

4.函数

4.1 函数的概念

如果在开发程序时，需要某块代码多次，但是为了提高编写的效率以及代码的重用，所以把具有独立功能的代码块组织为一个小模块，这就是函数。

4.2 函数定义和调用

4.2.1 定义函数

定义函数的格式如下：

```
def 函数名():  
    代码
```

demo:

```
# 定义一个函数，能够完成打印信息的功能  
def printInfo():  
    print '-----'  
    print '          人生苦短，我用Python'  
    print '-----'
```

4.2.2 调用函数

定义了函数之后，就相当于有了一个具有某些功能的代码，想要让这些代码能够执行，需要调用它

调用函数很简单的，通过 **函数名()** 即可完成调用

demo:

```
# 定义完函数后，函数是不会自动执行的，需要调用它才可以  
printInfo()
```

4.3 函数参数

4.3.1 定义带有参数的函数

示例如下：

```
def add2num(a, b):  
    c = a+b  
    print c
```

4.3.2 调用带有参数的函数

以调用上面的add2num(a, b)函数为例:

```
def add2num(a, b):           #在“函数定义”的时候，a, b称为“形式参数”
    c = a+b
    print c

add2num(110, 22)            #在“函数调用”的时候，110和22称为“实际参数”。
                             #调用带有参数的函数时，需要在小括号中，传递数据
```

调用带有参数函数的运行过程:

➡ #定义接收2个参数的函数

```
def add2num( a, b):
    c = a+b
    print c
```

#调用带有参数的函数

```
add2num(110, 22)
```

终端

函数作为一个程序块，也会遵循一个规律:

输入 -> 处理 -> 输出 (中间的运算数值，存储在内存中)

函数定义的格式:

```
def 函数名 (形式参数):      #形式参数，是代码段中临时定义的变量，指代“输入进来的数值”
    函数体 (具体的功能性代码)
    return 返回值           #返回值，是代码段执行后，返回到函数调用处的“输出结果”
```

函数调用的格式:

```
变量名 = 函数名 (实际参数)  #变量名 所指向的最终结果是函数执行后返回的“输出结果”
```

4.4 函数返回值

4.4.1 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 add2num(a, b):  
    c = a+b  
    return c
```

或者

```
def add2num(a, b):  
    return a+b
```

4.4.2 保存函数的返回值

如果一个函数返回了一个数据，那么想要用这个数据，那么就需要保存

保存函数的返回值示例如下：

```
#定义函数  
def add2num(a, b):  
    return a+b  
  
#调用函数，顺便保存函数的返回值  
result = add2num(100,98)  
  
#因为result已经保存了add2num的返回值，所以接下来就可以使用了  
print result
```

4.4.3 “返回”即“结束”return所在的函数

return“返回”了结果；return结束了函数的执行

例如：

```
def subtraction(a,b):  
    # if a<b:  
    #     return b-a  
    # else:  
    #     return a-b  
    if a<b:  
        print("a<b")  
        return b-a    #返回调用自身函数的地方  
    print("a>=b")    #如果上面执行了return语句，后面的代码将不再执行  
    return a-b  
  
print(subtraction(-3,5))  
print(subtraction(5,-3))
```

4.4.4 函数“调用”的嵌套

```
def add2Num(a,b):  
    return a+b  
  
def add3Num(a,b,c):  
    #sum = a+b+c  
    return add2Num(add2Num(a,b),c)    #可以直接将add2Num(a,b)返回的结果作为另一个函数的实参  
  
print(add3Num(10,20,30))
```

课堂练习:

练习1: 根据输入的数量, 打印对应行数的线

1. 写一个打印一条横线的函数。(提示: 横线是若干个“-”组成)
2. 写一个函数, 可以通过输入的参数, 打印出自定义行数的横线。(提示: 调用上面的函数)

练习2: 求3个数值的平均值

1. 写一个函数求三个数的和
2. 写一个函数求三个数的平均值 (提示: 调用上面的函数)

【建议每题15分钟以内】

参考代码1:

```
# 打印一条横线  
def printOneLine():  
    print("-"*30)  
  
# 打印多条横线  
def printNumLine(num):  
    i=0  
  
    # 因为printOneLine函数已经完成了打印横线的功能,  
    # 只需要多次调用此函数即可  
    while i<num:  
        printOneLine()  
        i+=1  
  
printNumLine(3)
```

参考代码2:

```

# 求3个数的和
def sum3Number(a,b,c):
    return a+b+c          # return 的后面可以是数值，也可是一个表达式

# 完成对3个数求平均值
def average3Number(a,b,c):

    # 因为sum3Number函数已经完成了3个数的求和，所以只需调用即可
    # 即把接收到的3个数，当做实参传递即可
    aveResult = sum3Number(a,b,c)/3.0
    return aveResult

# 调用函数，完成对3个数求平均值
result = average3Number(11,2,55)
print("average is %d"%result)

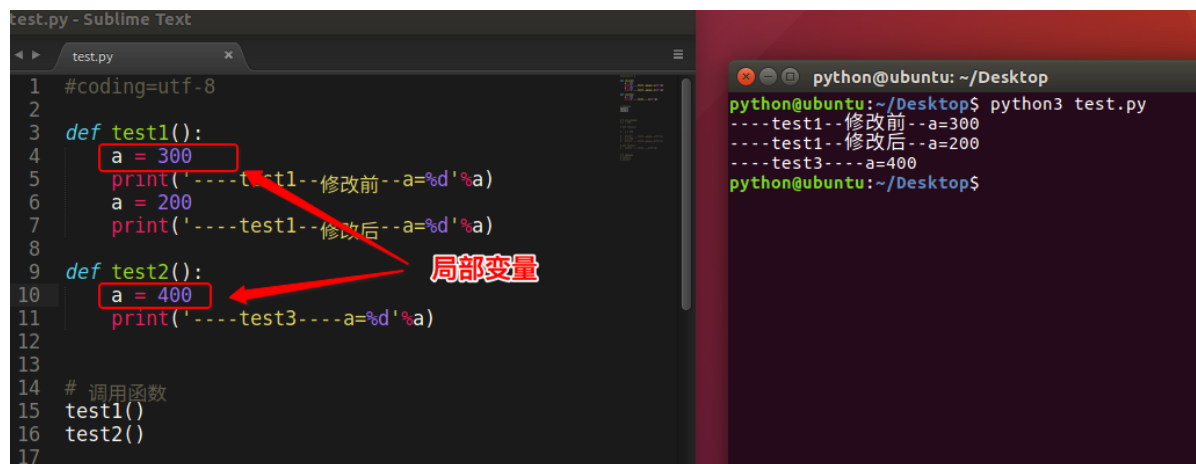
```

4.5 局部变量和全局变量

4.5.1 局部变量

什么是局部变量

如下图所示:



小总结

- 局部变量，就是在函数内部定义的变量
- 不同的函数，可以定义相同的名字的局部变量，但是自己用自己的不会产生影响
- 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，这就是它的作用

4.5.2 全局变量

什么是全局变量

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

demo如下:

```
# 定义全局变量
a = 100

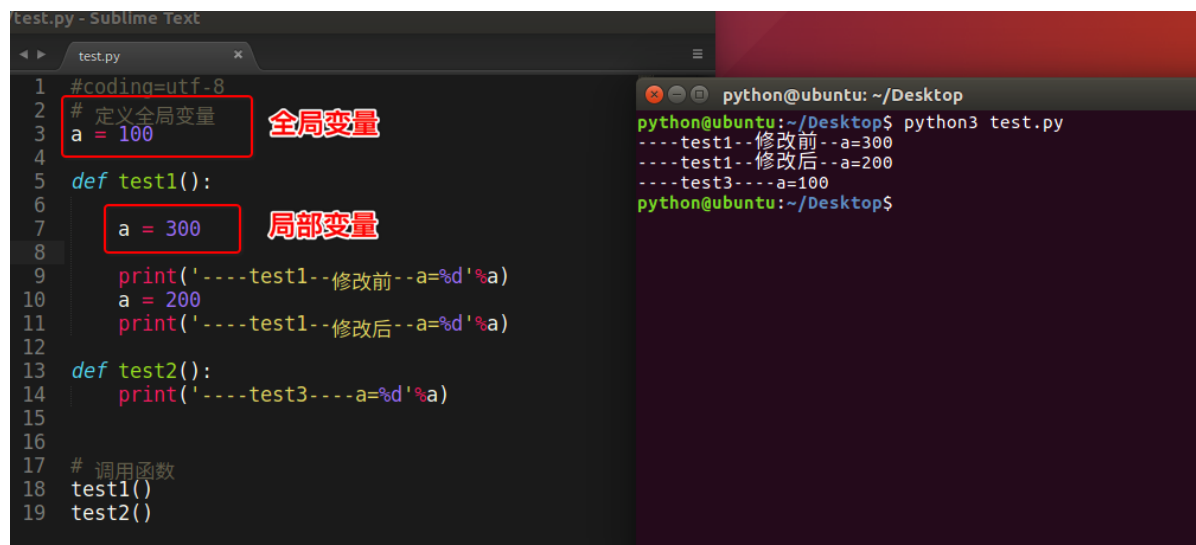
def test1():
    print(a)

def test2():
    print(a)

