第 3 章
函数与模块

学习目标

- 掌握调用函数及函数的参数。
- 掌握函数嵌套的定义，理解 lambda 函数。
- 理解递归函数、函数列表、作用域分类。
- 掌握 global 语句、nonlocal 语句。
- 掌握函数的调用关系及参数传递的规则。
- 了解函数的默认参数及函数参数默认值的规则。
- 掌握异常的传递机制、模块的导入及执行。
- 掌握搜索路径、对象属性、隐藏的模块变量。

函数是实现某一特定功能的语句集合。函数可以重复使用，提高了代码的可重用性；函数通常实现较为单一的功能，提高了程序的独立性；同一个函数，可以通过接收不同的参数实现不同的功能，提高了程序的适应性。Python 提供了很多内置函数，用户也可以使用自己定义的函数。

Python 作为高级编程语言，适合开发各类应用程序。编写 Python 程序可以使用内置的标准库、第三方库，也可以使用用户自己开发的函数库，从而方便代码复用。Python 的编程思想注重运用各种函数库完成应用系统的开发。可以使用库、模块、包、类、函数等多个概念从不同角度来构建 Python 程序。为方便描述，本书不严格区分库和模块的概念。

程序在运行过程中发生错误是不可避免的，这种错误就是异常(Exception)。用户在开发一个完整的应用系统时，在程序中应提供异常处理策略。

Python 中包含丰富的异常处理措施。Python 的异常处理机制使得程序运行时出现的问题可以用统一的方式进行处理，增加了程序的稳定性和可读性，规范了程序的设计风格，提高了程序的质量。

本章主要介绍函数的定义、调用及参数传递，以及一些内置函数的应用、模块的概念、Python 标准库中的模块、下载和使用第三方库、构建用户自定义的模块等内容。Python 的异常处理技术包括用户自定义的异常。

3.1 函数

3.1.1 定义函数

def 语句用于定义函数，其基本格式为

```
def 函数名(参数表):
    函数语句
return 返回值
```

参数和返回值都可以省略,示例代码如下:

```
>>>def hello():                          #定义函数
...    print('Python 你好')
...
>>>hello()                               #调用函数
Python 你好
```

hello()函数没有参数和返回值,它调用 print()函数输出一个字符串。

为函数指定参数时,参数之间用逗号分隔。

下面的代码为函数定义了两个参数,并返回两个参数的和。

```
>>>def add(a,b):                         #定义函数
...    return a+b
...
>>>add(1,2)                              #调用函数
3
```

3.1.2　调用函数

调用函数的基本格式为

```
函数名(参数表)
```

在 Python 中,所有的语句都是解释执行的,不存在 C/C++ 语言中的编译过程。

def 也是一条可执行语句,它完成了函数的定义。Python 中,函数的调用必须出现在函数的定义之后。在 Python 中,函数也是对象(function 对象)。def 语句在执行时会创建一个函数对象。函数名是一个变量,它引用 def 语句创建的函数对象。可将函数名赋值给变量,使变量引用同一个函数。示例代码如下:

```
>>>def add(a,b):                         #定义函数
...    return a+b
>>>add                                   #直接用函数名,可返回函数对象的内存地址
<function add at 0x00D41078>
>>>add(10,20)                           #调用函数
30
>>>x=add                                 #将函数名赋值给变量
>>>x(1,2)                               #通过变量调用函数
3
```

3.1.3　函数的参数

函数定义的参数表中的参数称为形式参数,简称形参。调用函数时,参数表中提供的参

数称为实际参数，简称实参。实参可以是常量、表达式或变量。当实参是常量或表达式时，直接将常量或表达式的计算结果传递给形参。

在 Python 中，变量保存的是对象的引用，当实参为变量时，参数传递会将实参对对象的引用赋值给形参。

1. 参数的多态性

多态是面向对象的一个特点，指不同对象执行同一个行为可能会得到不同的结果。当同一个函数传递的实际参数类型不同时，可获得不同的结果，这体现了多态性。例如：

```
>>>def add(a,b):
...     return a+b                          #两个参数执行加法运算
...
>>>add(1,2)                                 #执行数字加法
3
>>>add('abc','def')                         #执行字符串连接
'abcdef'
>>>add((1,2),(3,4))                         #执行元组合并
(1,2,3,4)
>>>add([1,2],[3,4])                         #执行列表合并
[1,2,3,4]
```

