迭代器与异常

金融科技协会 2020年11月26日

目录

[1 异常 2](#_Toc20443)

[1.1 定义 2](#_Toc30169)

[1.2 raise故意引发指定异常 2](#_Toc14220)

[1.3 常见的内置异常类 3](#_Toc22625)

[1.3.1 AttributeError 3](#_Toc328)

[1.3.2 IndexError 3](#_Toc27854)

[1.3.3 KeyError 3](#_Toc7741)

[1.3.4 NameError 4](#_Toc31192)

[1.3.5 SyntaxError 4](#_Toc29564)

[1.3.6 TypeError 5](#_Toc3238)

[1.3.7 ZeroDivisionError 5](#_Toc20809)

[1.3.8 ValueError 5](#_Toc22609)

[1.4 捕获异常 6](#_Toc31183)

[1.4.1 捕获单种指定异常 6](#_Toc14035)

[1.4.2 多个except子句 6](#_Toc12356)

[1.4.3 一个except捕获多种异常 6](#_Toc27824)

[1.4.4 捕获任何异常，并对异常进行识别 7](#_Toc15855)

[1.4.5 代码捕获异常后，仍报错 7](#_Toc27418)

[1.4.6 代码捕获异常后，报指定错误 8](#_Toc9103)

[1.4.7 else子句 8](#_Toc4672)

[1.4.8 finally子句 9](#_Toc13870)

[2 迭代器和生成器 10](#_Toc4566)

[2.1 迭代器、可迭代对象 10](#_Toc8757)

[2.2 初始化一个迭代器 11](#_Toc23042)

[2.3 使迭代器输出下一个迭代值 11](#_Toc16145)

[2.4 迭代器转序列 11](#_Toc21741)

[2.5 生成器 11](#_Toc17649)

[2.6 生成器的使用 13](#_Toc7697)

[2.6.1 for 13](#_Toc7517)

[2.6.2 next() 13](#_Toc11491)

[2.6.3 \_next\_() 14](#_Toc3523)

[2.6.4 next()与send() 14](#_Toc21950)

[2.6.5 throw和close 15](#_Toc13097)

**1 异常**

* 1. **定义**

Python使用**异常对象**来表示异常状态，并在遇到错误时引发异常。异常对象未被处理（或**捕获**）时，程序将终止并显示一条错误消息（traceback）。

每个异常都是某个类（这里是ZeroDivisionError）的实例。你能以各种方式引发和捕获这些实例，从而**逮住错误并采取措施，而不是放任整个程序失败**。

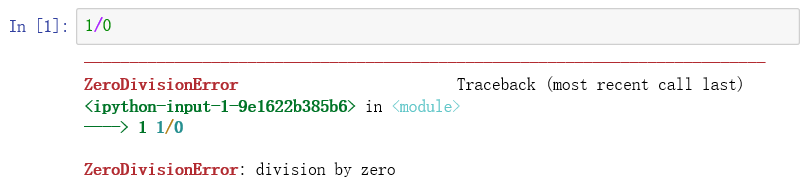


图1：ZeroDivisionError错误示例

* 1. **raise故意引发指定异常**

要引发异常，可使用raise语句，并将一个类（必须是Exception的子类）或实例作为参数。将类作为参数时，将自动创建一个实例。下面的示例使用的是内置异常类Exception：

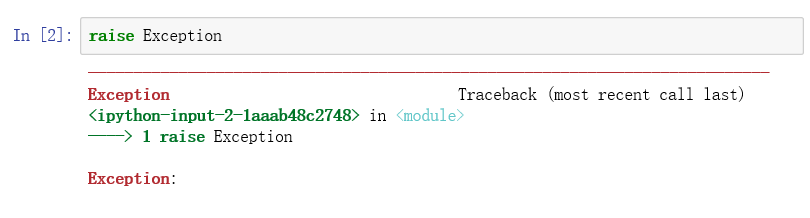


图2：raise引发内置异常类Exception示例

在第一个示例（raise Exception）中，引发的是通用异常，没有指出出现了什么错误。在第二个示例中，添加了错误消息hyperdrive overload。

