Python函数

一、课前准备

1. 根据课程大纲复习Python基础语法及数据结构；
2. 根据课程大纲提前预习Python函数；

二、课堂主题

本小节主要学习Python的基础中的函数的创建和调用以及参数的传递。

三、课堂目标

1. 掌握Python中函数的创建和调用。
2. 对函数的调用顺序以及参数的使用。
3. 区分全局变量与局部变量的使用形式。

四、知识要点

1、函数的简介

在程序中，如果实现了某个功能的代码需要多次使用，就把**该段代码块组织为一个小模块，这就是函数。**函数可以提高编写的效率以及代码的重用。在前面课程中，我们已经接触了Python中得内建函数，比如print()、input()。也可以自己创建函数，这被叫做用户自定义函数。

例如：我们知道圆的面积计算公式为：`S = πr2`

当我们知道半径r的值时，就可以根据公式计算出面积。假设我们需要计算3个不同大小的圆的面积：

```
r1 = 12.34
r2 = 9.08
r3 = 73.1
s1 = 3.14 * r1 * r1
s2 = 3.14 * r2 * r2
s3 = 3.14 * r3 * r3
```

当代码出现有规律的**重复**的时候，你就需要当心了，每次写`3.14 * x * x`不仅很麻烦，而且，如果要把3.14改成`3.14159265359`的时候，得全部替换。

有了函数，我们就不再每次写`s = 3.14 * x * x`，而是写成更有意义的函数调用`s = area_of_circle(x)`，而函数`area_of_circle`本身只需要写一次，就可以多次调用。

基本上所有的高级语言都支持函数，Python也不例外。Python不但能非常灵活地定义函数，而且本身内置了很多有用的函数，可以直接调用。

2、函数定义和调用

函数必须先定义，后调用。

(1) 定义函数

定义函数的格式如下：

```
def 函数名():
    代码（代码前有缩进,tab）

# 定义函数，完成打印信息的功能
def printInfo():
    print('************************************')
    print('          Python数据分析')
    print('************************************')
```

(2) 调用函数

定义函数之后，就相当于完成了某个功能的代码，想要让代码能够执行，需要调用函数。

调用函数很简单的，通过 **函数名()** 即可完成调用

```
# 定义函数后，函数是不能够自动执行的，需要调用函数，函数才能执行
printInfo()
```

(3) 注意:

- 调用函数时，函数会从头开始执行，当函数中的代码执行完毕后，则函数调用结束
- 函数中如果存在return语句，执行到return语句时，函数调用结束

(4) 函数的文档说明

- help(函数名)能够看到test函数的相关说明

例如: help(printInfo)

- `test.__doc__` 直接查看文档说明

  `print(printInfo.__doc__)`

3、函数的参数

Python中函数定义非常简单，由于函数参数的存在，使函数变得非常灵活应用广泛，不但使得函数能够处理复杂多变的参数，还能简化函数的调用。Python中的函数参数有如下几种：位置参数、默认参数、可变参数、关键字参数；

3.1 位置参数

计算 `x^2` 的函数：

```
def power(x):
    return x * x
```

对于 `power(x)` 函数，参数x就是一个**位置参数**，也叫做**必选参数。**

当我们调用 `power` 函数时，必须传入有且仅有的一个参数x：

```
>>> power(5)
25
>>> power(15)
225
```

现在，如果我们要计算 `x^3` 怎么办？可以再定义一个 `power3` 函数，但是如果要计算 `x^4` 、 `x^5` ..... 怎么办？我们不可能定义无限多个函数。

你也许想到了，可以把 `power(x)` 修改为 `power(x, n)` ，用来计算 `x^n` ，说干就干：

```
def power(x, n):
    s = 1
    while n > 0:
        n = n - 1
        s = s * x
    return s
```

对于这个修改后的 `power(x, n)` 函数，可以计算任意n次方：

```
>>> power(5, 2)
25
>>> power(5, 3)
125
```

修改后的`power(x, n)`函数有两个参数：x和n，这两个参数都是位置参数，调用函数时，传入的两个值按照位置顺序依次赋给参数`x`和`n`。

3.2 默认参数

新的`power(x, n)`函数定义没有问题，但是，旧的调用代码失败了，原因是我们增加了一个参数，导致旧的代码因为缺少一个参数而无法正常调用：

```
>>> power(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: power() missing 1 required positional argument: 'n'
```