# 调用函数
test1()
test2()
```

全局变量和局部变量名字相同问题

看如下代码:



The screenshot shows a code editor window titled 'test.py - Sublime Text' and a terminal window. The code in the editor is as follows:

```
1 #coding=utf-8
2 # 定义全局变量
3 a = 100
4
5 def test1():
6     a = 300
7
8     print('----test1--修改前--a=%d'%a)
9     a = 200
10    print('----test1--修改后--a=%d'%a)
11
12 def test2():
13    print('----test3----a=%d'%a)
14
15 # 调用函数
16 test1()
17 test2()
```

Red boxes and labels highlight the variable 'a' in the code:

- A red box around line 3, `a = 100`, is labeled **全局变量** (Global Variable).
- A red box around line 6, `a = 300`, is labeled **局部变量** (Local Variable).

The terminal window shows the output of running the script:

```
python@ubuntu: ~/Desktop
python@ubuntu:~/Desktop$ python3 test.py
----test1--修改前--a=300
----test1--修改后--a=200
----test3----a=100
python@ubuntu:~/Desktop$
```

修改全局变量

既然全局变量，就是能够在所有的函数中进行使用，那么可否进行修改呢？

代码如下:

```
test.py - Sublime Text
1 #coding=utf-8
2 # 定义全局变量
3 a = 100
4
5 def test1():
6
7     global a
8
9     print('----test1--修改前--a=%d'%a)
10    a = 200
11    print('----test1--修改后--a=%d'%a)
12
13 def test2():
14     print('----test3----a=%d'%a)
15
16
17 # 调用函数
18 test1()
19 test2()
```

```
python@ubuntu: ~/Desktop
python@ubuntu:~/Desktop$ python3 test.py
----test1--修改前--a=100
----test1--修改后--a=200
----test3----a=200
python@ubuntu:~/Desktop$
```

程序ok,
没有出错

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
- 如果在函数中修改全局变量，那么就需要使用 `global` 进行声明，否则出错
- 如果全局变量的名字和局部变量的名字相同，那么使用的是局部变量的，小技巧 **强龙不压地头蛇**

4.6 函数使用注意事项

1. 自定义函数

<1>无参数、无返回值

```
def 函数名():
    语句
```

<2>无参数、有返回值

```
def 函数名():
    语句
    return 需要返回的数值
```

注意:

- 一个函数到底有没有返回值，就看有没有 `return`，因为只有 `return` 才可以返回数据
- 在开发中往往根据需求来设计函数需不需要返回值
- 函数中，可以有多个 `return` 语句，但是只要执行到一个 `return` 语句，那么就意味着这个函数的调用完成

<3>有参数、无返回值

```
def 函数名(形参列表):  
    语句
```

注意:

- 在调用函数时, 如果需要把一些数据一起传递过去, 被调用函数就需要用参数来接收
- 参数列表中变量的个数根据实际传递的数据的多少来确定

<4>有参数、有返回值

```
def 函数名(形参列表):  
    语句  
    return 需要返回的数值
```

<5>函数名不能重复

2. 调用函数

<1>调用的方式为:

```
函数名([实参列表])
```

<2>调用时, 到底写不写 实参

- 如果调用的函数 在定义时有形参, 那么在调用的时候就应该传递参数

<3>调用时, 实参的个数和先后顺序应该和定义函数中要求的一致

<4>如果调用的函数有返回值, 那么就可以用一个变量来进行保存这个值

3. 作用域

<1>在一个函数中定义的变量, 只能在本函数中用(局部变量)

<2>在函数外定义的变量, 可以在所有的函数中使用(全局变量)

5.函数 (进阶)

不同标签所对应的学习目标:

- 【补充】: 对知识点的补充, 扩大知识面
- 【进阶】: 对知识点的深入理解, 增加应用能力
- 【扩展】: 从业者标准, 需要了解的知识

5.1 函数的概念

函数的命名，和变量名规则相同，同样建议：见名知意。

```
函数定义
def func():
    pass          # pass是空语句，占位作用
    print("函数被调用了")

# 函数调用
func()

print(id(func), type(func))      #函数名本身也查看地址（Hash值），以及类型
```

函数名也可以作为变量进行赋值\传递\存储

```
def func1():
    print("func1...")

def func2():
    print("func2...")

def func3():
    print("func3...")

res = func1          #函数名本身也是引用类型，保存的是函数体所在地址空间的内存地址（hash值）

print(id(res), id(func1))

res()                #指向函数体的变量名，可以通过追加小括号，实现函数的调用

myfuncs = [func1, func2, func3]      #函数名可以作为元素，存储在列表中等容器类型中

for f in myfuncs:
    f()                  #可以进行迭代
                        #通过括号实现列表中函数逐一调用的效果
```

5.2 参数

5.2.1 标准参数

（定义时）默认的形式参数，与调用时的位置，逐一对应赋值。

（调用时）标准参数：主要分为“位置参数”和“默认参数”两种方式。

位置参数：

```
def printInfo(a, b):
    print(a, b)

#printInfo(10)          #报错
printInfo(10, 20, 30)    #使用位置参数时，实参、形参数量要一致；否则报错
```

默认参数:

(定义时) 形式参数指定默认值。

(调用时) 可以和位置参数一样进行赋值, 也可以不进行赋值; 不赋新值, 则为默认值。

```
def printInfo(name, age=18):  
    print(f"姓名: {name}, 年龄: {age}")  
  
printInfo("吴彦祖", 19)  
printInfo("彭于晏")      #调用时, 默认参数可以不赋值  
  
def printInfo(name="帅哥", age): #报错: SyntaxError: non-default argument follows  
    default argument  
    print(f"姓名: {name}, 年龄: {age}")  
  
printInfo(20)
```

注意: “默认参数”只能在“位置参数”后, 进行定义。

指定参数名赋值

指的是函数调用时, 根据参数名, 进行针对性赋值

```
def printInfo(name="帅哥", age=18, gender="男"):  
    print(f"姓名: {name}, 年龄: {age}, 性别: {gender}")  
  
# printInfo(name="金城武", age=20)  
printInfo(age=20, name="金城武") # 因为制定了形参变量名, 所以可以不按照位置循序传递参数  
  
printInfo("Samantha", gender="女", age=18)  
#printInfo(gender="女", age=18, "Samantha") #如果 关键字实参 和 位置实参 混合使用, 位置  
实参在前面。否则报错。
```

5.2.2 可变 (个数的) 参数

可变参数: *args : 将多个参数存储为元组对象

关键字参数: **kwargs: 将多个参数存储为字典对象

5.2.2.1 可变参数

函数定义时*args可以接受任意多个实际参数, args可以替换为任意变量名, 符合变量名命名规则就可以

```
def printInfo(name, age, gender, *args):  
    print(f"姓名: {name}, 年龄: {age}, 性别: {gender}")  
    #print(args, type(args))  
    print("作品评分自高到低为: ", args)  
  
# 他所演过的电视剧/电影评分  
printInfo("彭于晏", 28, "男", 8.9, 8.3, 8.1)
```

输出内容为:

姓名：彭于晏，年龄：28，性别：男
作品评分自高到低为：(8.9, 8.3, 8.1)

5.2.2.2 关键字参数

函数定义时，**kwargs可以接受任意多个键值对形式的实际参数。