2. 参数赋值传递

调用函数时，会按参数的先后顺序依次将实参传递给形参。例如，调用 add(1,2)时，1 传递给 a，2 传递给 b。

Python 允许以形参赋值的方式将实参传递给指定形参。例如：

```
>>>def add(a,b):
...     return a+b
...
>>>add(a='ab',b='cd')                       #通过赋值来传递参数
'abcd'
>>>add(b='ab',a='cd')                       #通过赋值来传递参数
'cdab'
```

采用参数赋值传递时，因为指明了形参名称，所以参数的先后顺序已无关紧要。

参数赋值传递的方式称为关键字传递。

3. 参数传递与共享引用

示例代码如下：

```
>>>def f(x):
...     x=100
...
>>>a=10
>>>f(a)
>>>a
10
```

从结果可以看出，将实参 a 传递给形参 x 后，在函数中重新赋值 x 并不会影响实参 a，这是

因为 Python 中的赋值是建立变量到对象的引用，重新赋值时，意味着形参引用了新的对象。

4. 传递可变对象的引用

当实参引用的是可变对象（如列表、字典等）时，若在函数中修改形参，通过共享引用，实参也可以获得修改后的对象。

示例代码如下：

```
>>>def f(a):
...     a[0]='abc'                          #修改列表第一个值
...
>>>x=[1,2]
>>>f(x)                                     #调用函数，传递列表对象的引用
>>>x                                        #变量 x 引用的列表对象在函数中被修改
['abc',2]
```

如果不希望函数中的修改影响函数外的数据，应注意避免传递可变对象的引用。如果要避免列表在函数中被修改，可使用列表的拷贝作为实参。

示例代码如下：

```
>>>def f(a):
...     a[0]='abc'                          #修改列表第一个值
...
>>>x=[1,2]
>>>f(x[:])                                  #传递列表的拷贝
>>>x                                        #结果显示原列表不变
[1,2]
```

还可以在函数内对列表进行拷贝，调用函数时实参仍使用变量，示例代码如下：

```
>>>def f(a):
...     a=a[:]                              #拷贝列表
...     a[0]='abc'                          #修改列表的拷贝
...
>>>x=[1,2]
>>>f(x)                                     #调用函数
>>>x                                        #结果显示原列表不变
[1,2]
```

5. 有默认值的可选参数

在定义函数时，可以为参数设置默认值。

调用函数时，如果未提供实参，则形参取默认值，示例代码如下：

```
>>>def add(a,b=-100):                       #参数 b 默认值为-100
...     return a+b
...
>>>add(1,2)                                 #传递指定参数
3
>>>add(1)                                   #形参 b 取默认值
-99
```

需要注意的是，带默认值的参数为可选参数，在定义函数时，应放在参数表的末尾。

6. 接收任意个数的参数

在定义函数时，如果在参数名前面使用星号“*”，则表示形参是一个元组，可接收任意个数的参数。调用函数时，可以不为带星号的形参提供数据。

示例代码如下：

```
>>>def add(a, *b):
...     s=a
...     for x in b:                          #用循环迭代元组 b 中的对象
...         s+=x                             #累加
...     return s                             #返回累加结果
...
>>>add(1)                                   #不为带星号的形参提供数据，此时形参 b 为空元组
1
>>>add(1,2)                                 #求两个数的和，此时形参 b 为元组(2,)
3
>>>add(1,2,3)                               #求 3 个数的和，此时形参 b 为元组(2,3)
6
>>>add(1,2,3,4,5)                           #求 5 个数的和，此时形参 b 为元组(2,3,4,5)
15
```

