非常好理解的，就像python的类生成了对象，而对象通过\_\_init\_\_方法绑定了很多对象的属性，同样，对象也可以在执行过程中绑定其他属性。

# 创建一个Person类，拥有属性say\_hello ------------------------- 二的起源

class Person():

def \_\_init\_\_(self, name="人类"):

self.name = name

def say\_hello(self):

print('Hello, %s.' % self.name)

# 从Person类创建一个实例jim --------------------------- 二生三

jim = Person ('Jim')

# 使用jim调用方法say\_hello ---------------------------- 三生万物

jim.say\_hello()

# 使用jim创造了比Person还多的属性 ---------------------------- 三生万物

jim.sex = '男'

如何来证明这个想法那，在python中有一个内置的方法type。

* 1. type方法

type最开始我们知道的意思是查看一个对象的类型，那么就可以理解为是谁创建了这个对象。那么

>>> print type(1) #数值的类型

<type 'int'>

>>> print type("1") #字符串的类型

<type 'str'>

>>> print type(jim) #实例对象的类型

<class '\_\_main\_\_.Person'>

>>> print type(Person) #类的类型

<type 'type'>

从中我们发现了，类是由type创建的。我们来看下type的申明：

type(object\_or\_name, bases, dict)

type(object) -> the object's type

type(name, bases, dict) -> a new type

那么type是怎么创造类的那。

def hello(self):

print("你好{}".format(self.name))

HelloWorld = type("HiWorld", (object,), {'fn\_hello': hello})

a1 = HelloWorld()

a1.name = 'Jim'

a1.fn\_hello()

这个例子就是道生而二了。

type的三个参数，符合人类社会的客观规律：我是谁，我从哪里来，我要到哪里去；

* 1. type和类的关系

type可以直接生成class，但其实这中间还有一个中间体，他就称之为元类。他的名字为了区别普通类，都建议使用MetaClass作为后缀。元类相当于生成普通类的模具。

比如我们现在有一个需求，需要生成很多类，这个类都要具备一个方法叫say\_xxx，例如一个叫Student的类，就要有一个say\_student方法；一个叫Doctor的类，就要有一个say\_doctor的方法；那么我们就构造一个这样规律的元类，然后通过这个元类去生成这些普通类。

下面看一个元类：

class SayMetaClass(type):

def \_\_new\_\_(cls, name, bases, attrs):

attrs['say\_'+name] = lambda self,value,saying=name: print(saying+','+value+'!')

return type.\_\_new\_\_(cls, name, bases, attrs)

要定义一个元类，必须满足如下条件：

元类必须由type生成，这就是道生一的概念。

元类的操作必须在\_\_new\_\_中完成，同时也会传递3要素，我是谁，我从哪里来，我要到哪里去。

如何生成刚才需求：

# 道生一：传入type

class SayMetaClass(type):

# 传入三大永恒命题：类名称、父类、属性

def \_\_new\_\_(cls, name, bases, attrs):

# 创造“天赋”

attrs['say\_'+name] = lambda self,value,saying=name: print(saying+','+value+'!')

# 传承三大永恒命题：类名称、父类、属性

return type.\_\_new\_\_(cls, name, bases, attrs)

# 一生二：创建类

class Hello(object, metaclass=SayMetaClass):

pass

# 二生三：创建实列

hello = Hello()

# 三生万物：调用实例方法

hello.say\_Hello('world!')

* 1. 使用元类的场景

设计一种类模型时候，找到一个一系列类的共同规律，比如后期django的ORM框架。

除非是要设计框架，工作中99%是不会用到元类的，学习元类的目的就是理解下python生成对象的一个过程。

1. 课程总结
   1. 重点
      * 1. 列表生成器的语法和特点
        2. 如何判断一个对象是否可以迭代
        3. property装饰器的用法
   2. 难点
      * 1. python中的属性动态性的理解
        2. 函数和生成器函数的区别
   3. 如何掌握？
      * 1. 翻译成中文，然后翻译成代码，写3遍
        2. 利用ipython，快速验证信息
2. 课后练习
   * + 1. 兔子繁殖问题：

有一对兔子，从出生后第三个月起每个月都生一对兔子，小兔子长到第三个月后又生一对兔子，假如兔子都不死，每个月兔子对数为多少？

提示，使用函数生成器方式来做。

* + - 1. 汇率计算器

设计一个类，用户初始化账户里的人民币数量，提供美元、港元、泰铢的转换功能，要求用户直接通过属性访问。

扩展：在上一个类的基础上，增加用户收入美元，港元，泰铢的功能。

1. 面试题
   * + 1. Python中单下划线\_foo与双下划线\_\_foo与\_\_foo\_\_的成员，下列说法正确的是？（多选）

A. \_foo 不能直接用于’from module import \*’

B. \_\_foo解析器用\_classname\_\_foo来代替这个名字，以区别和其他类相同的命名

C. \_\_foo\_\_代表python里特殊方法专用的标识

D. \_\_foo 可以直接用于’from module import \*’

* + - 1. python中单下划线和双下划线的区别。

单下划线：私有化属性或方法，from somemodule import \*禁止导入,类对象和子类可以访问。

双下划线：无法在外部直接访问，被名字重整了。

* + - 1. 简述如何使用迭代器？

通过iter()将一个可迭代对象生成为迭代器A。

通过next()方法从A中获取元素。

当收到StopIteration异常时，说明迭代器里无元素了。

也可以使用for...in的方式自动获取完所有元素。

* + - 1. 列表推导式和列表生成器的区别？

列表推导式：最终生成一个列表数据结构，使用[]作为语法限定符，里面跟循环规律表达式。

列表生成器：最终生成一个生成器对象，通过next方法动态获取数据，使用()作为语法限定符。

1. 扩展知识或课外阅读推荐（可选）
   1. 如何自定义一个迭代器
      1. 迭代器的本质

* **可迭代对象：** 具备\_\_iter\_\_方法的对象
* **迭代器：** 具备\_\_next\_\_和\_\_iter\_\_方法的对象
  + 1. 示例

定义一个迭代器，获取初始化数据的平方。

class Actor(object): # 自定义一个迭代器

def \_\_init\_\_(self, objs):

self.objs = objs

self.number = len(objs)

self.index = 0

def \_\_next\_\_(self): # 实现\_\_next\_\_方法

if self.index >= self.number:

raise StopIteration

res = self.objs[self.index] \*\* 2

self.index += 1

return res

def \_\_iter\_\_(self): # 实现\_\_iter\_\_返回自己

return self

class Able(object): # 自定义一个可迭代对象

def \_\_init\_\_(self):

self.data = []

def append(self, obj):

self.data.append(obj)

def \_\_iter\_\_(self):

return Actor(self.data)

a1 = Able()

a1.append(1)

a1.append(3)

a1.append(7)

for x in a1:

print(x)

* 1. 斐波那契数列

使用生成器来实现一个可以不断获取的斐波那契数列，可以初始化该数列的最大项。

def fib(n):

current = 0

num1, num2 = 0, 1

while current < n:

num = num1

num1, num2 = num2, num1+num2

current += 1

yield num

return 'done'