```
def printInfo(name,age,gender,**kwargs):  
    print(f"姓名: {name}, 年龄: {age}, 性别: {gender}")  
    print(kwargs,type(kwargs))  
  
#身高, 国籍, 出生地, 学历  
printInfo("彭于晏",20,"男",身高=1.82,国籍="中国",出生地="台湾")
```

输出内容为：

姓名：彭于晏，年龄：20，性别：男
{'身高': 1.82, '国籍': '中国', '出生地': '台湾'} <class 'dict'>

5.2.2.3 参数解包

在“函数调用”时，*和**的作用是：“解包”。

- 符号 * 将传进来的字符串、元组、列表、集合转化为多个标准参数
- 符号 ** 将传进来的字典，转化为多个关键字参数

```
def printInfo(name,age,gender,*args):  
    print(f"姓名: {name}, 年龄: {age}, 性别: {gender}")  
    print("作品评分自高到低为: ",args,type(args))  
  
rate = [8.9,8.3,8.1]  
rate = (8.9,8.3,8.1)  
rate = {8.9,8.3,8.1}    #无序  
  
printInfo("彭于晏",28,"男",*rate)
```

输出内容为：

姓名：彭于晏,年龄：28,性别：男
作品评分自高到低为：(8.1, 8.9, 8.3) <class 'tuple'>

5.2.2.4 参数传递顺序

函数调用时，参数传递的顺序：

(调用时) 实际参数：位置参数、关键字参数

(定义时) 匹配形式参数顺序：args、*args、**kwargs

```
def testArgs(a,b,c=10,d=20,*args,**kwargs):
    print(f"a={a},b={b},c={c},d={d},args={args},kwargs={kwargs}")

testArgs(1,2,3,c=4,5,6,7,x=100,y=200) #报错。SyntaxError: positional argument
follows keyword argument

# 关键字参数之后不能再使用位置参数（只能使用关键字
参数）

def testArgs(*args,a,b,c=10,d=20,**kwargs):    #可变参数在最前面时，匹配到关键字赋值的
其他实际参数
    print(f"a={a},b={b},c={c},d={d},args={args},kwargs={kwargs}")

testArgs(1,2,3,4,5,a=6,c=100,b=7,x=100,y=200)
```

输出内容为：

```
a=6,b=7,c=100,d=20,args=(1, 2, 3, 4, 5),kwargs={'x': 100, 'y': 200}
```

5.2.3 命名关键字参数

*后面的参数，被称为“命名关键字参数”

```
def testStar(a,b,c,* ,name,age):
    print(f"a={a},b={b},c={c},name={name},age={age}")

testStar(1,2,3,name="吴彦祖",age=30) # *的作用，表示*后面的参数必须使用命名的方式来进行参
数赋值

#name 和 age要使用命名的方式来赋值
```

输出内容为：

```
a=1,b=2,c=3,name=吴彦祖,age=30
```

5.3 返回值

5.3.1 返回值类型

函数可以返回全部数据类型，没有返回值时，默认返回为None

```
def test():
    pass
    #return "abc"
    #return 2.0
    #return True
    #return 200
    #return ["abc"]
    #return {"a":3}
    #return (3,5)
    #return {1,2,3}
    #return None    #没有返回值时，默认返回为None

res = test()
```

```
print(res,type(res))
```

5.3.2 返回多个值

返回多个值的函数，将需要返回的多个值，封装在列表、字典、元组容器中。

注意：set会改变数据的顺序

```
def divid(a,b):
    shang = a//b
    yushu = a%b
    #return [shang,yushu]
    #return {1,2,3}
    return shang,yushu

#对应上面的 return [shang,yushu] 或 return {1,2,3}
res = divid(5,2)
print(f"商: {res[0]}, 余数: {res[1]}")
print(res,type(res))

#对应上面的 return shang,yushu
print(divid(5,2),type(divid(5,2)))
sh,yu = divid(5,2)
print(f"商: {sh}, 余数: {yu}")
```

输出内容为：

```
商：2，余数：1
```

5.4 局部变量和全局变量

5.4.1 局部变量

局部变量：函数内部定义的变量

```
def test1():
    a = 100          #局部变量
    print("test1-----修改前: ",a,id(a))
    a = 300
    print("test1-----修改后: ", a, id(a))

def test2():
    a = 100          #不同的函数可以定义相同名字的变量，彼此无关
    print("test2-----",a,id(a))

test1()
test2()
#print(a)           #报错，局部变量只在函数内部有效
```

输出内容为：

```
test1-----修改前: 100 1681173984
test1-----修改后: 300 21184112
test2----- 100 1681173984
```

注意：上面id打印的hash值每次运行都会不同，但是test1修改前和test2输出的a的id值是相同的

5.4.2 全局变量

全局变量：在文件内有效，能在多个函数中使用的变量。

```
a = 100          #定义全局变量

def test1():
    print(a)      #获取全局变量a，输出：100

def test2():
    print(a)      #输出：100

test1()
test2()
print(a)         #输出：100
```

在test1和test2的函数中，访问到的a都是100

5.4.3 全局变量和局部变量的作用域

全局变量和局部变量名字相同时，优先使用最近定义的变量。

有局部变量时，优先使用局部变量；没有局部变量时，看是否有全局变量。

```
a = 100
def test1():
    a = 300          #局部变量优先使用
    print("修改前，获取到局部的a: ", a)
    a = 500          #修改局部变量的数值
    print("修改后，获取到局部的a: ", a)

def test2():
    #a = 1000
    print("test2获取到全局的a: ",a)      #没有局部变量，默认使用全局变量

print("全局的a: ",a)
test1()
test2()
print("test2执行后的全局的a: ",a)
```

输出内容为：

```
全局的a: 100
修改前，获取到局部的a: 300
修改后，获取到局部的a: 500
test2获取到全局的a: 1000
```

【如果在test2中定义了局部变量a=1000时显示，否则

为100】

test2执行后的全局的a: 100

5.4.4 在函数中修改全局变量

在函数中修改全局变量前，需要在函数中声明变量为global

被修改的全局变量，在其他函数中访问时数据也会跟着改变。

```
a = 100

def test1():
    global a          #在函数中声明全局变量的关键字: global
    a = 200
    print("test1.....",a)

def test2():
    print("test2中的全局变量: ",a)

print("全局的: ",a)
test1()
print("test1执行后全局的: ",a)
test2()
```

输出内容为:

```
全局的: 100
test1..... 200
test1执行后全局的: 200
test2中的全局变量: 200
```

可以通过local () 和global () 打印局部和全局变量

```
a = 100
# b = 500
def test1():
    a = 200
    global b          #即使上面不定义全局变量b，此处声明起到了定义全局变量b的作用
    b = 500
    print(locals())   #显示函数内的“本地变量”，即为函数的局部变量
    print(globals())  #不论写在哪里，始终显示的都是全局变量

test1()
print(b)             #即使上面不在函数外定义b，此处也能访问到b的数值，因为test1运行过
print(locals())       #显示模块（文件）内的局部变量，即为“全局变量”
print(globals())      #不论写在哪里，始终显示的都是全局变量
```

输出内容为:

```
{'a': 200}
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__':
<_frozen_importlib_external.SourceFileLoader object at 0x01B3BF40>, '__spec__': None,
 '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, '__file__':
'D:/itsishu/python_workspace/demo3/demo3.py', '__cached__': None, 'a': 100, 'test1':
<function test1 at 0x0383C8E0>, 'b': 500}

500
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__':
<_frozen_importlib_external.SourceFileLoader object at 0x01B3BF40>, '__spec__': None,
 '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, '__file__':
'D:/itsishu/python_workspace/demo3/demo3.py', '__cached__': None, 'a': 100, 'test1':
<function test1 at 0x0383C8E0>, 'b': 500}