7. 必须通过赋值传递的参数

Python 允许使用必须通过赋值传递的参数。在定义函数时，带星号参数之后的参数必须通过赋值传递。

示例代码如下：

```
>>>def add(a, *b,c):
...     s=a+c
...     for x in b:
...         s+=x
...     return s
...
>>>add(1,2,3)                               #形参 c 未使用赋值传递，出错
Traceback(most recent call last):
  File "<stdin>",line 1,in <module>
TypeError:add() missing 1 required keyword-only argument:'c'
>>>add(1,2,c=3)                            #形参 c 使用赋值传递
6
>>>add(1,c=3)                               #带星号参数可以省略
4
```

在定义函数时，也可单独使用星号，但其后的参数必须通过赋值传递。

示例代码如下：

```
>>>def f(a, *,b,c):                         #参数 b 和 c 必须通过赋值传递
...     return a+b+c
...
>>>f(1,b=2,c=3)
6
```

3.1.4　函数嵌套定义

Python 允许在函数内部定义函数，示例代码如下：

```
>>>def add(a,b):
...     def getsum(x):              #在函数内部定义的函数，将字符串转换为 Unicode 码求和
...         s=0
...         for n in x:
...             s+=ord(n)
...         return s
...     return getsum(a)+getsum(b)          #调用内部定义的函数 getsum()
...
>>>add('12','34')                          #调用函数
202
```

注意：内部函数只能在函数内部使用。

3.1.5　lambda 函数

lambda 函数也称表达式函数，用于定义匿名函数。可将 lambda 函数赋值给变量，通过变量调用函数。

lambda 函数定义的基本格式为

```
lambda 参数表:表达式
```

示例代码如下：

```
>>>add=lambda a,b:a+b                      #定义表达式函数，赋值给变量
>>>add(1,2)                                #函数调用格式不变
3
>>>add('ab','ad')
'abad'
```

lambda 函数非常适合定义简单的函数。

与 def 不同，lambda 的函数体只能是一个表达式。

可在表达式中调用其他函数，但不能使用其他语句。

示例代码如下：

```
>>>add=lambda a,b:ord(a)+ord(b)          #在 lambda 表达式中调用其他函数
>>>add('1','2')
99
```

3.1.6　递归函数

递归函数是指在函数体内调用函数本身。

例如，下面的函数 fac()实现了计算阶乘。

```
>>>def fac(n):                              #定义函数
...    if n==0:                            #递归调用的终止条件
...        return 1
...    else:
...        return n*fac(n-1)               #递归调用函数本身
...
>>>fac(5)
120
```

注意：递归函数必须在函数体中设置递归调用的终止条件。如果没有设置递归调用终止条件，程序会在超过 Python 允许的最大递归调用深度后产生 RecursionError 异常（递归调用错误）。

3.1.7 函数列表

因为函数是一种对象，所以可将其作为列表元素使用，然后通过列表索引来调用函数。示例代码如下：

```
>>>d=[lambda a,b:a+b,lambda a,b:a*b]    #使用 lambda 函数建立列表
>>>d[0](1,3)                            #调用第一个函数
4
>>>d[1](1,3)                            #调用第二个函数
3
```

也可使用 def 定义的函数来创建列表，示例代码如下：

```
>>>def add(a,b):                        #定义求和函数
...    return a+b
...
>>>def fac(n):                          #定义求阶乘函数
...    if n==0:
...        return 1
...    else:
...        return n*fac(n-1)
>>>d=[add,fac]                          #建立函数列表
>>>d[0](1,2)                            #调用求和函数
3
>>>d[1](5)                              #调用求阶乘函数
120
>>>d=(add,fac)                          #建立包含函数列表的元组对象
>>>d[0](2,3)                            #调用求和函数
5
>>>d[1](5)                              #调用求阶乘函数
120
```

Python 还允许使用字典来建立函数映射，示例代码如下：

```
>>>d={'求和':add,'求阶乘':fac}          #用函数 add 和 fac 建立函数映射
>>>d['求和'](1,2)                       #调用求和函数
3
>>>d['求阶乘'](5)                       #调用求阶乘函数
120
```

3.2　变量范围

3.2.1　作用域分类

变量的范围就是作用域，是变量的可使用范围，也称为变量的命名空间。在第一次给变量赋值时，变量的创建位置决定了变量的作用域。Python 中变量的作用域可分为 4 类：内置作用域、文件作用域、函数嵌套作用域和本地作用域，如图 3-1 所示。