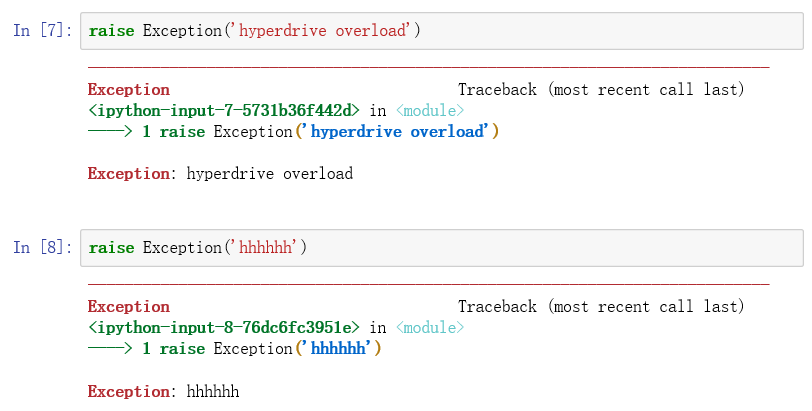


图3：raise引发错误并添加错误消息

* 1. **常见的内置异常类**

• Exception 几乎所有的异常类都是从它派生而来的

• AttributeError 引用属性或给它赋值失败时引发

• OSError 操作系统不能执行指定的任务（如打开文件）时引发，有多个子类

• IndexError 使用序列中不存在的索引时引发，为LookupError的子类

• KeyError 使用映射中不存在的键时引发，为LookupError的子类

• NameError 找不到名称（变量）时引发

• SyntaxError 代码不正确时引发

• TypeError 将内置操作或函数用于类型不正确的对象时引发

• ValueError 将内置操作或函数用于这样的对象时引发：其类型正确但包含的值不合适

• ZeroDivisionError 在除法或求模运算的第二个参数为零时引发

* + 1. **AttributeError**

当引用属性或给它赋值失败时引发，示例如下：

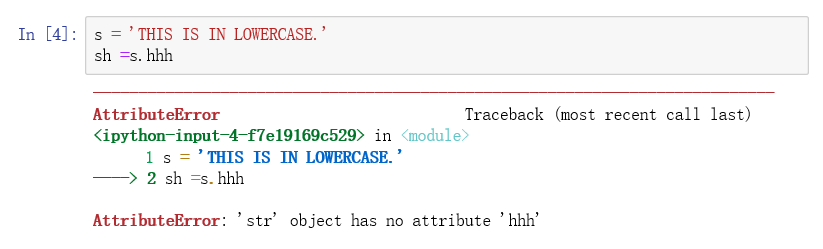


图4：attribute error示例

* + 1. **IndexError**

使用**序列**中不存在的索引时引发，为LookupError的子类，示例如下：

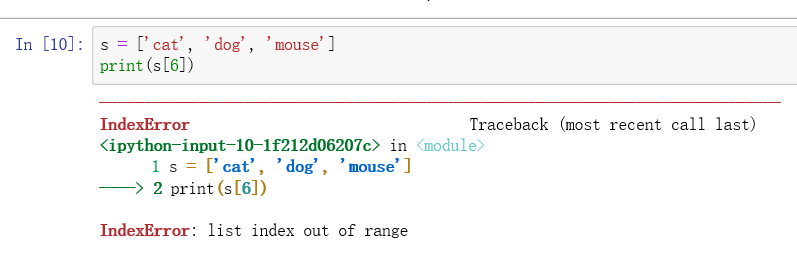


图5：IndexError示例

* + 1. **KeyError**

使用**映射**中不存在的**键**时引发，为LookupError的子类，示例如下：

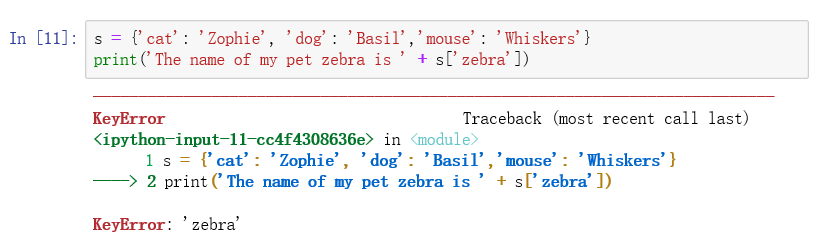


图6：KeyError示例

* + 1. **NameError**

找不到名称（变量,函数等）时引发，示例如下：

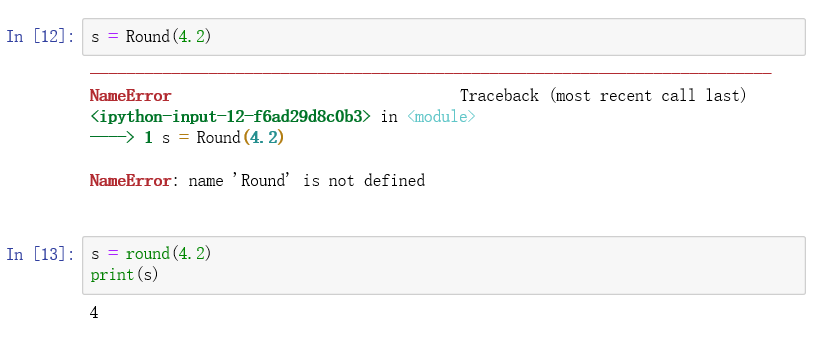