{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__':
<_frozen_importlib_external.SourceFileLoader object at 0x01B3BF40>, '__spec__': None,
 '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, '__file__':
'D:/itsishu/python_workspace/demo3/demo3.py', '__cached__': None, 'a': 100, 'test1':
<function test1 at 0x0383C8E0>, 'b': 500}
```

5.4.5 形式参数和局部变量

- 对于不可变参数，在函数内，每次都是让局部变量指向新的地址值
所以a = 10，是局部变量a 指向了新数值的新地址，不影响外部的x
- 对于可变参数，在函数内，是传递进来的（原有的）地址值上修改数据内容
所以b.append("bbb"),是修改了（和y相同的）地址空间的列表内容，所以外部参数内容会受影响。

具体概念的理解，可以参考 3.2.4，视频《扩展_深拷贝和浅拷贝》

```
def test(a,b):
    print(f"test函数中的变量值,初始, a={a},b={b}",id(b))
    a = 10
    b.append("bbb")
    print(f"test函数中的变量值, 修改后, a={a},b={b}",id(y))

x = 20
y = ["aaa"]
print(f"调用函数前, x={x},y={y}",id(y))
test(x,y)
print(f"调用函数后, x={x},y={y}",id(y))
```

输出内容为：

```
调用函数前, x=20,y=['aaa'] 55127368
test函数中的变量值,初始, a=20,b=['aaa'] 55127368
test函数中的变量值, 修改后, a=10,b=['aaa', 'bbb'] 55127368
调用函数后, x=20,y=['aaa', 'bbb'] 55127368
```


5.5 递归函数（进阶）

5.5.1 递归的概念（recursion）

假设你在一个电影院，你想知道自己坐在哪一排，但是前面人很多，你懒得去数了，于是你问前一排的人「你坐在哪一排？」，这样前面的人（代号 A）回答你以后，你就知道自己坐在哪一排了——只要把 A 的答案加一，就是自己所在的排了。不料 A 比你还懒，他也不想数，于是他也问他前面的人 B「你坐在哪一排？」，这样 A 可以用和你一模一样的步骤知道自己所在的排。然后 B 也如法炮制。直到他们这一串人问到了最前面的一排，第一排的人告诉问问题的人「我在第一排」。最后大家就都知道自己在哪一排了。



递归函数：

形式：自己调用自己的函数

本质：循环。

通过循环运行相同的运算过程，每次改变输入参数获取返回值参与下一次循环。

【递】：递进、放入

【归】：返回

```

#打印从100到1的整数
for i in range(100,0,-1):
    print(i)

def printNum(n):
    print(n)          #执行的运算
    if n == 1:        #最终结束的条件
        return        #返回的数值
    printNum(n-1)      #printNum(99) #调用自身函数

printNum(100)

```

递归的写法:

- 1.编写函数体 (功能)
- 2.确定结束条件和返回值
- 3.调用函数自身, 修改参数

5.5.2 递归的运算过程

例题：求指定位数的阶乘。

阶乘的概念： $n! = n * (n-1)!$

分析为函数表达式：

$f(1) = 1 \quad n \leq 1$

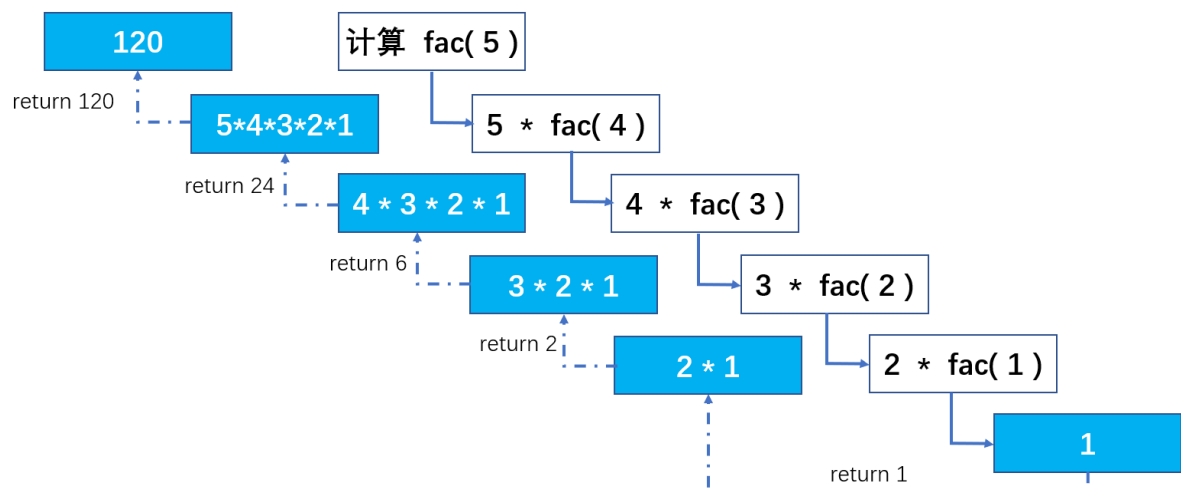
$f(n) = f(n-1) * n \quad n > 1$

```

def fac(n):
    if n <=1 :
        return 1
    return fac(n-1) * n

# 5*4*3*2*1
print(fac(5))          #输出: 120

```



课堂练习

使用递归，根据用户指定的项数，计算斐波那契数列对应位置的值。

斐波那契数列 (Fibonacci sequence)

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144.....

$$f(n) = n \quad n \leq 1$$
$$f(n) = f(n-1) + f(n-2) \quad n > 1$$

【建议用时：20分钟】

参考答案：

使用递归完成

```
def feb(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return feb(n - 1) + feb(n - 2)

#另一种写法:
# if n<=1:
#     return n
# else:
#     return feb(n-1) + feb(n-2)

print(feb(9))
```

使用for循环完成：

```
def feb(n):
    if n<=1:
        return n
    res = 0          #始终存储的和
    feb0 = 0         #用于存储下一次运算的内容
    feb1 = 1         #用于下一次运算的和
    for i in range(2,n+1):
        print(f"res:{res},feb0:{feb0},feb1:{feb1}")
        res = feb0 + feb1
        feb0 = feb1
        feb1 = res
    return res

print(feb(10))
```

6.文件操作

文件，就是把一些数据存放起来，可以让程序下一次执行的时候直接使用，而不必重新制作一份，省时省力。

文件类型：

- 文本类型：以文字存储为主，读写均以“字符”为单位
- 二进制类型：以图形、声音、影像为存储内容的形式，读写均以“字节”为单位。

文件后缀：

- 文本类型：txt、py、doc、docx、pdf、csv、xls、xml、html.....
- 二进制类型：jpg、png、MP3、wav、mp4、mov、avi.....

后缀名主要是为了帮助操作系统识别文件类型，以选择合适的打开方式。

6.1 文件打开与关闭

6.1.1 打开文件

在python，使用open函数，可以打开一个已经存在的文件，或者创建一个新文件

```
open(文件名, 访问模式)
```

示例如下：

```
f = open('test.txt', 'w')
```

6.1.2 关闭文件

```
close()
```

示例如下：

```
f = open('test.txt', 'w')    #打开文件，w模式（写模式）  
  
f.close()                   # 关闭这个文件
```

6.1.3 相对路径和绝对路径

绝对路径：从盘符开始的路径

相对路径：相对当前源码所在的路径

绝对路径中为了不让\产生转义效果，需要在路径前面添加字母 r

否则需要在将每个\进行转义，写为\\

```
f = open(r"D:\itsishu\python_workspace\demo3\test.txt", "w")
f.write("IT sishu")
f.close()
```

6.1.4 中文编码问题