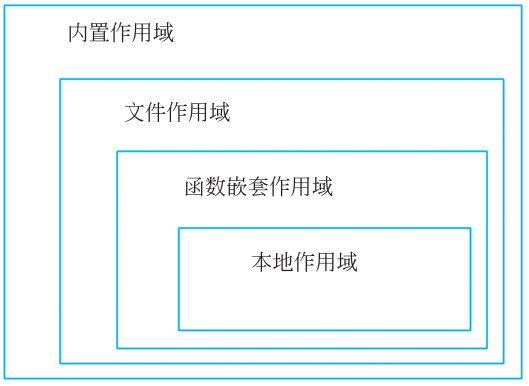

图 3-1　变量的作用域

① 本地作用域：当没有内部函数时，函数体为本地作用域；函数内通过赋值创建的变量、函数参数都属于本地作用域。

② 函数嵌套作用域：当包含内部函数时，函数体为函数嵌套作用域。

③ 文件作用域：程序文件(也称模块文件)的内部为文件作用域。

④ 内置作用域：Python 运行时的环境为内置作用域，它包含 Python 的各种预定义变量和函数。

内置作用域和文件作用域称为全局作用域。根据作用域范围的大小，作用域外部的变量和函数可以在作用域内使用。相反，作用域内的变量和函数不能在作用域外使用。

通常将变量名分为两种：全局变量和本地变量。在内置作用域和文件作用域中定义的变量和函数都属于全局变量。在函数嵌套作用域和本地作用域内定义的变量和函数都属于本地变量，本地变量也称为局部变量。

考察下面的代码：

```
#文件作用域
a=10                                #a 是全局变量
def add(b):                         #参数 b 是函数 add 内的本地变量
    c=a+b                           #c 是函数 add 内的本地变量，a 是函数外部的全局变量
    return c
print(add(5))                       #调用函数
```

该程序在运行过程中会创建 4 个变量：a、b、c 和 add。a 和 add 是文件作用域内的全局变量，b 和 c 是函数 add 内部的本地变量。

另外，该程序还用到了 print()这个内置函数，它是内置作用域中的全局变量。

当作用域外的变量与作用域内的变量名称相同时，遵循“本地”优先原则，此时外部的变量将被屏蔽，称为作用域隔离原则。例如：

```
a=10                                    #赋值,创建全局变量 a
def show():
    a=100                               #赋值,创建本地变量 a
    print('inshow():a=',a)              #输出本地变量 a
show()
inshow():a=100
a                                       #输出全局变量 a
10
```

3.2.2 global 语句

在函数内部给变量赋值时，默认情况下该变量为本地变量。为了在函数内部给全局变量赋值，Python 提供了 global 语句，用于在函数内部声明全局变量。

示例代码如下：

```
def show():
    global a                            #声明 a 是全局变量
    print('a=',a)                       #输出全局变量 a
    a=100                               #给全局变量 a 赋值
    print('a=',a)
a=10
show()
a=10
a=100
a
100
```

3.2.3 nonlocal 语句

作用域隔离原则同样适用于嵌套函数。在嵌套函数内使用与外层函数同名的变量时，若该变量在嵌套函数内没有被赋值，则该变量就是外层函数的本地变量。

示例代码如下：

```
>>>def test():
...     a=10                           #创建 test 函数的本地变量 a
...     def show():
...         print('inshow(),a=',a)     #使用 test 函数的本地变量 a
...     show()
...     print('intest(),a=',a)         #使用 test 函数的本地变量 a
...
>>>test()
inshow(),a=10
intest(),a=10
```

修改上面的代码，在嵌套函数 show()内为 a 赋值，代码如下：

```
>>>def test():
...     a=10                                  #创建 test 函数的本地变量 a
...     def show():
...         a=100                             #创建 show 函数的本地变量 a
...         print('inshow(),a=',a)            #使用 show 函数的本地变量 a
...     show()
...     print('intest(),a=',a)                #使用 test 函数的本地变量 a
...
>>>test()
inshow(),a=100
intest(),a=10
```

