



# Python基础教程

## 第7章 异常处理与模块



错误是语法或是逻辑上的。

(1) 语法错误指示软件的结构上有错误，导致不能被解释器解释或编译器无法编译，这样的错误必须在程序执行前纠正。

(2) 语法正确后，逻辑错误可能是由于不完整或是不合法的输入导致；逻辑无法生产、计算、或是输出结果需要的过程无法执行，通常称为域错误和范围错误。

(3) 当Python检测到一个错误时，解释器就会指出当前程序已经无法继续执行下去，这个时候就出现了异常。

程序出现了错误而在正常控制流以外采取的行为：引起异常发生的错误和检测（和采取可能的措施）阶段。

（1）只要检测到错误并且意识到异常条件，解释器会引发（触发、抛出或生成）一个异常。

（2）异常引发后，可以调用很多不同的操作。

- 忽略（修补后终止程序）

- 减轻、解决问题后设法继续执行程序

（3）异常处理

采用“尝试（**try**）”块和“捕获（**catching**）”块



异常即是一个事件，该事件在程序执行过程中发生，影响了程序的正常执行。

一般情况下，在**Python**无法正常处理程序时就会发生异常。

异常是**Python**对象，表示一个错误。当**Python**脚本发生异常时需要捕获处理它，否则程序会终止执行。



## 7.4 Python中的异常

在Python中出现异常，会看到“跟踪记录（**traceback**）”消息以及随后解释器向你提供的信息，包括错误的名称、原因和发生错误的行号。

**NameError: 尝试访问一个未声明的变量**

**ZeroDivisionError: 除数为零**

```
>>> foo
```

```
>>> 1/0
```

```
Traceback (most recent call last):
```

```
File "<pyshell#1>", line 1, in <module>
```

```
foo
```

```
NameError: name 'foo' is not defined
```

```
Traceback (most recent call last):
```

```
File "<pyshell#2>", line 1, in <module>
```

```
1/0
```

```
ZeroDivisionError: integer division or modulo by zero
```

**IndexError: 请求的索引超出序列范围**

```
>>> aList=[]
```

```
>>> aList[0]
```

```
Traceback (most recent call last):
```

```
File "<pyshell#5>", line 1, in <module>
```

```
aList[0]
```

```
IndexError: list index out of range
```





## 7.4.1 try-except异常处理实例

没有异常处理，程序出错便停止运行：

```
>>> a = 10 + '10'
```

Traceback (most recent call last):

File "<pyshell#0>", line 1, in <module>

a = 10 + '10'

TypeError: unsupported operand type(s) for +: 'int' and 'str'



检测异常使用**try**语句。任何在**try**语句块里的代码都会被监测，检查有无异常发生。

**try**语句有两种主要形式：

(1) **try-except**。

(2) **try-finally**。

这两个语句是互斥的；

一个**try**语句可以对用一个或多个**except**子句

一个**try**语句只能对应一个**finally**子句

可以有 **try-except-finally**复合语句。

- 在程序运行时，解释器尝试执行**try**块里所有的代码，如果代码块完成后没有异常发生，执行流就会忽略**except**语句继续执行。
- 当**except**语句发现指定的异常发生后，程序保存错误的原因，控制流立即跳转到对应的处理器，**try**子句的剩余部分永远也不会执行。





**try-except**语句用来检测**try**语句块中的错误，而让**except**语句捕获异常信息并处理。

如果你不想在异常发生时结束你的程序，只需在**try**里捕获它。也可以添加一个可选的**else**字句处理没有探测到异常的执行的代码。

**try:**

*try\_suite*

# 监控这里的异常

**except Except[, reason]:**

*except\_suite* # 异常处理代码





## 7.4.4 try-except-包装内建函数

```
>>> float(12345)
```

```
12345.0
```

```
>>> float('12345')
```

```
12345.0
```

```
>>> float('123.45e67')
```

```
1.2345e+69
```

```
>>> float('foo')
```

```
Traceback (most recent call last):
```

```
File "<pyshell#4>", line 1, in <module>
```

```
float('foo')
```

```
ValueError: could not convert string to float: 'foo'
```

```
>>> float(['this is', 1, 'list'])
```

```
Traceback (most recent call last):
```

```
File "<pyshell#5>", line 1, in <module>
```

```
float(['this is', 1, 'list'])
```

```
TypeError: float() argument must be a string or a number, not 'list'
```