```
f = open(r"D:\itsishu\python_workspace\demo3\test.txt", "w")
f.write("IT私塾")
f.close()

f = open("test.txt", "w", encoding="UTF-8")      #可以通过encoding指定写入中文的字符集
f.write("IT私塾")
f.close()
```

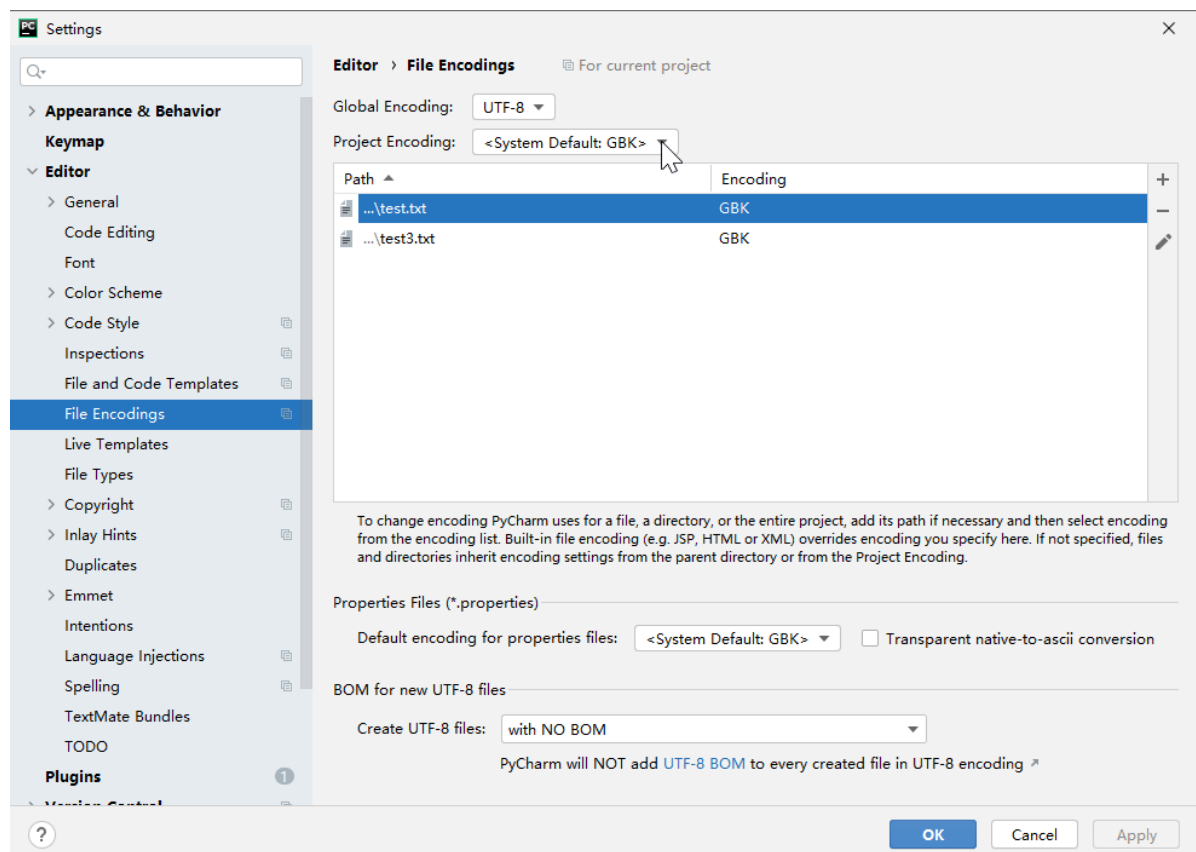
在未指定写入字符集时，运行pycharm中编写的程序，输出中文到文件中。

- 默认使用“写字板”软件打开文件：正常显示（windows系统默认使用GBK字符集）
- pycharm默认打开文件：乱码（默认使用UTF-8字符集）

Pycharm中修改打开文本的默认字符集：

菜单栏选择File -> Setting -> Editor -> File Encodings，可以设置当前项目为GBK或UTF-8；

也可以设置对特定文件读取时的字符集（下方图片右侧蓝色部分）



建议：打开文件时指定字符集，避免乱码问题。

6.2 文件读写

6.2.1 写数据(write)

使用write()可以完成向文件写入数据

demo:

```
f = open('test.txt', 'w')
f.write('hello world, i am here!')
f.close()
```

注意:

- 如果文件不存在那么创建; 如果存在那么就先清空, 然后写入数据

6.2.2 写数据(writelines)

```
f = open("test.txt", "w", encoding="UTF-8")
#写法一:
f.write("IT私塾\n高效学习\n学以致用\n")          #通过换行符实现输出多行效果
#写法二:
content = ["IT私塾\n", "高效学习\n", "学以致用\n"]
f.writelines(content)                             #通过writelines一次性写出列表的每个元素
#写法三:
content = ["IT私塾", "高效学习", "学以致用"]
f.write("\n".join(content))                        #使用字符串的join函数, 为每个元素添加换行符
f.close()
```

6.2.3 读数据(read)

使用read(num)可以从文件中读取数据, num表示要从文件中读取的数据的长度(单位是字符), 如果没有传入num, 那么就表示读取文件中所有的数据

demo:

```
#准备好需要读取的数据文件
f = open("test2.txt", "w", encoding="UTF-8")
f.write("IT私塾, 高效学习")
f.close()

#-----
f = open("test2.txt", "r", encoding="UTF-8")
#data = f.read()          #没有指定读取的字符数, 表示读取文件中所有的数据
#print(data)

data = f.read(2)
print("读取到的文件内容:", data)
data = f.read(4)          #文件读取过程中的指针的定位会向后移动指定"字符数"
print("读取到的文件内容:", data)
f.close()
```

输出内容为:

读取到的文件内容: IT
读取到的文件内容: 私塾, 高

注意:

- 如果open是打开一个文件, 那么可以不用写打开的模式, 即只写 `open('test.txt')`
- 如果使用读了多次, 那么后面读取的数据是从上次读完后的位置开始的

6.2.4 读数据 (readline)

readline, 读取一行, 返回一个字符串。

```
#准备好需要读取的数据文件
f = open("test2.txt", "w", encoding="UTF-8")
f.write("itsishu\n"*10)
f.close()

#-----
f = open("test2.txt", "r", encoding="utf-8")

while True:
    content = f.readline()
    if content:
        print(f"{content}", end="")
    else:
        #print("content:", content, type(content))
        break;

f.close()
```

6.2.5 读数据 (readlines)

就像read没有参数时一样, readlines可以按照行的方式把整个文件中的内容进行一次性读取, 并且返回的是一个列表, 其中每一行的数据为一个元素。

```
f = open("test2.txt", "r", encoding="utf-8")
content = f.readlines()
print(content, type(content))          #输出的类型为列表类型 (list)
#遍历列表中的内容, 并在前面添加行号
#写法一:
# i = 1
# for data in content:
#     print(f"{i}:{data}", end="")
#     i+=1

#写法二:
for i, data in enumerate(content):
    print(f"{i}:{data}", end="")

f.close()
```

6.2.6 所在位置 (tell)

tell, 返回指针当前所在的位置 (指针所在位置前面的字节数)。

```
#准备好需要读取的数据文件
f = open("test3.txt", "w", encoding="gbk")
f.write("IT私塾, 高效学习1")
f.close()

#-----
f = open("test3.txt", "r", encoding="gbk")
content = f.read(2)
print(content)
content = f.read(4)
print(content)
print("当前指针所在位置: ", f.tell())      #输出内容为: 14 或 10
#UTF-8中1个汉字3个字节, GBK中1个汉字为2个字节
#同样是读取2个字符, UTF-8返回为14, GBK返回为10
f.close()
```

6.2.7 定位 (seek)

seek, 定位文件读取的指针所在位置 (字节)

seek的语法规则:

```
seek(offset[, whence])
    offset -- 开始的偏移量, 也就是代表需要移动偏移的字节数
    whence: 可选, 默认值为 0。
    给offset参数一个定义, 表示要从哪个位置开始偏移;
    0代表从文件开头开始算起, 1代表从当前位置开始算起, 2代表从文件末尾算起。
```

```
#准备好需要读取的数据文件
f = open("test3.txt", "w", encoding="gbk")
f.write("IT私塾, 高效学习1")
f.close()