如果要在嵌套函数内部为外层函数的本地变量赋值，可使用 Python 提供的 nonlocal 语句。

nonlocal 语句与 global 语句类似，它声明的变量是外层函数的本地变量，示例代码如下：

```
>>>def test():
...     a=10                                  #创建 test 函数的本地变量 a
...     def show():
...         nonlocal a                        #声明 a 是 test 函数的本地变量
...         a=100                             #为 test 函数的本地变量 a 赋值
...         print('inshow(),a=',a)            #使用 test 函数的本地变量 a
...     show()
...     print('intest(),a=',a)                #使用 test 函数的本地变量 a
...
>>>test()
inshow(),a=100
intest(),a=100
```

3.3　函数调用简介

3.3.1　函数调用

程序的执行总是从主程序函数开始，完成对其他函数的调用后再返回到主程序函数，最后由主程序函数结束整个程序。嵌套调用就是一个函数调用另一个函数，被调用的函数又进一步调用另一个函数，形成一层层的嵌套关系，一个复杂的程序存在多层的函数调用。

图 3-2(左)展示了这种关系，主程序函数调用函数 A，在 A 中又调用函数 B，B 又调用 C，在 C 完成后返回 B 的调用处，继续 B 的执行，之后 B 执行完毕，返回 A 的调用处，A 又接着往下执行，随后 A 又调用函数 D，D 执行完后返回 A，A 执行完后返回主程序函数，主程序接着往下执行，主程序完成后程序就结束了。

函数调用可以这样一层层地嵌套下去，但函数调用一般不可以出现循环，图 3-2(右)所示是一个循环，函数 X 调用函数 Y，Y 又反过来调用 X，之后 X 又调用 Y，形成了死循环。

问题描述：

输入整数 n，计算 1＋(1＋2)＋(1＋2＋3)＋…＋(1＋2＋3＋…＋n)的和。

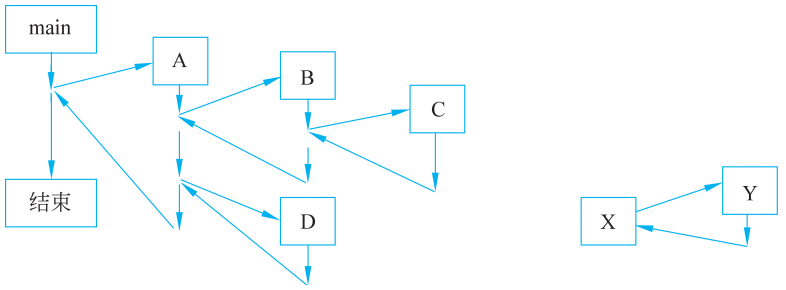

图 3-2　函数的调用

算法分析：

显然第 m 项是(1+2+…+m)，设计一个函数计算(1+2+…+m)的和，函数为 sum(m)，之后再累计 sum(1)+sum(2)+…+sum(n)就可以了。

示例代码如下：

```
#实例 3-3-1:ex3-3-1.py
#输入整数 n,计算 1+(1+2)+(1+2+3)+…+(1+2+3+…+n)的和
def sum(m):
    s=0
    for n in range(1,m+1):
        s=s+n
    return s
def sumAll(n):
    s=0
    for m in range(1,n+1):
        s=s+sum(m)
    return s
n=input("n=")
n=int(n)
print("总和是",sumAll(n))
```

运行结果：

```
n=3
总和是 10
```

问题描述：

输入一个正整数，找出它的所有素数因数。

算法分析：

例如，12 的因素有 1、2、3、4、6、12，但其中只有 1、2、3 是素数，因此 12 的素数因数为 1、2、3。

示例代码如下：

```
#实例 3-3-2:
#输入一个正整数,找出它的所有素数因数
>>> def IsPrime(m):
...     for n in range(2, m):
```

```
...         if m % n == 0:
...             return 0
...     return 1
...
>>> n = input("n=")
>>> n = int(n)
>>> for p in range(1, n+1):
...     if n % p == 0 and IsPrime(p) == 1:
...         print(p)
```