## 7.4.4 try-except-包装内建函数

```
>>> def safe_float1(obj):  
    try:  
        return float(obj)  
    except ValueError:  
        pass
```

`pass`, 表示不进行任何处理, 忽略这个错误。

```
>>> safe_float1('foo')  
>>> safe_float1(['this is', 1, 'list'])
```

Traceback (most recent call last):

File "<pyshell#13>", line 1, in <module>

safe\_float1(['this is', 1, 'list'])

File "<pyshell#11>", line 3, in safe\_float1

return float(obj)

TypeError: float() argument must be a string or a number, not 'list'





```
>>> def safe_float2(v):
```

```
    try:
```

```
        return float(v)
```

```
    except ValueError as err:
```

```
        return err
```

```
>>> safe_float2('f')
```

```
ValueError("could not convert string to float: 'f'")
```

得到了出错的提示!





## 7.4.5 try-except (带多个except语句)

可以把多个except语句连接在一起，处理一个try块中可能发生的多种异常。

*try:*

*try\_suite*

# 监控这里的异常

*Except Except1[ as reason1]:*

*except\_suite1* # 异常处理代码1

*Except Except2[ as reason2]:*

*except\_suite2* # 异常处理代码2

...

同样，首先尝试执行try子句，如果没有错误，忽略所有的except从句继续执行。





```
>>> def safe_float3(v):  
    try:  
        ret = float(v)  
    except ValueError as err:  
        ret = err  
    except TypeError as err:  
        ret = err  
    return ret
```

```
>>> safe_float3( 'Hello')  
ValueError("could not convert string to float: 'Hello'")  
>>> safe_float3( [1, 2])  
TypeError("float() argument must be a string or a number, not 'list'")  
>>> safe_float3( '23.4' )  
23.4
```



## 7.4.6 try-except (处理多个except语句)

可以在一个except子句里处理多个异常。

**注意：** 要求异常必须放在一个元组里面。

*try:*

*try\_suite*

# 监控这里的异常

*except (Except1 , Except2 )[ as reason]:*

*except\_suite\_for\_exceptions\_Exc1\_to\_ExcN* # 异常处理代码



```
>>> def safe_float4(v):
```

```
    try:
```

```
        ret = float(v)
```

```
    except ( ValueError, TypeError ) as err:
```

```
        ret = err
```

```
    return ret
```

```
>>> safe_float4( 'Hello' )
```

```
ValueError("could not convert string to float: 'Hello'")
```

```
>>> safe_float4( [1, 2] )
```

```
TypeError("float() argument must be a string or a number, not 'list'")
```

```
>>> safe_float4( '23.4' )
```

```
23.4
```





如果想要捕获所有的异常呢？

解决办法：使用顶层异常类。

如果查询异常继承的树结构，会发现**Exception**在最顶层。

*try:*

*try\_suite*                      # 监控这里的异常

*except Exception:*

*except\_suite\_for\_exceptions\_Exc1\_to\_ExcN* # 异常处理代码



## 7.4.7 try-except Exception

```
>>> def safe_float5( obj ):
    try:
        retval = float( obj )
    except Exception:
        retval = "could not convert non-number to float"
    return retval

>>> safe_float5( 'foo' )
'could not convert non-number to float'
>>> safe_float5( [' this is ', 1 , ' list ' ] )
'could not convert non-number to float'
```





```
>>> from random import randint
>>> num = randint(1, 10)
>>> while True:
```

```
    guess = int(input('输入数字 1~10: '))
    if guess > num:
        print('大了')
    elif guess < num:
        print('小了')
    else:
        print('恭喜, 猜对了!')
    break
```

```
>>> while True:
    guess = int(input('输入数字 1~10: '))
    if guess > num:
        print('大了')
    elif guess < num:
        print('小了')
    else:
        print('恭喜, 猜对了!')
    break
```