Python的错误信息很明确：调用函数`power()`缺少了一个位置参数`n`。

这个时候，默认参数就排上用场了。由于我们经常计算`x^2`，所以，完全可以把第二个参数`n`的默认值设定为2：

```
def power(x, n=2):
    s = 1
    while n > 0:
        n = n - 1
        s = s * x
    return s
```

这样，当我们调用`power(5)`时，相当于调用`power(5, 2)`：

```
>>> power(5)
25
>>> power(5, 2)
25
```

而对于`n > 2`的其他情况，就必须明确地传入`n`，比如`power(5, 3)`。

从上面的例子可以看出，默认参数**可以简化函数的调用**。设置默认参数时，有几点要注意：

- 一是**必选参数在前，默认参数在后**，否则Python的解释器会报错；
- 二是当函数有多个参数时，把变化大的参数放前面，变化小的参数放后面。变化小的参数就可以作为默认参数。

使用默认参数有什么好处?

举个例子，我们写个一年级小学生注册的函数，需要传入`name`和`gender`两个参数：

```
def enroll(name, gender):
    print('name:', name)
    print('gender:', gender)
```

这样，调用`enroll()`函数只需要传入两个参数：

```
>>> enroll('Sarah', 'F')
name: Sarah
gender: F
```

如果要继续传入年龄、城市等信息怎么办？这样会使得调用函数的复杂度大大增加。

我们可以把年龄和城市设为默认参数：

```
def enroll(name, gender, age=6, city='Beijing'):
    print('name:', name)
    print('gender:', gender)
    print('age:', age)
    print('city:', city)
```

这样，大多数学生注册时不需要提供年龄和城市，只提供必须的两个参数：

```
>>> enroll('Sarah', 'F')
name: Sarah
gender: F
age: 6
city: Beijing
```

只有与默认参数不符的学生才需要提供额外的信息：

```
enroll('Bob', 'M', 7)
enroll('Adam', 'M', city='Tianjin')
```

可见，默认参数**降低了函数调用的难度**，而一旦需要更复杂的调用时，又可以传递更多的参数来实现。无论是简单调用还是复杂调用，函数只需要定义一个。

有多个默认参数时，调用的时候，既可以按顺序提供默认参数，比如调用`enroll('Bob', 'M', 7)`，意思是，除了`name`，`gender`这两个参数外，最后1个参数应用在参数`age`上，`city`参数由于没有提供，仍然使用默认值。

也可以不按顺序提供部分默认参数。当不按顺序提供部分默认参数时，需要把参数名写上。比如调用`enroll('Adam', 'M', city='Tianjin')`，意思是，`city`参数用传进去的值，其他默认参数继续使用默认值。

注意：默认参数有个最大的坑，演示如下：

先定义一个函数，传入一个list，添加一个`END`再返回：

```
def add_end(L=[]):
    L.append('END')
    return L
```

当你正常调用时，结果似乎不错：

```
>>> add_end([1, 2, 3])
[1, 2, 3, 'END']
>>> add_end(['x', 'y', 'z'])
['x', 'y', 'z', 'END']
```

当你使用默认参数调用时，一开始结果也是对的：

```
>>> add_end()
['END']
```

但是，再次调用`add_end()`时，结果就不对了：

```
>>> add_end()
['END', 'END']
>>> add_end()
['END', 'END', 'END']
```

很多初学者很疑惑，默认参数是`[]`，但是函数似乎每次都“记住了”上次添加了`'END'`后的list。

原因解释如下：

Python函数在定义的时候，默认参数`L`的值就被计算出来了，即`[]`，因为默认参数`L`也是一个变量，它指向对象`[]`，每次调用该函数，如果改变了`L`的内容，则下次调用时，默认参数的内容就变了，不再是函数定义时的`[]`了。

定义默认参数要牢记一点：默认参数必须指向不变对象！

要修改上面的例子，我们可以用`None`这个不变对象来实现：

```
def add_end(L=None):
    if L is None:
        L = []
    L.append('END')
    return L
```