运行结果：

```
n=12
1
2
3
```

3.3.2　案例：验证哥德巴赫猜想

1. 案例描述

著名的哥德巴赫猜想预言任何一个大于 2 的偶数都可以分解成为两个素数的和，例如：6＝3＋3，8＝3＋5，10＝5＋5，12＝5＋7 等。请编写一个程序，验证在 100 以内的偶数都可以这样分解。

2. 案例分析

一个偶数 n 可以分解成两个数 p 与 q 的和，即 n＝p＋q，显然只要找到 p 与 q 都是素数的分解就可以，为此可以设计一个判断素数的函数：

```
IsPrime(m);
```

它用来判断 m 这个整数是否素数，如果是就返回 1，否则返回 0。通过调用 IsPrime(p)及 IsPrime(q)就可以知道 p 与 q 是否同时为素数。

示例代码如下：

```
#实例 3-3-3:ex3-3-3.py
#案例:验证哥德巴赫猜想
>>> def IsPrime(m):
...     for n in range(2, m):
...         if m % n == 0:
...             return 0
...     return 1
...
>>> for n in range(2, 101, 2):
...     for p in range(3, n + 1, 2):
...         q = n - p
...         if IsPrime(p) and IsPrime(q):
...             print(n, "=", p, "+", q)
...             break
```

运行结果：

```
6=3+3
8=3+5
10=3+7
12=5+7
14=3+11
16=3+13
18=5+13
```

3.4 函数默认参数

3.4.1 默认参数的使用

函数默认参数是指在定义函数时为一些参数预先设定一个值，在调用时，如果不提供这个参数的实际值，就使用默认的参数值。例如：

```
#实例 3-4-1:ex3-4-1.py
def fun(a,b=1,c=2):
    print(a,b,c)
fun(0)
fun(1,2)
fun(1,2,3)
```

运行结果：

```
012
122
123
```

在 fun(0)调用中，a=0，而没有为 b、c 提供参数值，故使用默认的 b=1、c=2 的值；在 fun(1,2)调用中，a=1，b=2，而没有为 c 提供参数值，故使用默认的 c=2 的值；在 fun(1,2,3)调用中，a=1，b=2，c=3，函数调用时，实际参数值是按顺序给函数参数的，也可以指定参数名称而不按顺序进行调用。在 fun(a，b=1，c=2)调用中，我们把 a 称为位置参数(positional argument)，把 b、c 称为键值参数(keyword argument)。

3.4.2 默认参数的位置

Python 规定默认的键值参数必须出现在函数中没有默认值的位置参数的后面，例如下面的函数是正确的：

```
def fun(a,b=1,c=2):
print(a,b,c)
```

但是下列函数是错误的：

```
def fun(a=0,b,c=2):
print(a,b,c)
```

因为键值参数 a=0 出现在了位置参数 b 的前面。

除了在定义函数时要求键值参数出现在位置参数的后面以外，在调用时也要求键值参数在位置参数的后面，例如：

```
def fun(a,b=1,c=2):
print(a,b,c)
```

那么调用：

```
fun(a=0,1,c=2)
```

是错误的，因为 a=0 是键值参数，它出现在了位置参数 1 的前面，但是下列调用是正确的：

```
fun(0)
fun(0,1)
fun(0,c=3)
fun(a=0)
```

一般来说，实际的位置参数值也可以赋值给函数的位置参数和键值参数，例如：

```
fun(0,1)
```

实际的键值参数也可以赋值给函数的位置参数与键值参数，例如：

```
fun(a=0,c=3)
```

3.4.3　案例：print()函数的默认参数

1. 案例描述

print()函数是使用频繁的函数之一，了解它的参数结构是十分重要的。

2. 案例分析

在 Python 的“>>>”提示符下输入 help(print)并按 Enter 键，可以看到 print 函数的参数为

```
print(value,...,sep='',end='\n',file=sys.stdout,flush=False)
```

参数 sep=''表示 print 中各个输出项的分隔符号是空格。

end='\n'表示 print 的结束符号是换行，这就是 print 输出的内容独占一行的原因。

file=sys.stdout 表示内容默认输出到标准输出设备，即控制台。

flush=False 表示输出的内容不是即刻发送到输出端。

3. 案例分析

设计程序改变 sep、end 参数，可以看到 print 语句的不同输出结果。例如：

```
print(1,2)
print(1,2,sep='-')
print("line")
print('line',end='*')
print('end')
```