输入数字 1~10: 6

Traceback (most recent call last):

File "<pyshell#14>", line 1, in <module>

guess = int(input('输入数字 1~10: '))

ValueError: invalid literal for int() with base 10: '6'

输入数字 1~10: 13

大了

>>> num

6



```
from random import randint
num = randint(1,10)
while True:
    try:
        guess = int(input('輸入數字1~10: '))
    except:
        print('輸入不正確! 請輸入數字1~10')
        continue
    if guess > num:
        print('大了')
    ...
```





## 7.5 try-except-else-finally

*try:*

*A*

*except Myexception:*

*B* (1) 最少有一个**except**语句;

*else:* (2) **else**和**finally**都是可选的;

*C* (3) 无论异常发生在**A**、**B**和/或**C**都将执行**finally**块。

*finally:*

*D*



## 7.5.1 try-except工作原理

当开始一个try语句后，Python就在当前程序的上下文中作标记，这样当异常出现时就可以回到这里，try子句先执行，接下来会发生什么依赖于执行时是否出现异常。

- 如果当try后的语句执行时发生异常，Python就跳回到try并执行第一个匹配该异常的except子句，异常处理完毕，控制流就通过整个try语句（除非在处理异常时又引发新的异常）。

## 7.5.1 try-except工作原理

- 如果在try后的语句里发生了异常，却没有匹配的except子句，异常将被递交到上层的try，或者到程序的最上层（这样将结束程序，并打印缺省的出错信息）。
- 如果在try子句执行时没有发生异常，Python将执行else语句后的语句（如果有else的话），然后控制流通过整个try语句。



## 7.6 try-except-else-finally

有异常处理，程序出错可以继续运行：

7-1.py - C:/Users/reebox/Desktop/7-1.py (3.7.4)

File Edit Format Run Options Window Help

```
try:
    a = 10 + '10'
except Exception as e:
    print( e )
    print( 'Error!' )
finally:
    print('Always Excuted!')
```

===== RESTART: C:/Users/reebox/Desktop/7-1.py

=====

unsupported operand type(s) for +: 'int' and 'str'

Error!

Always Excuted!

Excuted





## 7.7 使用except而不帶任何異常類型

出現異常：將會執行except語句，且不執行else語句；

未出現異常：不執行except語句，會執行else語句；

= RESTART: C:/Users/reebox/AppData/Local/Programs/Python/Python37-32/7-2.py =  
Error!

7-2.py - C:/Users/reebox/AppData/Local/Programs/Python/Python37-32/7-2.py (3.7.4)

File Edit Format Run Options Window Help

Else Suite

```
try:
    a = 10 + '10'
except:
    print('Error!')
else:
    print('Else Suite!')
print('-'*60)
```

```
try:
    a = 10 + 10
except:
    print('Erro!')
else:
    print('Else Suite')
```



## 7.8 使用except而带多种异常类型

有多个类型的异常处理，异常发生时进入匹配的相应except语句，没有匹配的异常处理则使用不带类型的except。无论是否发生异常finally语句始终会执行。

7-3.py - C:/Users/reebox/AppData/Local/Programs/Python/Python37-32/7-3.py (3.7.4)  
File Edit Format Run Options Window Help

```
try:  
    a = 10 + '10'  
except TypeError as e:  
    print ( e )
```

```
= RESTART: C:/Users/reebox/AppData/Local/Programs/Python/Python37-32/7-3.py =  
unsupported operand type(s) for +: 'int' and 'str'  
TypeError!  
finally will always Excuted!  
Excuted!
```

```
    print ( e )  
    print ( 'except!' )  
finally:  
    print( 'finally will always Excuted!' )  
  
print( 'Excuted!' )
```