图7：NameError示例

* + 1. **SyntaxError**

代码不正确时引发：

• 缩进

• 符号漏打

• 中文符号

• == 打成了 =

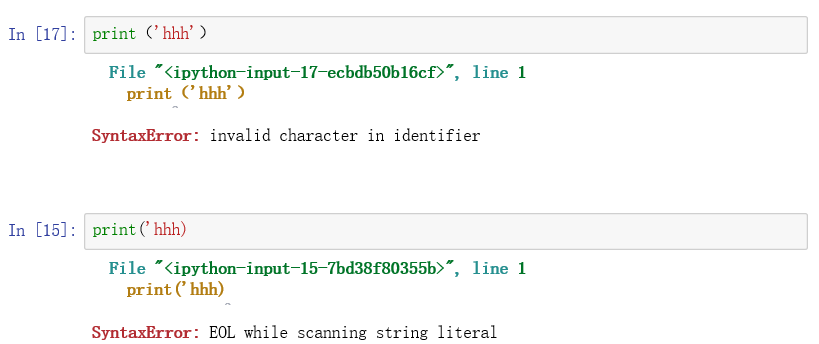


图8：SyntaxError示例

* + 1. **TypeError**

将内置操作或函数用于类型不正确的对象时引发，示例如下：

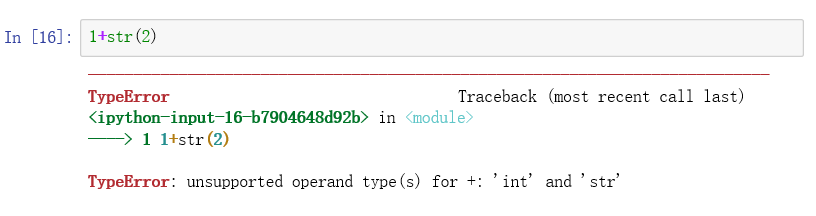


图9：TypeError示例

* + 1. **ZeroDivisionError**

在除法或求模运算的第二个参数为零时引发，示例如下：

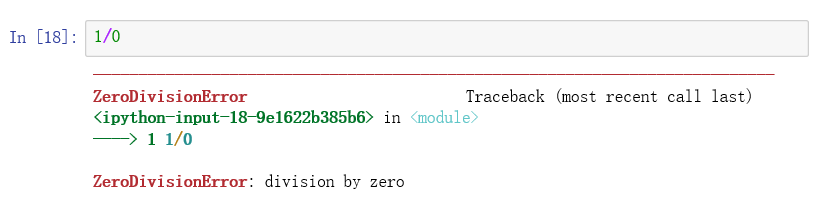


图10：ZeroDivisionError示例

* + 1. **ValueError**

将内置操作或函数用于这样的对象时引发：其类型正确但包含的值不合适，示例如下：



图11：ValueError示例

* 1. **捕获异常**
     1. **捕获单种指定异常**

异常比较有趣的地方是可对其进行处理，通常称之为捕获异常。为此，可使用try...except:

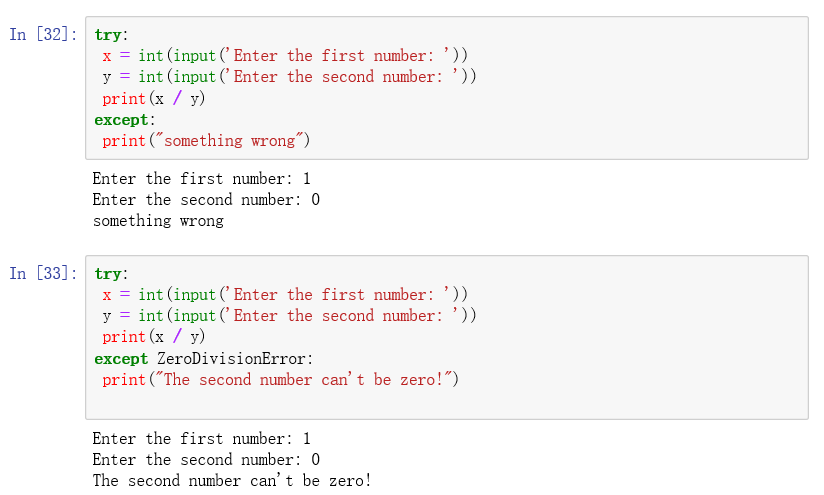


图12：try...except捕获异常示例

* + 1. **多个except子句**

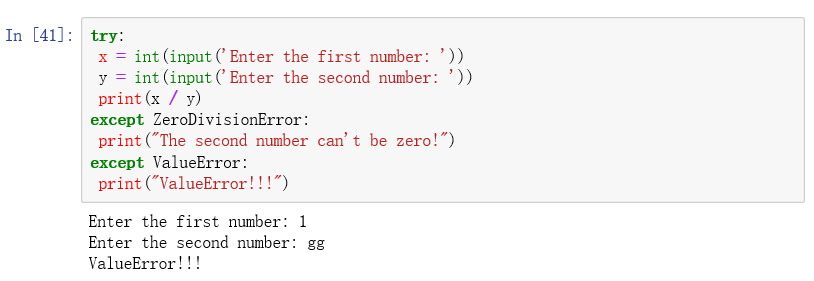


图13：多个except子句捕获异常使用示例

* + 1. **一个except捕获多种异常**

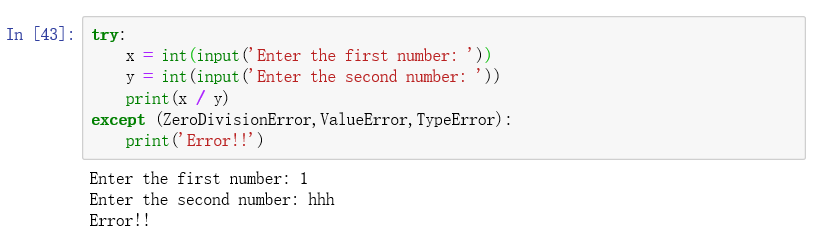


图14：使用一个except捕获多种异常示例

* + 1. **捕获任何异常，并对异常进行识别**

使用语句try...except Exception as e:

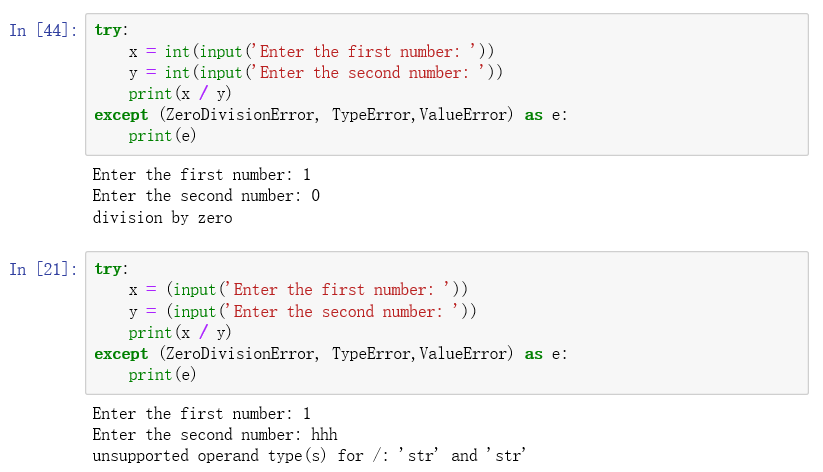


图15：try...except Exception as e使用示例

* + 1. **代码捕获异常后，仍报错**

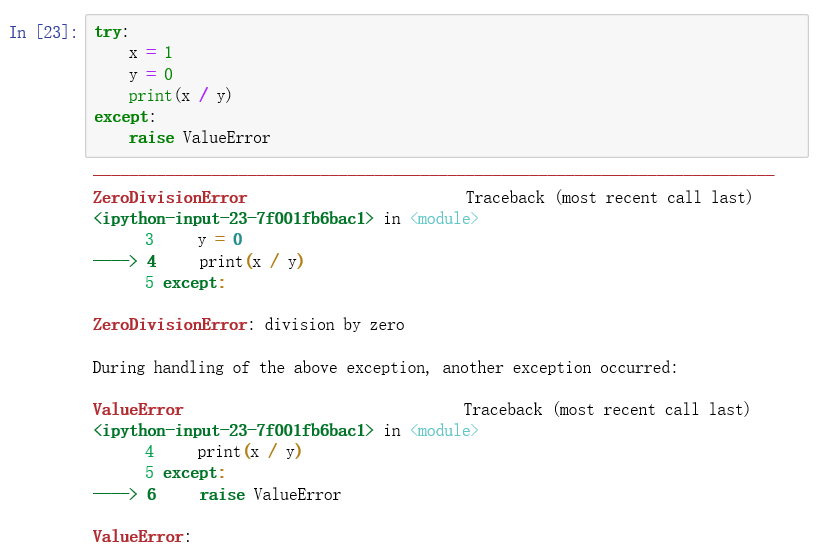


图16：代码捕获异常后仍报错处理示例

* + 1. **代码捕获异常后，报指定错误**



图17：代码捕获异常后报指定错误处理示例

* + 1. **else子句**

在有些情况下，在没有出现异常时执行一个代码块很有用。为此，可像条件语句和循环一样，给try/except语句添加一个else子句。



图18：else子句使用示例

使用except Exception as e，就可利用8.3.4节介绍的技巧在这个小型除法程序中打印更有用的错误消息

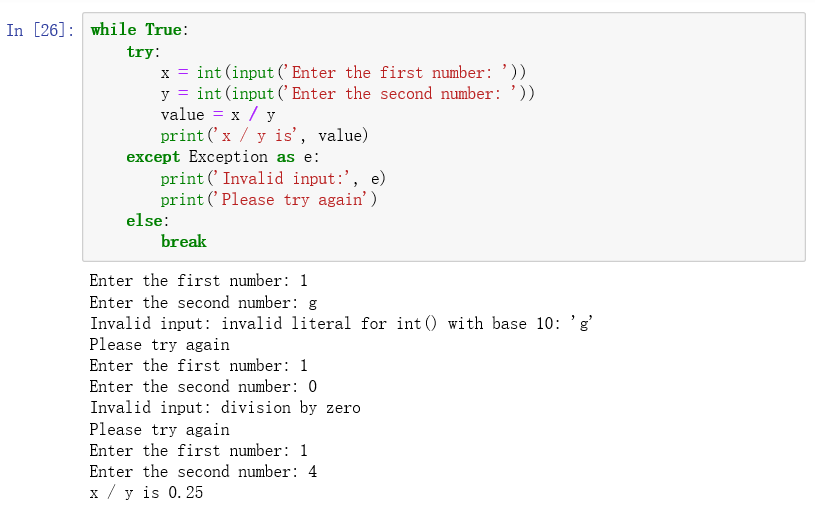


图19：else子句结合except Exception as e使用示例

* + 1. **finally子句**

用于在发生异常时执行清理工作。这个子句是与try子句配套的。

不管try子句中发生什么异常，都将执行finally子句。

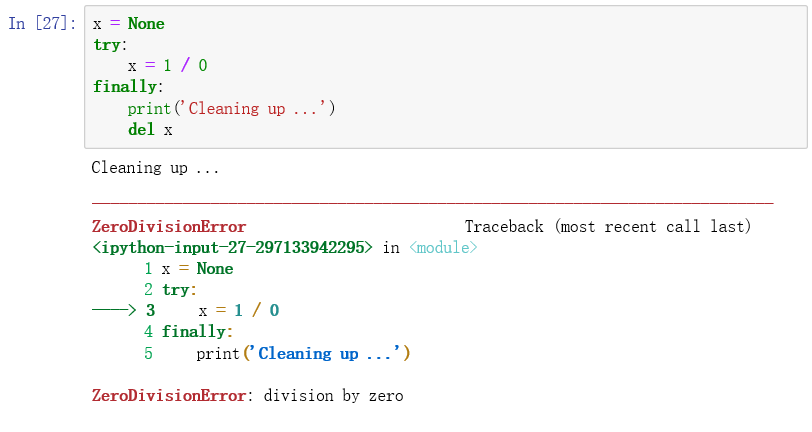


图20：finally子句使用示例

1. **迭代器和生成器**
   1. **迭代器、可迭代对象**