现在，无论调用多少次，都不会有问题：

```
>>> add_end()
['END']
>>> add_end()
['END']
```

为什么要设计`str`、`None`这样的不变对象呢？因为不变对象一旦创建，对象内部的数据就不能修改，这样就减少了由于修改数据导致的错误。此外，由于对象不变，多任务环境下同时读取对象不需要加锁，同时读一点问题都没有。我们在编写程序时，如果可以设计一个不变对象，那就尽量设计成不变对象。

3.3 可变参数

在Python函数中，还可以定义可变参数。顾名思义，可变参数就是**传入的参数个数是可变的**，可以是1个、2个到任意个，还可以是0个。

我们以数学题为例子，给定一组数字`a，b，c……`，请计算a^2 + b^2 + c^2 +....。

要定义出这个函数，我们必须**确定输入的参数**。由于参数个数不确定，我们首先想到可以把a，b，c……作为一个`list`或`tuple`传进来，这样，函数可以定义如下：

```
def calc(numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

但是调用的时候，需要先组装出一个`list`或`tuple`：

```
>>> calc([1, 2, 3])
14
>>> calc((1, 3, 5, 7))
84
```

如果利用可变参数，调用函数的方式可以简化成这样：

```
>>> calc(1, 2, 3)
14
>>> calc(1, 3, 5, 7)
84
```

所以，我们**把函数的参数改为可变参数**：

```
def calc(*numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

定义可变参数和定义一个`list`或`tuple`参数相比，仅仅在参数前面加了一个`*`号。在函数内部，**参数numbers接收到的是一个tuple**，因此，函数代码完全不变。但是，调用该函数时，可以传入任意个参数，包括0个参数：

```
>>> calc(1, 2)
5
>>> calc()
0
```

如果已经有一个list或者tuple，要调用一个可变参数怎么办？可以这样做：

```
>>> nums = [1, 2, 3]
>>> calc(nums[0], nums[1], nums[2])
14
```

这种写法当然是可行的，问题是太繁琐，所以Python允许你在list或tuple前面加一个`*`号，把list或tuple的元素变成可变参数传进去：

```
>>> nums = [1, 2, 3]
>>> calc(*nums)
14
```

`*nums`表示把`nums`这个list的所有元素作为可变参数传进去。这种写法相当有用，而且很常见。

3.4 命名关键字参数

对于关键字参数，函数的调用者可以传入任意不受限制的关键字参数。至于到底传入了哪些，就需要在函数内部通过`kw`检查。

仍以`person()`函数为例，我们希望检查是否有`city`和`job`参数：

```
def person(name, age, **kw):
    if 'city' in kw:
        # 有city参数
        pass
    if 'job' in kw:
        # 有job参数
        pass
    print('name:', name, 'age:', age, 'other:', kw)
```

但是调用者仍可以传入不受限制的关键字参数：

```
>>> person('Jack', 24, city='Beijing', addr='Chaoyang', zipcode=123456)
```

如果要限制关键字参数的名字，就可以用命名关键字参数，例如，只接收`city`和`job`作为关键字参数。这种方式定义的函数如下：

```
def person(name, age, *, city, job):
    print(name, age, city, job)
```

和关键字参数`**kw`不同，命名关键字参数需要一个特殊分隔符`*`，`*`后面的参数被视为命名关键字参数。

调用方式如下：

```
>>> person('Jack', 24, city='Beijing', job='Engineer')
Jack 24 Beijing Engineer
```

如果函数定义中已经有了一个可变参数，后面跟着的命名关键字参数就不再需要一个特殊分隔符`*`了：

```
def person(name, age, *args, city, job):
    print(name, age, args, city, job)
```

命名关键字参数必须传入参数名，这和位置参数不同。如果没有传入参数名，调用将报错：

```
>>> person('Jack', 24, 'Beijing', 'Engineer')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: person() takes 2 positional arguments but 4 were given
```

由于调用时缺少参数名`city`和`job`，Python解释器把这4个参数均视为位置参数，但`person()`函数仅接受2个位置参数。

命名关键字参数可以有缺省值，从而简化调用：

```
def person(name, age, *, city='Beijing', job):
    print(name, age, city, job)
```

由于命名关键字参数`city`具有默认值，调用时，可不传入`city`参数：

```
>>> person('Jack', 24, job='Engineer')
Jack 24 Beijing Engineer
```