## 7.9 异常的参数 (1)

要想访问提供的异常原因，必须保留一个变量来保存这个参数。  
把这个参数放在**except**语句后，接在要处理的异常后面。

**# single exception**

*except Exception[, reason]:*

*suite\_for\_Exception\_with\_Argument*

**# multiple exceptions**

*except (Exception1, ..., ExceptionN)[as reason]:*

*suite\_for\_Exception1\_to\_ExceptionN\_with\_Argument*

传递异常参数：  
使用as语句

## 7.9 异常的参数 (2)

一个异常可以带上参数，可作为输出的异常信息参数。可以通过**except**语句来捕获异常的参数。变量接收的异常值通常包含在异常的语句中。在元组的表单中变量可以接收一个或者多个值。元组通常包含错误字符串，错误数字，错误位置。

\*7-5.py - C:/Users/reebox/AppData/Local/Programs/Python/Python37-32/7-5.py (3.7.4)\*

File Edit Format Run Options Window Help

# 使用as传递参数

try:

a = 10 + '10'

except TypeError as e:

print ( e )

print ( 'TypeError!' )

= RESTART: C:/Users/reebox/AppData/Local/Programs/Python/Python37-32/7-5.py =  
unsupported operand type(s) for +: 'int' and 'str'  
TypeError!

可以使用**raise**语句主动触发异常, **raise**语法格式如下:

```
raise [Exception [, args [, traceback]]]
```

语句中**Exception**是异常的类型（例如, **NameError**），参数是一个异常参数值。该参数是可选的，如果不提供，异常的参数是**"None"**。

最后一个参数是可选的（在实践中很少使用），如果存在，是跟踪异常对象。

一个异常可以是一个字符串，类或对象。**Python**的内核提供的异常，大多数都是实例化的类。为了能够捕获异常，**"except"**语句必须有用相同的异常来抛出类对象或者字符串。

抛出异常, 异常的类型为**Exception**。

```
>>> def error1(level):  
...     if level<1:  
...         raise Exception("Invalid level!")  
...  
>>> error1(0)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
    File "<stdin>", line 3, in error1  
Exception: Invalid level!
```

通过创建一个新的异常类，程序可以命名自己的异常。异常应该是典型的继承自 **Exception** 类，通过直接或间接的方式。

```
>>> class NetworkError( RuntimeError ):
    def __init__( self, arg ):
        self.args = arg
```

触发该异常：

```
>>> raise NetworkError( 'Bad hostname' )
```

```
Traceback (most recent call last):
```

```
File "<pyshell#8>", line 1, in <module>
```

```
    raise NetworkError( 'Bad hostname' )
```

```
NetworkError: ('B', 'a', 'd', ' ', 'h', 'o', 's', 't', 'n', 'a', 'm', 'e')
```

```
>>> raise NetworkError( 'IP Address is unreachable!' )
```

```
Traceback (most recent call last):
```

```
File "<pyshell#9>", line 1, in <module>
```

```
    raise NetworkError( 'IP Address is unreachable!' )
```

```
NetworkError: ('I', 'P', ' ', 'A', 'd', 'd', 'r', 'e', 's', 's', ' ', 'i', 's', ' ', 'u', 'n', 'r', 'e', 'a', 'c', 'h', 'a', 'b', 'l', 'e', '!')
```



断言是一句必须等价于布尔真的判定；此外，发生异常也意味着表达式为假。

断言语句等价于这样的Python表达式：

*assert expression[, arguments ]*

```
>>> assert 1==1  
>>> assert 1==0
```

```
Traceback (most recent call last):  
  File "<pyshell#65>", line 1, in <module>  
    assert 1==0  
AssertionError
```

也可以提供一个参数：

```
>>> assert 1==0, 'One does not equal zero silly!'
```

```
Traceback (most recent call last):  
  File "<pyshell#66>", line 1, in <module>  
    assert 1==0, 'One does not equal zero silly!'  
AssertionError: One does not equal zero silly!
```



- 使用不存在的字典关键字：引发 **KeyError** 异常。
- 搜索列表中不存在的值：引发 **ValueError** 异常。
- 尝试访问未知的对象属性：引发 **AttributeError** 异常。
- 引用不存在的变量：引发 **NameError** 异常。
- 未强制转换就混用数据类型：引发 **TypeError** 异常。
- 使用错误的编码解码类型：引发 **UnicodeError** 异常。
- 输入/输出错误：引发 **IOError** 异常
- 请求的索引超出序列范围：引发 **IndexError** 异常
- **Python**解释器语法错误：引发 **SyntaxError** 异常
- ...