• **迭代**是重复反馈过程的活动，其目的通常是为了接近并到达所需的目标或结果。每一次对过程的重复被称为一次“迭代”，而每一次迭代得到的结果会被用来作为下一次迭代的初始值。

• 可迭代对象不一定是迭代器，迭代器一定是可迭代对象。因为迭代器一定会实现 \_\_ iter \_\_ 方法，而可迭代对象尽管实现了 \_\_ iter \_\_ 也不一定实现 \_\_ next \_\_方法

• 迭代器协议是指：对象需要提供next方法，它要么返回迭代中的下一项，要么就引起一个StopIteration异常，以终止迭代

• 可迭代对象就是：实现了迭代器协议的对象

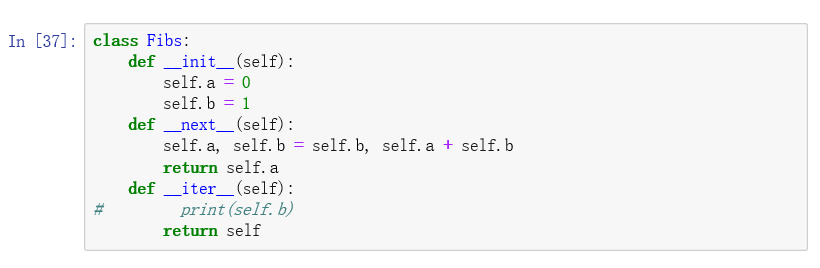
• 协议是一种约定，可迭代对象实现迭代器协议，Python的内置工具(如for循环，sum，min，max函数等)使用迭代器协议访问对象。

iter方法：

• 方法iter返回一个迭代器，它是包含方法next的对象，而调用这个方法时可不提供任何参数。

• 当你调用方法next时，迭代器应返回其下一个值。

• 如果迭代器没有可供返回的值，应引发StopIteration异常。



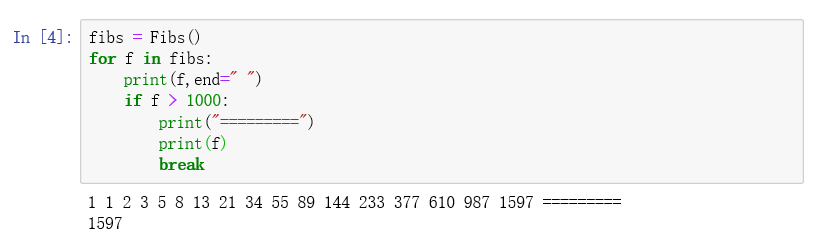


图21：iter使用示例

注意到这个迭代器实现了方法iter，而这个方法返回迭代器本身。在很多情况下，都在另一个对象中实现返回迭代器的方法iter，并在for循环中使用这个对象。但推荐在迭代器中也实现方法iter（并像刚才那样让它返回self），这样迭代器就可直接用于for循环中。