使用命名关键字参数时，要特别注意，如果没有可变参数，就必须加一个`*`作为特殊分隔符。如果缺少`*`，Python解释器将无法识别位置参数和命名关键字参数：

```
def person(name, age, city, job):
    # 缺少 *，city和job被视为位置参数
    pass
```

3.5 参数组合

在Python中定义函数，可以用**必选参数、默认参数、可变参数、关键字参数和命名关键字参数**，这5种参数都可以组合使用。但是请注意，参数定义的顺序必须是：**必选参数、默认参数、可变参数、命名关键字参数**和**关键字参数**。

比如定义一个函数，包含上述若干种参数：

```
def f1(a, b, c=0, *args, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw)

def f2(a, b, c=0, *, d, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'd =', d, 'kw =', kw)
```

在函数调用的时候，Python解释器自动按照参数位置和参数名把对应的参数传进去。

```
>>> f1(1, 2)
a = 1 b = 2 c = 0 args = () kw = {}
>>> f1(1, 2, c=3)
a = 1 b = 2 c = 3 args = () kw = {}
>>> f1(1, 2, 3, 'a', 'b')
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {}
>>> f1(1, 2, 3, 'a', 'b', x=99)
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}
>>> f2(1, 2, d=99, ext=None)
a = 1 b = 2 c = 0 d = 99 kw = {'ext': None}
```

最神奇的是通过一个`tuple`和`dict`，你也可以调用上述函数：

```
>>> args = (1, 2, 3, 4)
>>> kw = {'d': 99, 'x': '#'}
>>> f1(*args, **kw)
a = 1 b = 2 c = 3 args = (4,) kw = {'d': 99, 'x': '#'}
>>> args = (1, 2, 3)
>>> kw = {'d': 88, 'x': '#'}
>>> f2(*args, **kw)
a = 1 b = 2 c = 3 d = 88 kw = {'x': '#'}
```

所以，对于任意函数，都可以通过类似`func(*args, **kw)`的形式调用它，无论它的参数是如何定义的。

> 虽然可以组合多达5种参数，但不要同时使用太多的组合，否则函数接口的可理解性很差。

4、函数返回值

4.1 “返回值”介绍

所谓“返回值”，就是程序中函数完成一件事情后，最后给调用者的结果

4.2 带有返回值的函数

想要在函数中把结果返回给调用者，需要在函数中使用`return`

如下示例:

```
def cal(a, b):
    c = a+b
    return c
```

或者

```
def cal(a, b):
    return a+b
```

4.3 保存函数的返回值

保存函数的返回值示例如下:

```
#定义函数
def cal(a, b):
    return a+b

#调用函数，顺便保存函数的返回值
result = cal(100,98)

#result已经保存了cal的返回值，所以接下来就可以使用了
print(result)
结果:

198
```

4.4 多个返回值

(1) 多个return?

```
def cal_nums():
    print("---1---")
    return 1
    # 函数中下面的代码不会被执行，因为return除了能够将数据返回之外，还有一个隐藏的功能：结束函数
    print("---2---")
    return 2
    print("---3---")
```

总结1： 一个函数中可以有多个`return`语句，但是只要有一个`return`语句被执行到，那么这个函数就会结束了，因此后面的`return`没有什么用处 如果程序设计为如下，是可以的因为不同的场景下执行不同的`return`

```
def cal_nums(num):
    print("---1---")
    if num == 100:
        print("---2---")
        return num+1  # 函数中下面的代码不会被执行，因为return除了能够将数据返回之外，还有一个隐藏的功能：结束函数
    else:
        print("---3---")
        return num+2
    print("---4---")

result1 = cal_nums(100)
print(result1)  # 打印101
result2 = cal_nums(200)
print(result2)  # 打印202
```

(2) 一个函数返回多个数据的方式

```
def calculate(a, b):
    shang = a//b
    yushu = a%b
    return shang, yushu  #默认是元组

result = calculate(5, 2)
print(result)  # 输出(2, 1)
```

总结2： `return`后面可以是元组，列表、字典等，只要是能够存储多个数据的类型，就可以一次性返回多个数据

```
def my_function():
    # return [1, 2, 3]
    # return (1, 2, 3)
    return {"num1": 1, "num2": 2, "num3": 3}
```