另一种获取异常信息的途径是通过sys模块中的exc\_info()函数，此功能提供了一个3元组的信息。

```
>>> try:
    float( 'abc123' )
except:
    import sys
    exc_tuple = sys.exc_info()

>>> print(exc_tuple)
(<class 'ValueError'>, ValueError("could not convert string to float: 'abc123'"), <traceback object at 0x0302B828>)
```



- 错误无法避免，**try-except**的作用是提供一个可以提示错误或处理错误的机制，而不是一个错误过滤器：

**# this is really bad code**

**try:**

***Large\_block\_of\_code***      # 大段代码的“绷带”

***except Exception:***      # 与**except:** 相同

***pass***

**注意：**避免将大片的代码装入**try-except**中，然后使用**pass**忽略掉错误。可以捕获所有的异常，通过异常类实现，但不推荐！

## 为什么要用异常?

目的：减少程序出错的次数并在出错后仍能保证程序正常执行。

- 1、定位错误的位置
- 2、提供用户级别的错误信息
- 3、减小异常对业务的影响

模块是指相互间有一点联系并且有组织的代码片段：

➤ Python使用**import**关键字“调用”一个模块

➤ 利用之前的工作成果，实现代码重用

使用其他模块的属性和功能的操作叫做导入（**import**）。

**Python**数学运算类模块：**math**

**import math**

**X = math.fabs(x)**

## 8.1 Python模块-模块名称

每个模块都定义了自己唯一的名称空间

给定一个模块名，只可能有一个模块被导入到Python解释器

例：在自己定义的模块Test.py中创建一个add()函数，那么它的名字是

**Test.add()**

通过句点属性指定的完整授权名称防止了名称冲突的发生。

## 8.2 Python模块-导入模块格式

### 一、**import** 语句

➤ 多行导入

```
import module1
```

```
import module2
```

```
...
```

➤ 一行导入多个模块

```
import module1, module2, ...
```

### 二、**from-import**语句 导入指定的模块属性

```
from module import name1, name2...
```



## 8.3 Python模块-导入过程

解释器执行到导入语句，如果搜索找到指定的模块，就会加载它，没有找到指定的模块，程序报错：

```
>>> import x
```

```
Traceback (most recent call last):
```

```
File "<pyshell#7>", line 1, in <module>
```

```
import x
```

```
ModuleNotFoundError: No module named 'x'
```

|

导入过程遵循作用域原则，如果在程序的顶层导入，那么它的作用域就是全局的；在函数中导入，作用域是局部的。





包是一个有层次的文件目录结构，它定义了一个由模块和子包组成的**Python**应用程序执行环境。

- 允许程序员把有联系的模块组合到一起；
- 允许分发者使用目录结构而不是一大群混乱的文件；
- 帮助解决有冲突的模块名称；
- 使用句点属性标识来访问它们的元素；
- 使用标准的**import**和**from-import**语句导入包中的模块。

## 8.4.1 Python包-目录结构

假定包有如下的目录结构：**Phone**是最顶层的包

**Voicedta**、**Fax**、**Mobile**是它的子包

```
Phone/  
    __init__.py  
    common_util.py  
    Voicedta/  
        __init__.py  
        Pots.py  
        Isdn.py  
    Fax/  
        __init__.py  
        G3.py  
    Mobile/  
        __init__.py  
        Analog.py  
        Digital.py
```

导入子包中的模块：

```
import Phone.Mobile.Analog  
Phone.Mobile.Analog.dial()
```

也可以**from-import** 实现不同需求的导入。

```
from Phone import Mobile  
Mobile.Analog.dial( '555-1212' )  
from Phone.Mobile import Analog  
Analog.dial( '555-1212' )  
from Phone.Mobile.Analog import dial  
dial( '555-1212' )
```