* 1. **初始化一个迭代器**

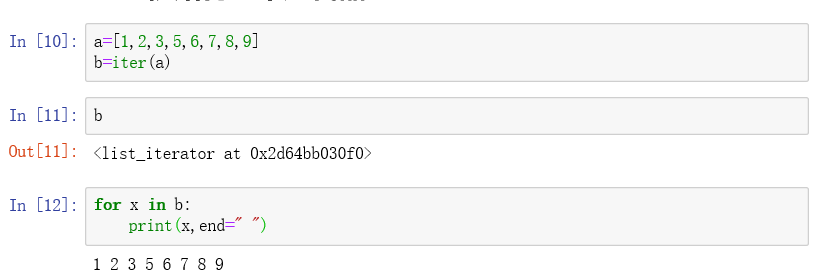


图22：初始化迭代器示例

* 1. **使迭代器输出下一个迭代值**

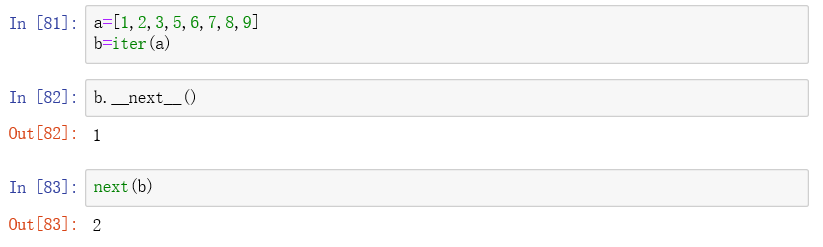


图23：使迭代器输出下一个迭代值示例

* 1. **迭代器转序列**

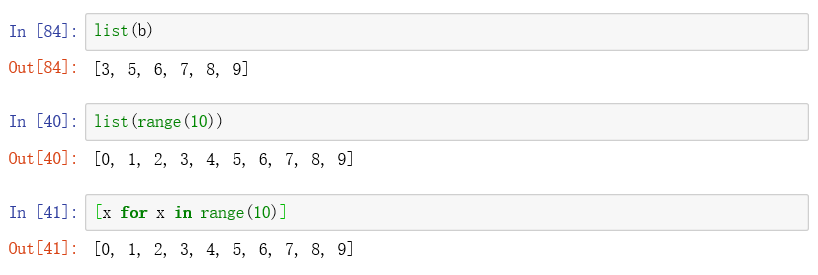


图24：迭代器转序列示例

* 1. **生成器**

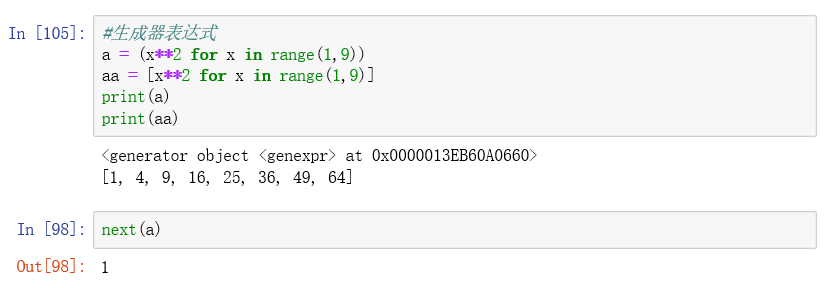
生成器的主要思想：对于可以公式自动生成的数字序列，由计算机不断迭代，每次只生成一个数字，从而通过循环遍历生成序列中的所有元素。所以说，生成器产生的不是一个静态的值（比如类似字符串、元组和列表等，都是一次性生成所有值），而是一个动态的数据流。

图25：生成器示例

函数体中有关键字yield。yield关键字类似于return，当生成器被next()函数调用时，会返回其后的变量，相当于程序中断；当再次调用next()函数后，生成器会从中断的yield语句处继续执行，也就是用多少，取多少，不占内存。

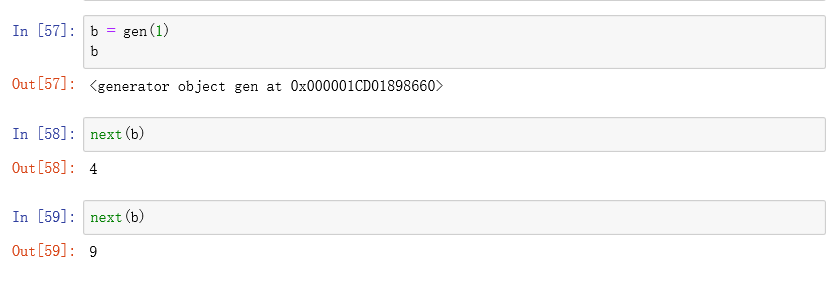
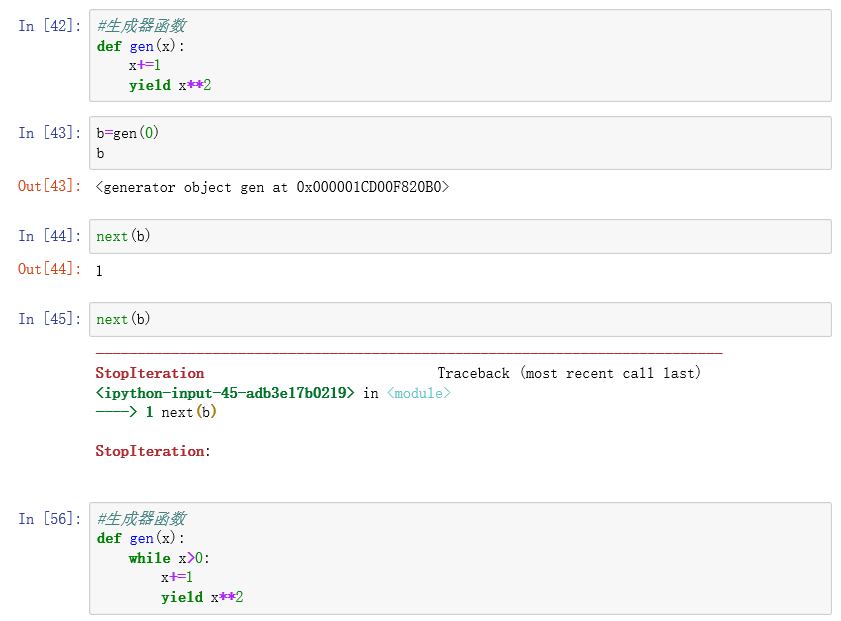