运行结果：

```
12
1-2
line
line*end
```

由此可见，print(1,2)输出 1 与 2 的默认分隔符号是空格，但是 print(1,2,sep='-')输出 1 与 2 的分隔符号是“-”。

print("line")输出的 line 独占一行，但是 print('line',end='*')输出 line*，而且不独占一行，print('end')的 end 接在后面。

3.5 函数与异常

3.5.1 异常处理

1. 函数的异常捕捉

在 Python 中，如果一个函数抛出了一个异常，那么在调用函数的地方就可以捕捉到这个异常。例如：

```
#实例 3-5-1:ex3-5-1.py
#函数的异常捕捉
def fun():
    print("start")
    n=1/0
    print("end")
try:
    fun()
except Exception as err:
    print(err)
```

运行结果：

```
start
division by zero
```

由此可见，fun()函数中出现的异常在主程序调用 fun 时可以捕捉到，如果 Python 程序中的一个地方出现异常，那么异常就会传递到上一级调用的地方，这个过程会一直传递下去，直到异常被捕捉到为止。如果整个过程没有遇到捕捉语句，程序就会因异常而结束。因此，如果在 fun()中已经捕捉了异常，那么调用的主程序位置就捕捉不到了。例如：

```
#实例 3-5-2:ex3-5-2.py
#函数的异常捕捉
def fun():
    print("start")
    try:
        n=1/0
        print("end")
    except:
        print("error")
try:
    fun()
except Exception as err:
    print(err)
```

运行结果：

```
start
error
```

2. 异常的传递

示例代码如下：

```
#实例 3-5-3:ex3-5-3.py
#异常的传递
def A():
    print("startA")
    n=1/0
    print("endA")
def B():
    print("startB")
    A()
    print("endB")
try:
    B()
    print("done")
except Exception as err:
    print("finish")
```

运行结果：

```
startB
startA
finish
```

由此可见，函数 A 中出现的异常它自己没有捕捉，在调用函数 B 中也没有捕捉，最后在主程序中被捕捉到，即异常有传递性。在一个函数中，没有被捕捉的异常会传递给调用这个函数的其他函数，这个过程会一直传递下去，直到异常被捕捉为止，就不再往后传递了。例如：

```
#实例 3-5-4:ex3-5-4.py
#异常的传递
def A():
    print("startA")
    n=1/0
    print("endA")
def B():
    print("startB")
    try:
        A()
    except Exception as err:
        print(err)
    print("endB")
try:
    B()
    print("done")
except Exception as err:
    print(err)
```

运行结果：

```
startB
startA
division by zero
endB
done
```

如果出现的异常一直没有被捕获，那么就传递到系统，程序就会终止。例如：

```
#实例 3-5-5:ex3-5-5.py
#异常终止程序
def A():
    print("startA")
    n=1/0
    print("endA")
def B():
    print("startB")
    A()
    print("endB")
B()
print("finish")
```

运行结果：

```
startB
startA
Traceback(most recent call last):
File "e:/广东开放大学/pythoncode/chapter3/ex3-5-5.py",line 11,in <module>
B()
File "e:/广东开放大学/pythoncode/chapter3/ex3-5-5.py",line 9,in B
A()
```

```
File "e:/广东开放大学/pythoncode/chapter3/ex3-5-5.py",line 5,in A
n=1/0
```

3.5.2　案例：时间的输入与显示

1. 案例描述

输入一个有效的时间，并显示该时间。

2. 案例分析

设置时间格式为 h：m：s，输入时保证 h、m、s 的值有效，否则会抛出异常。

示例代码如下：

```
#实例 3-5-6:ex3-5-6.py
#时间的输入与显示
def myTime():
    h=input("时:")
    h=int(h)
    if h<0 or h>23:
        raise Exception("无效的时")
    m=input("分:")
    m=int(m)
    if m<0 or m>59:
        raise Exception("无效的分")
    s=input("秒:")
    s=int(s)
    if s<0 or s>59:
        raise Exception("无效的秒")
    print("%02d:%02d:%02d"%(h,m,s))
try:
    myTime()
except Exception as e:
    print(e)
```