在每个包模块文件所在的目录下，有\_\_init\_\_.py文件，用来告诉Python解释器该目录是一个包。

包同样支持from-import all语句

**from package.module import \***



## 8.4.3 导入模块和包的查找路径顺序

一个模块被导入时，按照以下搜索路径的顺序搜索相关模块：

➤ 当前目录

➤ Python环境变量`path`所指的目录列表

➤ Python解释器的安装目录



## 8.4.4 导入模块和包的查找路径

**Python**解释器借助**sys.path**变量中包含的路径来搜索模块，打印

**sys.path**路径：

```
>>> import sys  
>>> sys.path
```

在F盘下建一个Python模块TestModule.py，内容如下：

```
print ("Hello TestModule!")
```

在交互模式下执行导入：

```
import TestModule
```

报错是预料中的，因为TestModule.py所在的F盘并不在Python模块的搜索路径中。

如何解决？



方法一：

需要动态的增加这个路径到搜索模块中，再执行导入。

```
>>> import sys
>>> sys.path.append("F:\\")
>>> import TestModule
Hello TestModule!
```

方法二：

把文件拷贝到当前工作路径（**cwd**），或修改**cwd**

```
>>> import os
>>> os.getcwd()
'C:\\Users\\reebox\\AppData\\Local\\Programs\\Python\\Python37-32'
>>> os.chdir("F:\\")
```



- 注释

注释可以帮助代码阅读者尽快的读懂程序，同时注释应兼备简洁明了和准确的特点；

- 文档

**Python**提供一种机制，可以通过\_\_doc\_\_动态获取文档字符串；

- 缩进

缩进对齐可以提高代码阅读效率。**Python**较多的使用4个空格的风格；

- 选择标识符名称

应该为变量选择短而意义丰富的标示符，这个原则同样适用于模块的命名。





## 8.5.1 模块结构和布局

### 合理的布局样式

- # (1) 起始行 (Unix)
- # (2) 模块文档
- # (3) 模块导入
- # (4) 变量定义
- # (5) 类定义
- # (6) 函数定义
- # (7) 主程序

```
#!/usr/bin/env python
```

(1) 起始行

```
"this is a test module"
```

(2) 模块文档 (文档字符串)

```
import sys  
import os
```

(3) 模块导入

```
debug = True
```

(4) (全局) 变量定义

```
class FooClass (object):  
    "Foo class"  
    pass
```

(5) 类定义 (若有)

```
def test():  
    "test function"  
    foo = FooClass()  
    if debug:  
        print 'ran test()'
```

(6) 函数定义 (若有)

```
if __name__ == '__main__':  
    test()
```

(7) 主程序



### (1) 起始行

通常只有在类**Unix**环境下才使用起始行，有起始行就能够仅输入脚本名字来执行脚本，无需直接调用解释器。

### (2) 模块文档

简要介绍模块的功能，模块外可通过**module.\_\_doc\_\_**访问这些内容。

### (3) 模块导入

导入当前模块的代码需要的所有模块；

每个模块仅导入一次（当前模块被加载时）；

函数内部的模块导入代码不会被执行，除非本函数正在执行。

### (4) 变量定义

全局变量，本模块中的所有函数都可直接使用；

要尽量使用局部变量代替全局变量。

### (5) 类定义语句

所有的类都需要在这里定义。

当模块被导入时`class`语句会被执行，类也就会被定义。

### (6) 函数定义语句

此处定义的函数可以通过`module.function()`在外部被访问到，当模块被导入时`def`语句会被执行。函数也就会定义好，函数的文档变量是`function.__doc__`。

### (7) 主程序

无论这个模块是被别的模块导入还是作为脚本执行，都会执行这部分代码。很多项目都是一个主程序，由它导入所有需要的模块。

所以，绝大部分模块的创建就是为了被其它模块调用，只有包含主程序的模块会被直接执行。

### (7) 主程序

- 1) 在Python中，那些没有缩进的代码行在模块被导入时就会执行，不管是不是真的需要执行；
  - 2) 比较安全的写代码的方式就是除了那些真正需要执行的代码（顶级可执行代码）以外，几乎所有的功能代码都应该封装在函数当中；
  - 3) 主程序中的代码通常包括变量赋值、类定义和函数定义；
  - 4) 检查\_\_name\_\_来决定是否调用另一个函数（通常调用main()）
- 函数，可放置测试代码；

