Errors and Debugging

错误和调试

> Code development and data analysis always require a bit of trial and error, and IPython contains tools to streamline this process. This section will briefly cover some options for controlling Python's exception reporting, followed by exploring tools for debugging errors in code.
>
> 开发和数据分析通常都需要很多的试验，伴随着很多的错误。IPython包含着能够将这个过程串联起起来的工具。这一章节会简要介绍Python的异常控制，然后介绍在代码中调试的工具。

Controlling Exceptions: `%xmode`

异常控制：`%xmode`

> Most of the time when a Python script fails, it will raise an Exception. When the interpreter hits one of these exceptions, information about the cause of the error can be found in the traceback, which can be accessed from within Python. With the `%xmode` magic function, IPython allows you to control the amount of information printed when the exception is raised. Consider the following code:
>
> 大部分情况下如果Python脚本执行失败了，都是由于抛出了异常导致的。当解释器碰到了这些异常的时候，会将错误产生的原因压到当前程序执行的堆栈当中，您可以通过Python的traceback访问到这些信息。使用 %xmode 魔术指令，IPython允许你控制异常发生时错误信息的数量。看例子：

In [1]:
```
def func1(a, b):
    return a / b

def func2(x):
    a = x
    b = x - 1
    return func1(a, b)
```

In [2]:
```
func2(1)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-2-7cb408ea7ed1> in <module>
----> 1 func2(1)

<ipython-input-1-586ccabd0db3> in func2(x)
      5     a = x
      6     b = x - 1
----> 7     return func1(a, b)

<ipython-input-1-586ccabd0db3> in func1(a, b)
      1 def func1(a, b):
----> 2     return a / b
      3
      4 def func2(x):
      5     a = x

ZeroDivisionError: division by zero
```

> Calling `Func2` results in an error, and reading the printed trace lets us see exactly what happened. By default, this trace includes a few lines showing the context of each step that led to the error. Using the `%xmode` magic function (short for *Exception mode*), we can change what information is printed.
>
> 调用 Func2 会发生错误，Python解释器会使用默认方式打印出堆栈信息，通过查看这些信息，你可以检查程序发生了什么问题。默认情况下，打印出来的信息会包括很多行，每行会输出函数调用的情况。使用 %xmode 魔术指令（名称是*Exception mode*的缩写），我们可以修改打印的信息内容。

> `%xmode` takes a single argument, the mode, and there are three possibilities: `Plain`, `Context`, and `Verbose`. The default is `Context`, and gives output like that just shown before. `Plain` is more compact and gives less information:
>
> `%xmode` 需要一个参数，就是输出错误的模式，有三种选择： `Plain`， `Context` 和 `Verbose`，默认是 `Context`，该模式下的输出就如上面所见。 `Plain` 会更简短，提供更少的内容：

In [3]:
```
%xmode Plain
```

```
Exception reporting mode: Plain
```

In [4]:
```
func2(1)
```

```
Traceback (most recent call last):

  File "<ipython-input-4-7cb408ea7ed1>", line 1, in <module>
    func2(1)

  File "<ipython-input-1-586ccabd0db3>", line 7, in func2
    return func1(a, b)

  File "<ipython-input-1-586ccabd0db3>", line 2, in func1
    return a / b

ZeroDivisionError: division by zero
```

> The `Verbose` mode adds some extra information, including the arguments to any functions that are called:
>
> `Verbose` 模式会增加一些额外的信息，包括每个函数调用时候的参数值：

In [5]:
```
%xmode Verbose
```

```
Exception reporting mode: Verbose
```

In [6]:
```
func2(1)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-6-7cb408ea7ed1> in <module>
----> 1 func2(1)
        global func2 = <function func2 at 0x7fee38cf0d08>

<ipython-input-1-586ccabd0db3> in func2(x=1)
      5     a = x
      6     b = x - 1
----> 7     return func1(a, b)
        global func1 = <function func1 at 0x7fee38cf07b8>
        a = 1
        b = 0

<ipython-input-1-586ccabd0db3> in func1(a=1, b=0)
      1 def func1(a, b):
----> 2     return a / b
        a = 1
        b = 0
      3
      4 def func2(x):
      5     a = x

ZeroDivisionError: division by zero
```

> This extra information can help narrow-in on why the exception is being raised. So why not use the `Verbose` mode all the time? As code gets complicated, this kind of traceback can get extremely long. Depending on the context, sometimes the brevity of `Default` mode is easier to work with.
>
> 这些额外的信息能帮助你迅速定位到异常发生的原因。那么为什么我们不一直使用 Verbose 模式呢？如果你的代码变得复杂了之后，这种堆栈的输出会变得十分冗长。根据实际情况，有时候简短的默认模式可能更加适合查错。

Debugging: When Reading Tracebacks Is Not Enough

调试：当分析堆栈已经不足够了

> The standard Python tool for interactive debugging is `pdb`, the Python debugger. This debugger lets the user step through the code line by line in order to see what might be causing a more difficult error. The IPython-enhanced version of this is `ipdb`, the IPython debugger.
>
> 标准Python解释器有一个交互式的调试工具叫做 pdb，这个调试工具能让用户一行一行的执行代码，然后定位到更困难的错误原因。IPython增强版的调试器叫做 ipdb 。

> There are many ways to launch and use both these debuggers; we won't cover them fully here. Refer to the online documentation of these two utilities to learn more.
>
> 实际上存在着很多种方法来启动和使用这两个调试器；我们在这里不会完整的介绍它们。你可以参考这两个工具的在线文档来学习更多的内容。