执行时如果输入的时间正确，则显示该时间，例如：

```
时:23
分:12
秒:34
23:12:34
```

执行时如果输入的时间错误，则抛出异常，例如：

```
时:24
无效的时
```

3.6 模块

3.6.1 导入模块

模块是一个包含变量、函数或类的程序文件，模块中也可包含其他 Python 语句。

模块需要先导入，然后才能使用其中的变量、函数或者类等。可使用 import 或 from 语句导入模块，基本格式为

```
import 模块名称
import 模块名称 as 新名称
from 模块名称 import 导入对象名称
from 模块名称 import 导入对象名称 as 新名称
from 模块名称 import *
```

1. import 语句

import 语句用于导入整个模块，可用 as 为导入的模块指定一个新名称。导入模块后，可以使用“模块名称.对象名称”的格式来引用模块中的对象。例如：

```
>>>import math                          #导入模块
>>>math.fabs(-5)                        #调用模块中的函数
5.0
>>>math.e                               #使用模块中的常量
2.718281828459045
>>>fabs(-5)                             #试图直接使用模块中的函数，出错
Traceback(most recent call last):
  File "<stdin>",line 1,in <module>
NameError:name'fabs'is not defined
>>>import math as m                     #导入模块并指定新名称
>>>m.fabs(-5)                           #通过新名称调用模块函数
5.0
>>>m.e                                  #通过新名称使用模块常量
2.718281828459045
```

2. from 语句

from 语句用于导入模块中的指定对象，导入的对象可直接使用，不需要使用模块名称作为限定符，示例代码如下：

```
>>>from math import fabs                #从模块导入指定函数
>>>fabs(-5)
5.0
>>>from math import e                   #从模块导入指定常量
>>>e
2.718281828459045
>>>from math import fabs as f1          #导入时指定新名称
>>>f1(-10)
10.0
```

3. from … import * 语句

使用星号时，可导入模块顶层的所有全局变量和函数。

示例代码如下：

```
>>>from math import *                    #导入模块顶层的全局变量和函数
>>>fabs(-5)                              #直接使用导入的函数
5.0
>>>e                                     #直接使用导入的常量
2.718281828459045
```

3.6.2　导入时执行模块

import 和 from 语句在执行导入操作时，会执行导入模块中的全部语句。只有执行了模块，模块中的变量和函数才会被创建，才能在当前模块中使用。只有在第一次执行导入操作时，才会执行模块；再次导入模块时，并不会重新执行模块。

import 和 from 语句是隐性的赋值语句，两者的区别如下。

- Python 执行 import 语句时，会创建一个模块对象和一个与模块文件同名的变量，并建立变量和模块对象的引用；模块中的变量和函数等均作为模块对象的属性使用；再次导入时，不会改变模块对象属性的当前值。
- Python 执行 from 语句时，会同时在当前模块和被导入模块中创建同名变量，这两个变量引用了同一个对象；再次导入时，会将被导入模块的变量的初始值赋给前一个模块的变量。

示例代码如下：

```
#模块文件 test.py
x=100                                   #赋值，创建变量 x
#输出字符串
print('这是模块 test.py 中的输出!')
#定义函数，执行时创建函数对象
def show():
    print('这是模块 test.py 中 show()函数中的输出!')
```

可以将 test.py 放在项目章节对应的目录中，然后进入系统命令提示符窗口，在章节对应的目录中执行 python.exe 进入 Python 交互环境。

示例代码如下：

```
>>>import test                          #导入模块，下面的输出说明模块在导入时被执行
这是模块 test.py 中的输出!
>>>test.x                              #使用模块变量
100
>>>test.x=200                          #为模块变量赋值
>>>import test                         #重新导入模块
>>>test.x                              #使用模块变量，输出结果显示重新导入未影响变量的值
200
>>>test.show()                         #调用模块函数
这是模块 test.py 中 show()函数中的输出!
```