### 主程序

5) 运行时检测该模块是被导入还是被直接执行;

- 如果模块是被导入, \_\_name\_\_的值为模块名字
- 如果模块是被直接执行, \_\_name\_\_的值为‘\_\_main\_\_’,

6) 通常不会有太多功能性代码, 根据执行的模式调用不同的函数。

### (8) 测试代码

利用 `__name__` 变量，将测试代码放到 `main()` 或 `test()`（或其他自定义函数名）中，如果该模块被当成脚本运行，就调用这个函数。

仅当该文件被直接执行时运行，这些测试代码应该随着测试条件及测试结果的变更及时修改，每次代码更新都应该运行这些测试代码，以确认修改没有引发新问题。

在主程序中放置测试代码是测试模块的简单快捷的手段。





### (8) 测试代码

```
1 #Mymodule.py
2 def main( ):
3     print( ' we are in %s ' % __name__ )
4 if __name__ == '__main__':
5     main( )
```





### (8) 直接运行

Mymodule.py - C:/Users/reebox/AppData/Local/Programs/Python/Python37-32/Mymodule.py (3.7.4)

File Edit Format Run Options Window Help

```
#Mymodule.py
```

```
def main():
```

```
    print('we are in %s' % __name__)
```

```
if __name__ == '__main__':
```

```
    main()
```

Python 3.7.4 Shell

File Edit Shell Debug Options Window Help

Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32

Type "help", "copyright", "credits" or "license()" for more information.

>>>

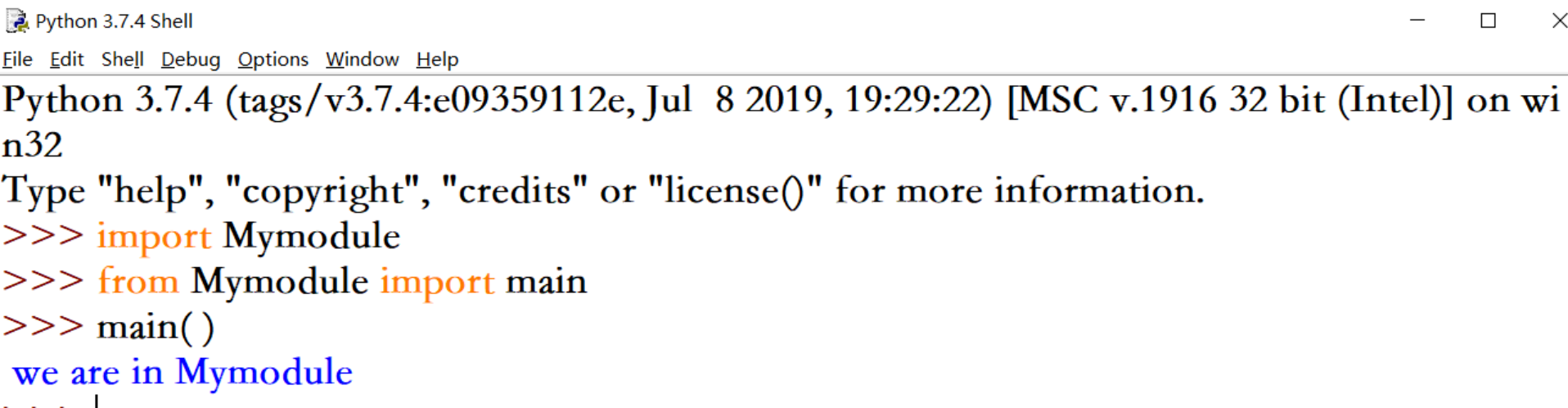
RESTART: C:/Users/reebox/AppData/Local/Programs/Python/Python37-32/Mymodule.py

py

we are in \_\_main\_\_



## (8) 作为模块导入



```
Python 3.7.4 Shell
File Edit Shell Debug Options Window Help
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import Mymodule
>>> from Mymodule import main
>>> main()
we are in Mymodule
... |
```



## 思考与练习

- (1) 如何创建异常？
- (2) 如何在模块文件中创建自己的测试代码？