图26：yield使用示例

可以看到，变量a和b都是生成器，我们不能直接使用a、b，因为它们实际上保存的是一个公式，使用时可以调用内置函数next()，由next(a)、next(b)来动态生成序列中的下一个值。采用生成器的好处是：节省内存空间，特别是对于数据量大的序列，一次性生成所有值将会耗费大量内存，而采用生成器可以极大地节省存储空间。同时，生成器还可以处理无限长的序列。比如，上述实例中，变量b就是一个无限序列，理论上可以永远next(b)，而且每次都是按顺序生成其中的一个值。

* 1. **生成器的使用**
     1. **for**

可以通过for循环遍历所有值，示例如下：

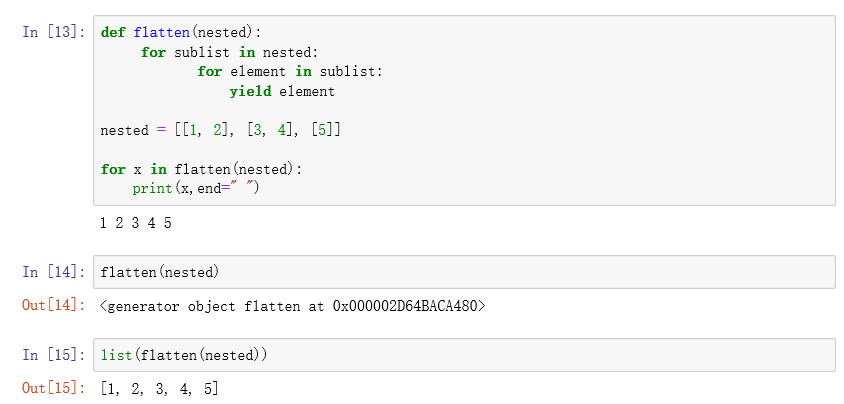


图27：for循环遍历示例

* + 1. **next()**

用内置函数next()循环生成下一个值，示例如下：

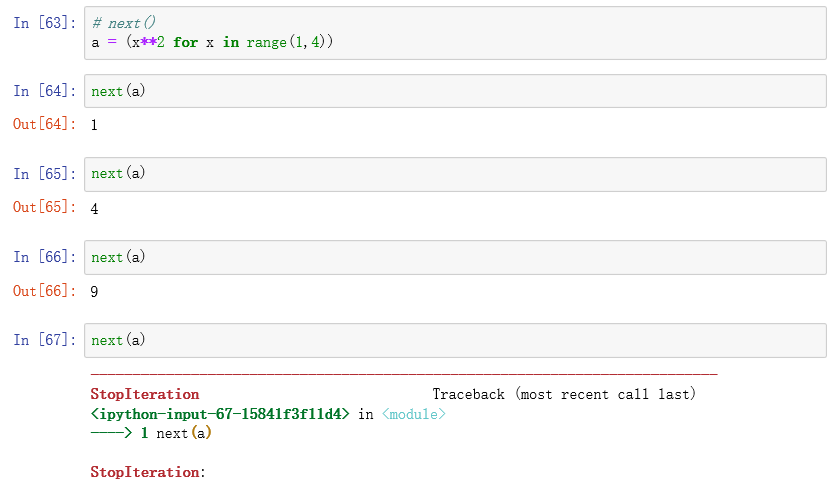


图28：next()使用示例

* + 1. **\_next\_()**

用生成器自身方法\_next()\_循环生成下一个值，示例如下：

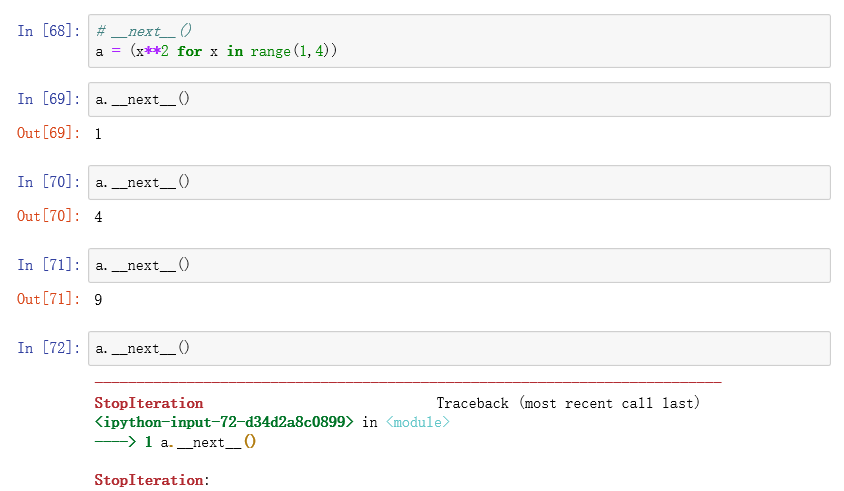


图29：\_next\_()使用示例

• 通常访问生成器元素的较常用的方法就是采用for循环，next()方法极少使用。因为采用for循环不需要关心StopIteration异常

• 内置函数next()和方法\_next\_()运行机制是相同的，因为内置函数next()实际上就是调用了生成器自身方法\_next\_()

• 生成器只能遍历一次，生成最后一个元素后，再次调用next()或\_next\_()会抛出StopIteration异常

* + 1. **next()与send()**