如果`return`后面有多个数据，那么默认是元组

```
In [1]: a = 1, 2
In [2]: a
Out[2]: (1, 2)

In [3]:
In [3]: b = (1, 2)
In [4]: b
Out[4]: (1, 2)

In [5]:
```

5、递归函数

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是**递归函数。**

举个例子，我们来计算阶乘`n! = 1 x 2 x 3 x ... x n`，用函数`fact(n)`表示，可以看出：

```
fact(n) = n! = 1 x 2 x 3 x ... x (n-1) x n = (n-1)! x n = fact(n-1) x n
```

所以，`fact(n)`可以表示为`n x fact(n-1)`，只有n=1时需要特殊处理。

于是，`fact(n)`用递归的方式写出来就是：

```
def fact(n):
    if n==1:
        return 1
    return n * fact(n - 1)
```

上面就是一个递归函数。可以试试：

```
>>> fact(1)
1
>>> fact(5)
120
>>> fact(100)
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000
```

如果我们计算`fact(5)`，可以根据函数定义看到**计算过程**如下：

```
===> fact(5)
===> 5 * fact(4)
===> 5 * (4 * fact(3))
===> 5 * (4 * (3 * fact(2)))
===> 5 * (4 * (3 * (2 * fact(1))))
===> 5 * (4 * (3 * (2 * 1)))
===> 5 * (4 * (3 * 2))
===> 5 * (4 * 6)
===> 5 * 24
===> 120
```

递归函数的**优点**是定义简单，逻辑清晰。理论上，所有的递归函数都可以写成循环的方式，但循环的逻辑不如递归清晰。

使用递归函数需要**注意**防止栈溢出。在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。可以试试`fact(1000)`：

```
>>> fact(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in fact
  ...
  File "<stdin>", line 4, in fact
RuntimeError: maximum recursion depth exceeded in comparison
```

解决递归调用栈溢出的方法是通过**尾递归**优化，事实上尾递归和循环的效果是一样的，所以，把循环看成是一种特殊的尾递归函数也是可以的。

尾递归是指，在函数返回的时候，调用自身本身，并且，`return`语句不能包含表达式。这样，编译器或者解释器就可以把尾递归做优化，使递归本身无论调用多少次，都只占用一个栈帧，不会出现栈溢出的情况。

上面的`fact(n)`函数由于`return n * fact(n - 1)`引入了乘法表达式，所以就不是尾递归了。要改成尾递归方式，需要多一点代码，主要是要把每一步的乘积传入到递归函数中：

```
def fact(n):
    return fact_iter(n, 1)

def fact_iter(num, product):
    if num == 1:
        return product
    return fact_iter(num - 1, num * product)
```

可以看到，`return fact_iter(num - 1, num * product)`仅返回递归函数本身，`num - 1`和`num * product`在函数调用前就会被计算，不影响函数调用。

`fact(5)`对应的`fact_iter(5, 1)`的调用如下：

```
===> fact_iter(5, 1)
===> fact_iter(4, 5)
===> fact_iter(3, 20)
===> fact_iter(2, 60)
===> fact_iter(1, 120)
===> 120
```

尾递归调用时，如果做了优化，栈不会增长，因此，无论多少次调用也不会导致栈溢出。

遗憾的是，大多数编程语言没有针对尾递归做优化，Python解释器也没有做优化，所以，即使把上面的`fact(n)`函数改成尾递归方式，也会导致栈溢出。

小结

- 使用递归函数的优点是逻辑简单清晰，缺点是过深的调用会导致栈溢出。
- 针对尾递归优化的语言可以通过尾递归防止栈溢出。尾递归事实上和循环是等价的，没有循环语句的编程语言只能通过尾递归实现循环。
- Python标准的解释器没有针对尾递归做优化，任何递归函数都存在栈溢出的问题。