#-----
f = open("test3.txt", "r", encoding="gbk")
content = f.read(2)      #定位在2个字节后 (一个英文字母占1个字节)
print(content)
f.seek(6)                #直接定位在6个字节后 (GBK字符集中一个中文算2个字节)
content = f.read(3)      #读取三个字符 (中文的逗号占2个字节)
print(content)
print("当前指针所在位置: ", f.tell())
```

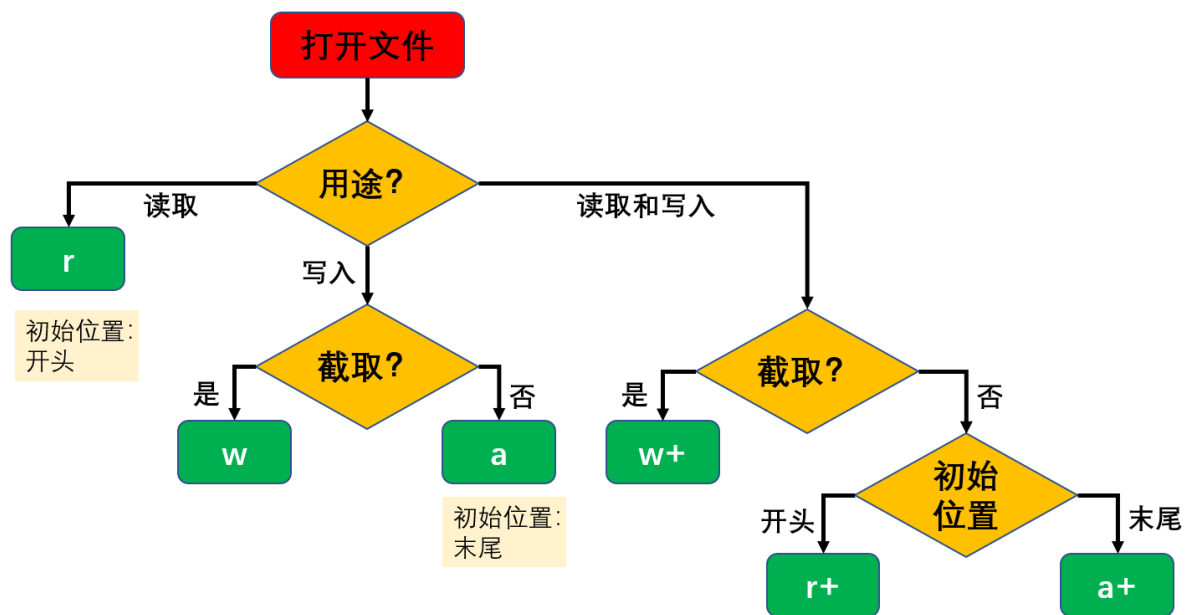
输出内容为:

```
IT
, 高效
当前指针所在位置: 12
```


6.2.8 访问模式

```
open("text.txt",mode="r+")
```

#mode 设定的就是访问模式，决定可以进行的文件操作能力



说明:

当文件不存在时，访问模式w和a会新建文件，r会报错：FileNotFoundError: [Errno 2] No such file or directory

```
#准备好需要读取的数据文件
f = open("test4.txt","a",encoding="utf8")
f.write("IT私塾")
f.close()

#-----
# r+ 可读可写
f = open("test4.txt","a+",encoding="utf8")
# f.seek(0,2)          #seek的模式2，为追加模式。在文件末尾读取或写入。
# data = f.read()
# print(data)

# f.seek(0)            #定位在内容头部，会将现有内容覆盖重写
f.write("学以致用")

f.close()
```

更多访问模式的说明:

访问模式	说明
r	以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。
w	打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
a	打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
rb	以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。
wb	以二进制格式打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
ab	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
r+	打开一个文件用于读写。文件指针将会放在文件的开头。
w+	打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
a+	打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会是追加模式。如果该文件不存在，创建新文件用于读写。
rb+	以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。
wb+	以二进制格式打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
ab+	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文件不存在，创建新文件用于读写。

课堂练习：

1. 应用文件操作的相关知识，定义一个函数。通过给定的文件名（例如：gushi.txt），新建文件后将一首古诗（自选）写入文件中。
2. 另外写一个函数：根据指定读取的文件名（例如：gushi.txt），将内容复制到copy.txt中，并在控制台输出“复制完毕”。

提示：定义两个函数，分别完成读文件和写文件的操作

在主程序中调用两个函数，完成上述任务。

6.2.9 二进制读写

二进制文件的读写（主要应用在图片、音乐、视频等文件的读写）

- rb: read binary 读取二进制文件
 - wb: write binary 写入二进制文件
- 将文件的内容单纯的用0和1来进行读取和存储。不用指定字符集。

案例：复制图片

```
fin = open("luo1uo.jpg",mode="rb")
fout = open("luo1uo_copy2.jpg",mode="wb")

data = fin.read(100)
print(data,type(data))

#输出的内容以b开头表示bytes（字节）方式读取的字符串，是以十六进制表示的内容

while True:
    data = fin.read(100)      #每次读取100个字符（自定义的缓冲区大小）
    if data:
        fout.write(data)
    else:
        break

while True:
    data = fin.read(100)
    if data != b"":          #此处需要注意为b""
        fout.write(data)
    else:
        break

fin.close()
fout.close()
```

6.3 文件对象的函数和属性

6.3.1 flush() 刷新缓冲区

文件写入过程：

数据 --> 缓冲区（内存中） --> 文件中（硬盘上）

【flush函数】刷新（清空）缓冲区

```
f = open("test5.txt",mode="w+",encoding="utf-8")
f.write("IT私塾")
f.flush()      #flush函数：刷新缓冲区
while True:
    pass

f.close()
```

如果没有flush函数，上面的代码将死循环的过程中断后，文件内没有文字输出；

有了flush函数，中断死循环后，文件内有文字输出。

缓冲区清空的时间节点：

- 当文件关闭的时候，自动清空缓冲区
- 当整个程序运行结束的时候自动清空缓冲区
- 当缓冲区写满了，会自动清空缓冲区
- 手动清空缓冲区

6.3.2 truncate() 截断文件

truncate函数，从指针位置到结束的字符全部删除掉，只保留指针之前的内容。

```
#准备好需要读取的数据文件
f = open("test5.txt",mode="w+",encoding="utf-8")
for i in range(5):
    f.write("IT私塾"+str(i)+"\n")          #11个字符
f.close()
#-----
f = open("test5.txt",mode="r+",encoding="utf-8")
# f.seek(24)
# f.truncate()                          #从指针位置到结束的字符全部删除掉，只保留之前的内容
f.truncate(8)
f.seek(0)
print(f.read())
f.close()
```

运行完毕上述内容，文件test5.txt中剩余文字为：

IT私塾

6.3.3 文件对象

- 文件对象，是一个可迭代对象；可以直接通过遍历访问到每一行内容。

```
f = open("test5.txt","r",encoding="utf-8")

for line in f:
    print(line,end="")

f.close()
```

- 可以通过其中的函数获取文件的属性和状态

```
f = open("test5.txt","r")
print("文件名: ",f.name)
print("文件打开的模式",f.mode)
print("文件可写: ",f.writable())
print("文件可读: ",f.readable())
```

输出内容为：

文件名: test5.txt
文件打开的模式 r
文件可写: False
文件可读: True

7. 错误与异常

错误和异常的概念:

- 错误 (Error)

```
print(a)                                     #NameError: name 'a' is not defined
```

指的是: 语法错误, 或者解析错误。

- 异常 (Exception)

```
a = input("请输入一个整数")    #输入a
print(int(a))                  #ValueError: invalid literal for int() with base
5: 'a'
```

指的是: 运行期检测到的错误被称为异常。

为什么要进行异常处理?

为了让我们的程序在运行过程中, 遇到一些“问题”(可以预知, 但不能无视)的时候, 仍然能够根据问题进行处理, 让程序能够继续的运行下去。

场景:

面对用户输入被除数为0, 是程序崩溃、终止? 还是给出用户提示, 让程序继续运行。

用户输入错误, 就是“异常”; 我们对这样的情况进行处理, 就是“异常处理”。

案例:

```
a = input("请输入一个整数: ")
print(int(a))

n = input("请输入被除数: ")
m = input("请输入除数: ")
res = float(n)/float(m)
print("res:", res)
```

上述代码在用户输入不同内容时，会产生不同的错误提示：

- 情况1：用户输入的不是数字，无法计算，例如：a,b。 错误类型：ValueError: could not convert string to float
- 情况2：用户输入的除数为0。 错误类型：ZeroDivisionError: float division by zero

2个处理步骤：

准备：预知错误的情况下，充分的条件判断 【业务逻辑代码】

等待：准备错误预案，出错后进行异常处理 【异常处理代码】

可以通过充分的条件判断，让程序继续运行，例如：