> In IPython, perhaps the most convenient interface to debugging is the `%debug` magic command. If you call it after hitting an exception, it will automatically open an interactive debugging prompt at the point of the exception. The `ipdb` prompt lets you explore the current state of the stack, explore the available variables, and even run Python commands!
>
> 在IPython中，也许最简单的调试方式就是使用 %debug 魔术指令了。如果当你遇到一个异常之后调用它，IPython会自动打开一个交互式的调试提示符，并定位在异常发生的地方。 ipdb 提示符允许你查看当前的堆栈信息，显示变量和它们的值，甚至执行Python命令！

> Let's look at the most recent exception, then do some basic tasks–print the values of `a` and `b`, and type `quit` to quit the debugging session:
>
> 让我们查看最近发生的那个异常，然后执行一些基础的指令来打印变量 a 和 b 的值，最后使用 quit 退出调试模式：

In [7]:
```
%debug
```

```
> <ipython-input-1-586ccabd0db3>(2)func1()
      1 def func1(a, b):
----> 2     return a / b
      3
      4 def func2(x):
      5     a = x

ipdb> print(a)
1
ipdb> print(b)
0
ipdb> quit
```

> The interactive debugger allows much more than this, though–we can even step up and down through the stack and explore the values of variables there:
>
> 这个交互式的调试器允许我们做更多的操作，我们可以向上或向下浏览不同级别的堆栈，然后再查看那个层级的变量的内容：

In [8]:
```
%debug
```

```
> <ipython-input-1-586ccabd0db3>(2)func1()
      1 def func1(a, b):
----> 2     return a / b
      3
      4 def func2(x):
      5     a = x

ipdb> up
> <ipython-input-1-586ccabd0db3>(7)func2()
      3
      4 def func2(x):
      5     a = x
      6     b = x - 1
----> 7     return func1(a, b)

ipdb> print(x)
1
ipdb> up
> <ipython-input-6-7cb498ea7ed1>(1)<module>()
----> 1 func2(1)

ipdb> down
> <ipython-input-1-586ccabd0db3>(7)func2()
      3
      4 def func2(x):
      5     a = x
      6     b = x - 1
----> 7     return func1(a, b)

ipdb> quit
```

> This allows you to quickly find out not only what caused the error, but what function calls led up to the error.
>
> 这样仅仅能够让你迅速定位问题的原因，还能让你一直回溯到错误最上层的函数调用。

> If you'd like the debugger to launch automatically whenever an exception is raised, you can use the `%pdb` magic function to turn on this automatic behavior:
>
> 如果你希望调试器保持打开状态，每当发生异常时就自动启动，你可以使用 %pdb 魔术指令，使用 on / off 参数就能打开或关闭调试器的自动启动模式。

In [9]:
```
%xmode Plain
%pdb on
func2(1)
```

```
Exception reporting mode: Plain
Automatic pdb calling has been turned ON

Traceback (most recent call last):

  File "<ipython-input-9-f80f6b5cecf3>", line 3, in <module>
    func2(1)

  File "<ipython-input-1-586ccabd0db3>", line 7, in func2
    return func1(a, b)

  File "<ipython-input-1-586ccabd0db3>", line 2, in func1
    return a / b

ZeroDivisionError: division by zero

> <ipython-input-1-586ccabd0db3>(2)func1()
      1 def func1(a, b):
----> 2     return a / b
      3
      4 def func2(x):
      5     a = x

ipdb> print(b)
0
ipdb> quit
```

> Finally, if you have a script that you'd like to run from the beginning in interactive mode, you can run it with the command `%run -d`, and use the `next` command to step through the lines of code interactively.
>
> 最后，如果你有一个Python脚本文件，然后希望在IPython中交互式运行，并且打开调试器的话，你可以使用 %run -d 魔术指令来执行这个脚本，然后你还能在调试模式提示符下使用 next 命令来单步执行脚本中的代码。

Partial list of debugging commands

调试命令部分列表

> There are many more available commands for interactive debugging than we've listed here; the following table contains a description of some of the more common and useful ones:

Command	Description
`list`	Show the current location in the file
`h(elp)`	Show a list of commands, or find help on a specific command
`q(uit)`	Quit the debugger and the program
`c(ontinue)`	Quit the debugger, continue in the program
`n(ext)`	Go to the next step of the program
`<enter>`	Repeat the previous command
`p(rint)`	Print variables
`s(tep)`	Step into a subroutine
`r(eturn)`	Return out of a subroutine

> 除了下面列出来的最常用的命令和简单解释之外，还有很多由于篇幅原因未列出说明的调试命令。

调试命令	描述
`list`	显示当前在文件中的位置信息
`h(elp)`	查看帮助文档，可以显示列表，或查看某个命令的具体帮助信息
`q(uit)`	退出调试模式并退出程序
`c(ontinue)`	退出调试模式，继续执行代码
`n(ext)`	执行下一行代码，单步调试
`<enter>`	直接重复执行上一条命令
`p(rint)`	打印变量内容
`s(tep)`	跟踪进入子函数内部进行调试
`r(eturn)`	直接执行到函数返回

> For more information, use the `help` command in the debugger, or take a look at `ipdb`'s online documentation.
>
> 需要了解更多信息，可以在调试器模式下使用 help 命令，或者参见 ipdb 的在线文档。