6、局部变量与全局变量

6.1 局部变量

- **局部变量**，就是在函数内部定义的变量。
- **作用范围**是这个函数内部，即只能在这个函数中使用，在函数的外部是不能使用的。因为其作用范围只是在自己的函数内部，所以不同的函数可以定义相同名字的局部变量（打个比方，把你、我是当做成函数，把局部变量理解为每个人手里的手机，你可有个iPhone11，我当然也可以有个iPhone11了， 互不相关）
- **局部变量的作用**，为了临时保存数据需要在函数中定义变量来进行存储。
- 当函数调用时，局部变量被创建，当函数调用完成后这个变量就不能够使用了。

```
def show():
    # 定义局部变量
    sal = 15000
    print("薪资:", sal)
# show()
# print(sal)
```

6.2 全局变量

如果一个变量，既能在一个函数中使用，也能在其他的函数中使用，这样的变量就是**全局变量**。

例如：有2个兄弟 各自都有手机，各自有自己的小秘密在手机里，不让另外一方使用（可以理解为**局部变量**）；但是家里的电话是2个兄弟都可以随便使用的（可以理解为**全局变量**）

```
# 定义全局变量
money = 1200

def test1():
    print(money)  # 虽然没有定义变量money但是可是使用全局变量money
```

```
def test2():
    print(money)  # 虽然没有定义变量money但是可是使用全局变量money

# 调用函数
test1()
test2()

运行结果:
1200
1200
```

总结:

- 在函数外边定义的变量叫做全局变量
- 全局变量能够在所有的函数中进行访问

6.3 全局变量和局部变量冲突问题

代码:

```
# 定义全局变量
x = 100

def test1():
    # 定义局部变量,与全局变量名字相同
    x = 300
    print('---test1---%d'%x)
    #修改
    x = 200
    print('修改后的%d'%x)

def test2():
    print('x = %d'%x)

test1()
test2()

结果:
---test1---300
修改后的200
x = 100
```

总结:

- 当函数内出现局部变量和全局变量相同名字时，函数内部中的 `变量名 = 数据` 此时理解为定义了一个局部变量，而不是修改全局变量的值

6.4 修改全局变量

函数中进行使用**全局变量**时可否进行修改呢?

```
# 定义全局变量
x = 100

def test1():

    # 定义全局变量,使用global函数声明变量x为全局变量
```

```
    global x
    print('修改之前：%d'%x)

    #修改
    x = 200
    print('修改后的%d'%x)

def test2():
    print('x = %d'%x)

test1()
test2()
```

结果：
修改之前：100
修改后的200
x = 200

7、捕获异常

一旦出错，还要一级一级上报，直到某个函数可以处理该错误（比如，给用户输出一个错误信息）。

所以高级语言通常都内置了一套`try...except...finally...`的**错误处理机制**，Python也不例外。

让我们用一个例子来看看`try`的机制：

```
try:
    print('try...')
    r = 10 / 0
    print('result:', r)
except ZeroDivisionError as e:
    print('except:', e)
finally:
    print('finally...')
print('END')
```

当我们认为某些代码可能会出错时，就可以用`try`来运行这段代码，如果执行出错，则**后续代码不会继续执行**，而是直接跳转至错误处理代码，即`except`语句块，执行完`except`后，如果有`finally`语句块，则执行`finally`语句块，至此，执行完毕。

上面的代码在计算`10 / 0`时会产生一个除法运算错误：

```
try...
except: division by zero
finally...
END
```

从输出可以看到，当错误发生时，后续语句`print('result:', r)`不会被执行，`except`由于捕获到`ZeroDivisionError`，因此被执行。最后，`finally`语句被执行。然后，程序继续按照流程往下走。

如果把除数`0`改成`2`，则执行结果如下：

```
try...
result: 5
finally...
END
```

由于没有错误发生，所以`except`语句块不会被执行，但是`finally`如果有，则一定会被执行（可以没有`finally`语句）。

你还可以猜测，错误应该有很多种类，如果发生了不同类型的错误，应该由不同的`except`语句块处理。没错，可以有多个`except`来捕获不同类型的错误：

```
try:
    print('try...')
    r = 10 / int('a')
    print('result:', r)
except ValueError as e:
    print('ValueError:', e)
except ZeroDivisionError as e:
    print('ZeroDivisionError:', e)
finally:
    print('finally...')
print('END')
```

`int()`函数可能会抛出`ValueError`，所以我们用一个`except`捕获`ValueError`，用另一个`except`捕获`ZeroDivisionError`。