```
n = input("请输入被除数：")
m = input("请输入除数：")
if n.isdigit() and m.isdigit():      #条件判断
    if m != "0":                      #条件判断
        res = float(n)/float(m)      #业务代码
        print("res:", res)
    else:
        print("除数不能为0")        #异常处理代码
else:
    print("请输入数字")
print("程序继续运行并正常结束")
```

问题：业务逻辑和异常处理的代码混在一起，不利于代码的维护和表达

7.1 异常简介

看如下示例：

```
print '-----test--1---'
open('123.txt','r')
print '-----test--2---'
```

运行结果：

```
code@ubuntu:~/python-test$ python test.py
-----test--1---
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    open('123.txt','r')
IOError: [Errno 2] No such file or directory: '123.txt'
```

说明：

打开一个不存在的文件123.txt，当找不到123.txt 文件时，就会抛出给我们一个IOError类型的错误，No such file or directory: 123.txt （没有123.txt这样的文件或目录）

异常：

当Python检测到一个错误时，解释器就无法继续执行了，反而出现了一些错误的提示，这就是所谓的“异常”

7.2 捕获异常

7.2.1 try...except...

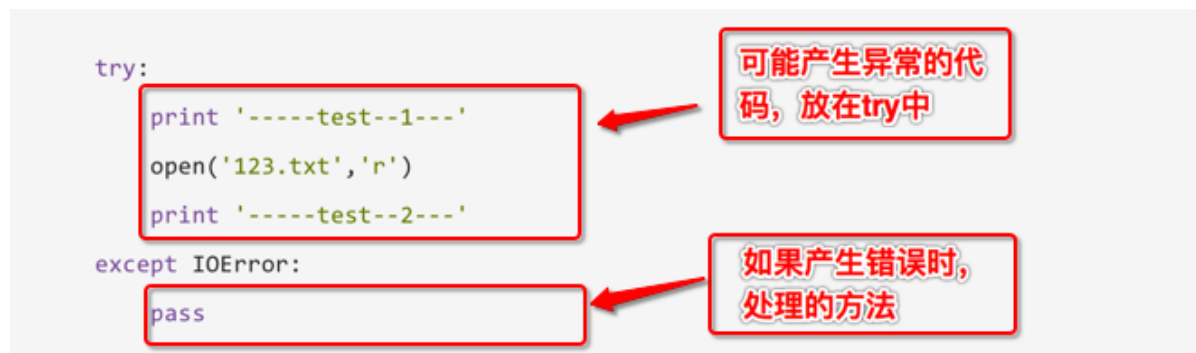
看如下示例:

```
try:
    print('-----test--1---')
    open('123.txt', 'r')
    print('-----test--2---')
except IOError:
    pass
```

说明:

- 此程序看不到任何错误, 因为用except 捕获到了IOError异常, 并添加了处理的方法
- pass 表示实现了相应的实现, 但什么也不做; 如果把pass改为print语句, 那么就会输出其他信息

小总结:



- 把可能出现问题的代码, 放在try中
- 把处理异常的代码, 放在except中

7.2.2 except捕获多个异常

看如下示例:

```
try:
    print num
except IOError:
    print('产生错误了')
```

想一想:

上例程序, 已经使用except来捕获异常了, 为什么还会看到错误的信息提示?

答:

except捕获的错误类型是IOError, 而此时程序产生的异常为 NameError, 所以except没有生效

修改后的代码为:

```
try:
    print num
except NameError:          #异常类型想要被捕获，需要一致
    print('产生错误了')
```

实际开发中，捕获多个异常的方式，如下：

```
#coding=utf-8
try:
    print('-----test--1---')
    open('123.txt','r')      # 如果123.txt文件不存在，那么会产生 IOError 异常
    print('-----test--2---')
    print(num)              # 如果num变量没有定义，那么会产生 NameError 异常

except (IOError,NameError):
    #如果想通过一次except捕获到多个异常可以用一个元组的方式
```

注意：

- 当捕获多个异常时，可以把要捕获的异常的名字，放到except 后，并使用元组的方式仅进行存储

如果需要对于不同的异常类型，进行不同的异常处理操作，可以使用多个except：

```
#1.输入不是数字，提示：请输入数字
#2.除数为0，提示：除数不能为0
try:
    n = input("请输入被除数： ")
    m = input("请输入除数： ")
    res = float(n)/float(m)
    print("res:",res)
except ValueError:
    print("请输入数字")          #数值错误
except ZeroDivisionError:
    print("除数不能为0")        #除零错误
print("程序继续运行并正常结束")
```

7.2.3 获取异常的信息描述

```
In [8]: try:
...:     print(a)
...: except NameError as result
...:     print(result)
...:
name 'a' is not defined
```

**存储异常的
基本信息** → (指向 `as result`)

要捕获的异常 → (指向 `NameError`)


```
In [13]: try:
...:     open("a.txt")
...: except (NameError,IOError) as result:
...:     print("哈哈，捕获到了异常")
...:     print("异常的基本信息是:",result)
```

捕获多个异常，并且存储异常的基本信息

哈哈，捕获到了异常
异常的基本信息是: [Errno 2] No such file or directory: 'a.txt'

```
try:
    n = input("请输入被除数: ")
    m = input("请输入除数: ")
    res = float(n) / float(m)
    print("res:", res)
except Exception as e:
    #print(e,type(e))
    print('str(Exception):\t', str(Exception))
    print('str(e):\t\t', str(e))      #根据错误对象不同，显示其中的错误信息
    print('repr(e):\t', repr(e))      #内置函数repr: 返回一个对象的 string 格式。
```

输出内容为:

```
请输入被除数: 3
请输入除数: 0
str(Exception): <class 'Exception'>
str(e): float division by zero
repr(e): ZeroDivisionError('float division by zero')
```

7.2.4 捕获所有异常

Exception 类型，是异常类型的父类。能够匹配所有异常类型。可以通过as来指向捕获到的异常对象。

```
In [14]: try:
...:     open("a.txt")
...: except:
...:     print("产生了一个异常")
```

没有存储异常的基本信息

产生了一个异常

```
In [15]: try:
...:     open("a.txt")
...: except Exception as result:
...:     print("捕获到了异常")
...:     print(result)
```

捕获所有异常，并且存储异常的基本信息

捕获到了异常
[Errno 2] No such file or directory: 'a.txt'

如果希望根据不同类型的异常进行不同的处理，又希望能够统一处理其他异常的时候，需要将Exception或BaseException放到最后一个。否则优先匹配到最通用的异常类型后，其他（子类）异常类型就不再逐一匹配了。

```

try:
    n = input("请输入被除数: ")
    m = input("请输入除数: ")
    res = float(n)/float(m)
    print("res:",res)

except ValueError:
    print("请输入数字")          #数值错误
except ZeroDivisionError:
    print("除数不能为0")        #除零错误
except BaseException:
    print("出错了")             #注意: 错误类型, 子类在前, 父类在后 (范围小的在前面)

print("程序继续运行并正常结束")

```

7.2.5 try... except... else... finally....

try...finally...语句用来表达这样的情况:

在程序中, 如果一个段代码必须要执行, 即无论异常是否产生都要执行, 那么此时就需要使用 finally。比如文件关闭, 释放锁, 把数据库连接返还给连接池等

demo:

```

try:
    n = input("请输入被除数: ")
    m = input("请输入除数: ")
    res = float(n) / float(m)
    #print("res:", res)
except Exception as e:
    print("出错了")          #出错了, 执行
else:
    print("res:",res)        #没出错, 执行
finally:
    print("用户输入完毕, 计算结束")    #不论程序是否正常运行, finally中的代码都会执行

print("程序继续运行并正常结束")

```

7.2.6 嵌套的异常处理

异常处理是可以嵌套的。

只要有可能发生异常, 并且不希望异常进行“传导”(被调用的时候才进行处理), 就需要进行异常处理。

及时修正错误, 可以保证后续的代码能够继续运行。

```

try:
    f = open("test11.txt", "r", encoding="utf-8")
    for line in f:
        try:
            3/0
        except ZeroDivisionError:
            print("计算错误")

```

```
        print(line,end="")
    except FileNotFoundError:
        print("文件没找到")
    finally:
        try:
            f.close()
        except NameError:
            print("文件没有正常打开，不需要关闭")
```

7.3 异常类型

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
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
LookupError	无效数据查询的基类
IndexError	序列中没有此索引(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
SyntaxError	Python 语法错误
IndentationError	缩进错误

异常名称	描述
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

7.4 with语句 (进阶)

with语句，是应用上下文管理器，实现了文件操作的简化：可以省略f.close() 代码,实现自动关闭。