对于普通的生成器，第一个next调用，相当于启动生成器，会从生成器函数的第一行代码开始执行，直到第一次执行完yield语句（第4行）后，跳出生成器函数；然后第二个next调用，进入生成器函数后，从yield语句的下一句语句（第5行）开始执行，然后重新运行到yield语句，执行后，跳出生成器函数。对于这个参考示例如下：

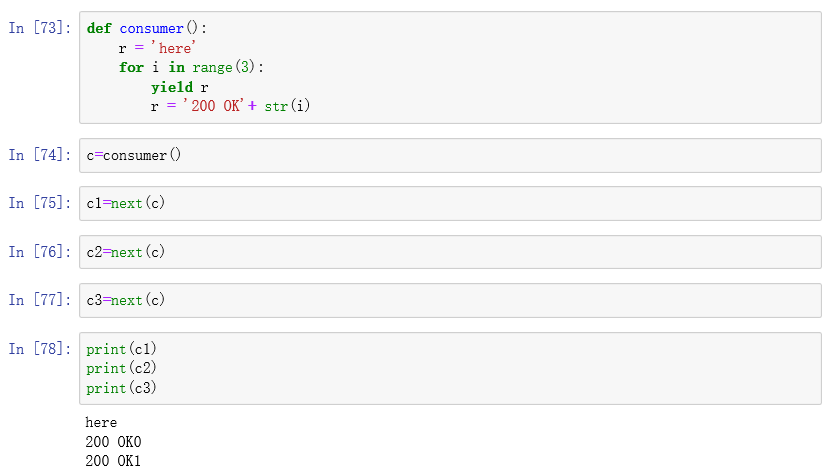


图30：普通生成器的next()调用示例

next()与send()区别是send()可以传递 yield 表达式的值进去，而next()不能传递特定的值，只能传递None进去。因此，我们可以看做c.next() 和 c.send(None) 作用是一样的。 需要提醒的是，第一次调用时，请使用next()语句或是send(None)，不能使用send发送一个非None的值，否则会出错的，因为没有Python yield语句来接收这个值

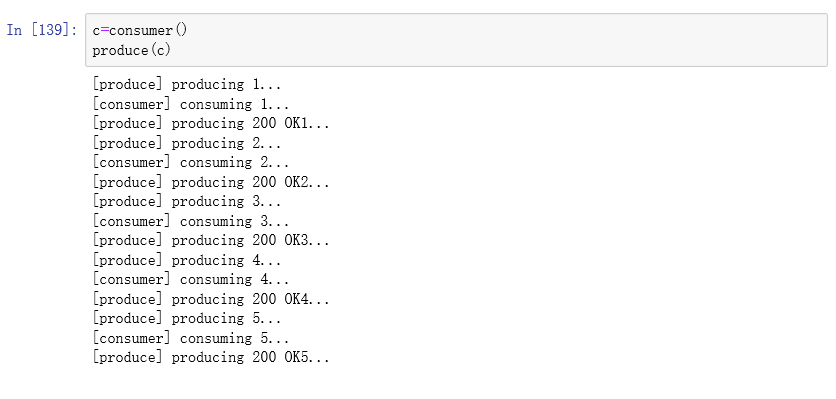


图31：send()使用示例

当第一次send（None）（对应11行）时，启动生成器，从生成器函数的第一行代码开始执行，直到第一次执行完yield（对应第4行）后，跳出生成器函数。这个过程中，n1一直没有定义。先进行 yield r 操作把 r 返回给aa，yield之后的操作（n1的赋值操作不进行） 下面运行到send（1）时，进入生成器函数，注意这里与调用next的不同。这里是从第4行开始执行，把1赋值给n1，但是并不执行yield部分。下面继续从yield的下一语句继续执行，然后重新运行到yield语句，执行后，跳出生成器函数。

即send和next相比，只是开始多了一次赋值的动作，其他运行流程是相同的。

* + 1. **throw和close**

throw有两方面的作用，首先是抛给生成器一个异常，然后如果生成器能处理掉异常的话，throw方法接着迭代一次取得返回值。



图32：throw()使用示例

close，他只有一个作用，就是向生成器抛出GeneratorExit异常。



图33：close()使用示例