此外，如果没有错误发生，可以在`except`语句块后面加一个`else`，当没有错误发生时，会自动执行`else`语句：

```
try:
    print('try...')
    r = 10 / int('2')
    print('result:', r)
except ValueError as e:
    print('ValueError:', e)
except ZeroDivisionError as e:
    print('ZeroDivisionError:', e)
else:
    print('no error!')
finally:
    print('finally...')
print('END')
```

Python的错误其实也是`class`，所有的错误类型都**继承**自`BaseException`，所以在使用`except`时需要注意的是，它不但捕获该类型的错误，还把其子类也“一网打尽”。比如：

```
try:
    foo()
except Exception as e:
    print('Exception')
except TypeError as e:
    print('TypeError')
finally:
    print('finally...')
```

第二个`except`永远也捕获不到TypeError，因为TypeError是`Exception`的子类，如果有，也被第一个`except`给捕获了。

Python所有的错误都是从`BaseException`类派生的，常见的错误类型和继承关系看这里：

https://docs.python.org/3/library/exceptions.html#exception-hierarchy

使用`try...except`捕获错误还有一个巨大的好处，就是可以跨越多层调用，比如函数`main()`调用`foo()`，`foo()`调用`bar()`，结果`bar()`出错了，这时，只要`main()`捕获到了，就可以处理：

```
def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    try:
        bar('0')
    except Exception as e:
        print('Error:', e)
    finally:
        print('finally...')
main()
```

也就是说，不需要在每个可能出错的地方去捕获错误，只要在合适的层次去捕获错误就可以了。这样一来，就大大减少了写`try...except...finally`的麻烦。

python所有的标准异常类：

异常名称	描述
BaseException	所有异常的基类
SystemExit	解释器请求退出
KeyboardInterrupt	用户中断执行(通常是输入^C)
Exception	常规错误的基类
StopIteration	迭代器没有更多的值
GeneratorExit	生成器(generator)发生异常来通知退出
SystemExit	Python 解释器请求退出
StandardError	所有的内建标准异常的基类
ArithmeticError	所有数值计算错误的基类
FloatingPointError	浮点计算错误
OverflowError	数值运算超出最大限制
ZeroDivisionError	除(或取模)零 (所有数据类型)
AssertionError	断言语句失败
AttributeError	对象没有这个属性
EOFError	没有内建输入,到达EOF 标记
EnvironmentError	操作系统错误的基类
IOError	输入/输出操作失败
OSError	操作系统错误
WindowsError	系统调用失败
ImportError	导入模块/对象失败
KeyboardInterrupt	用户中断执行(通常是输入^C)
LookupError	无效数据查询的基类
IndexError	序列中没有没有此索引(index)
KeyError	映射中没有这个键
MemoryError	内存溢出错误(对于Python 解释器不是致命的)
NameError	未声明/初始化对象 (没有属性)
UnboundLocalError	访问未初始化的本地变量
ReferenceError	弱引用(Weak reference)试图访问已经垃圾回收了的对象
RuntimeError	一般的运行时错误
NotImplementedError	尚未实现的方法

异常名称	描述
SyntaxError	Python 语法错误
IndentationError	缩进错误
TabError	Tab 和空格混用
SystemError	一般的解释器系统错误
TypeError	对类型无效的操作
ValueError	传入无效的参数
UnicodeError	Unicode 相关的错误
UnicodeDecodeError	Unicode 解码时的错误
UnicodeEncodeError	Unicode 编码时错误
UnicodeTranslateError	Unicode 转换时错误
Warning	警告的基类
DeprecationWarning	关于被弃用的特征的警告
FutureWarning	关于构造将来语义会有改变的警告
OverflowWarning	旧的关于自动提升为长整型(long)的警告
PendingDeprecationWarning	关于特性将会被废弃的警告
RuntimeWarning	可疑的运行时行为(runtime behavior)的警告
SyntaxWarning	可疑的语法的警告
UserWarning	用户代码生成的警告

五、总结

1. 本节课的所有知识点全是重点，后面的学习离不开基础语法；
2. 需要着重掌握的是函数的定义以及参数的传递；
3. 局部变量和全局变量的定义和使用
4. 所有的代码都要多敲几遍，练习是学习编程最简单的途径！！！