```
#f = open("test.txt","r",encoding="utf-8")
with open("test.txt","r",encoding="utf-8") as f:          #可以使用with语句打开文件，定义变量f
    for line in f:
        print(line,end="")
```

复制图片，使用with语句实现：

```
with open("luoluo.jpg","rb") as fin:
    with open("luoluo_copy3.jpg","wb") as fout:
        for data in fin:
            fout.write(data)

#简单写法： open 之间可以用逗号,隔开
with open("luoluo.jpg","rb") as fin,open("luoluo_copy4.jpg","wb") as fout:
    for data in fin:
        fout.write(data)
```

7.5 自定义异常 (扩展)

说明：自定义异常，用到了面向对象的知识，建议可以在学习完“面向对象”部分内容后，再来回顾。

raise 关键字的作用是：抛出异常

抛出异常后，所在函数后面的代码不再执行

```
x = 10
if x > 5:
    raise Exception(f'x 不能大于 5.x的值为{x}')

class AgeError(Exception):           #自定义异常类型，继承自Exception类
    def __init__(self,errorInfo):
        Exception.__init__(self)
        self.errorInfo = errorInfo   #定义异常信息

    def __str__(self):                #定义对象输出时的字符串内容
        return str(self.errorInfo)+" ,年龄范围错误! 应该在 1-125 之间"

try:
    age = int(input("输入一个年龄:"))
    if age<1 or age>125:              #自定义发生异常的情况
        raise AgeError(age)          #应用raise关键字，抛出自定义的异常对象
    else: print("正常的年龄: ",age)
except AgeError as e:                #捕获到异常对象并进行异常处理
    print(str(e))
```

7.6 代码调试 (进阶)

7.6.1 错误追踪

```
def a():
    return 1/0

def b():
    a()

def c():
    b()

c()
```

输出内容为：

```
Traceback (most recent call last):
  File "D:/itsishu/python_workspace/demo3/demo6.py", line 199, in
    c()
  File "D:/itsishu/python_workspace/demo3/demo6.py", line 197, in c
    b()
  File "D:/itsishu/python_workspace/demo3/demo6.py", line 194, in b
```

```
a()
File "D:/itsishu/python_workspace/demo3/demo6.py", line 191, in a
    return 1/0
ZeroDivisionError: division by zero
```

上述代码运行结果，可以看出，错误类型为：ZeroDivisionError: division by zero

错误发生的路径依次为：函数c，函数b，函数a

最终发生错误的位置在：a函数的191行代码，return 1/0

针对错误信息的解读，是我们修改和调试代码的基本能力。需要能够快速定位，并识别出错误的位置和类型。

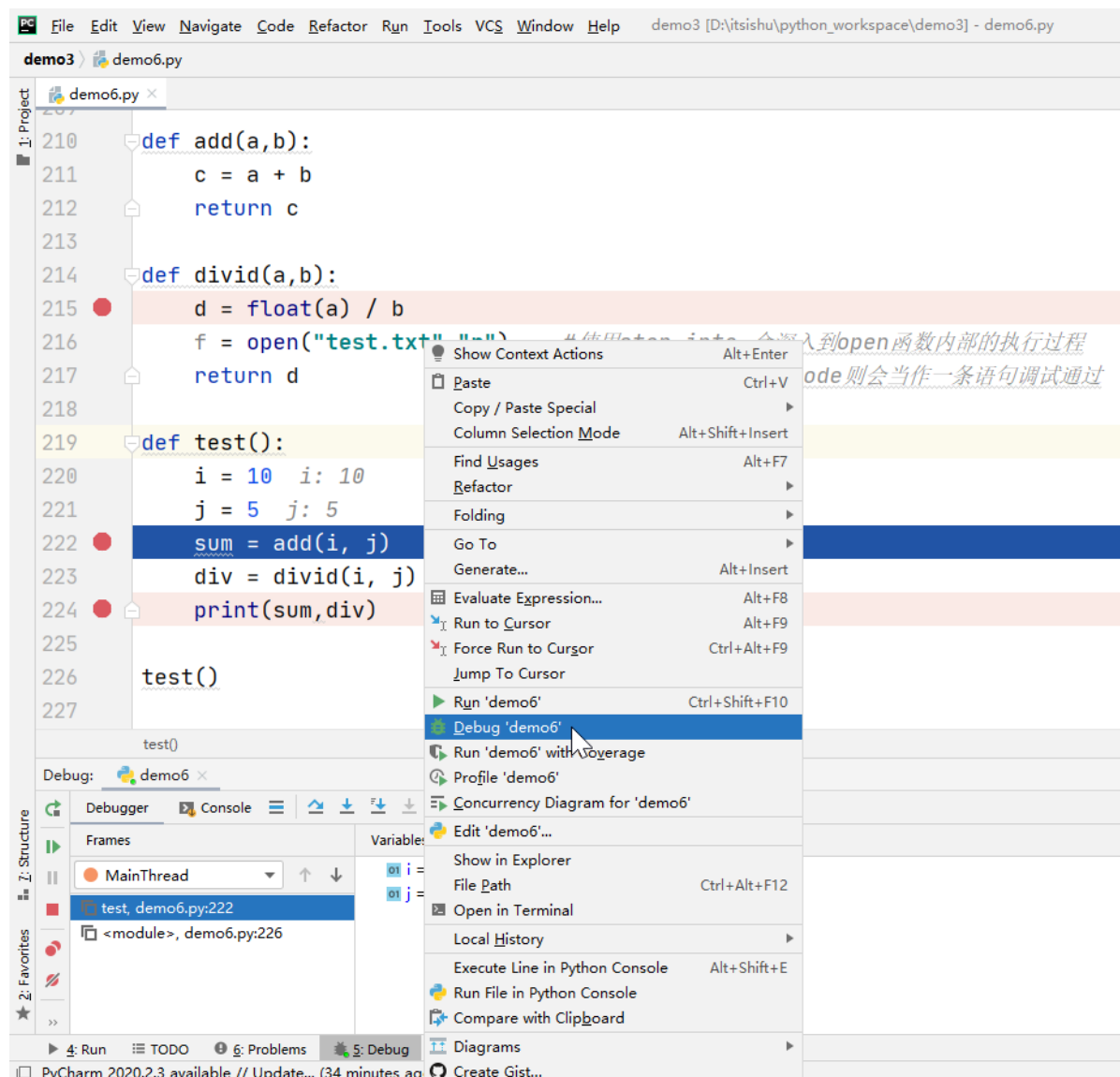
随着经验积累，应该能够具备快速解决问题的能力。

7.6.2 代码调试

代码调试，也称为Debug。应用集成开发环境提供的代码调试工具，辅助我们查看代码运行过程，找到错误，修改代码，是开发人员必不可少的技能。

在Pycharm中的代码编辑区点击鼠标右键，单击“Debug 'XXX'”即可进入调试界面。

程序会根据运行过程，运行暂停在左侧“红点”标记的“断点”处。



根据希望进行的操作，可以使用按键或快捷键，指导程序运行的步骤。

step into: 单步执行，遇到子函数就进入内部，继续单步执行。快捷键：F7

step over: 有子函数时，子函数作为一条语句执行通过（不会深入内部）；其他和单步执行相同。快捷键：F8

step out: 跳出当前所在函数。快捷键：Shift + F8

step into my code: 只调试自己的代码，遇到系统函数，会当作一条语句调试通过；不会深入内部。快捷键：Alt + Shift + F7

run to cursor: 跳到下一个断点。快捷键：Alt + F9

课堂练习：

应用上述调试工具，尝试查看变量赋值过程及代码运行过程。【预计15分钟】

测试代码：

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

def divid(a,b):
    d = float(a) / b
    f = open("test.txt","r")    #使用step into 会深入到open函数内部的执行过程
    return d                   #使用step into my code则会当作一条语句调试通过

def test():
    i = 10
    j = 5
    sum = add(i, j)
    div = divid(i, j)
    print(sum,div)

test()
```