2. 函数的参数与返回值

无参函数就是参数列表为空，即没有参数，例如 print_age()这个函数。

有参函数的参数列表非常重要，定义函数的时候，把各个参数确定下来，函数对外的接口也就确定了。对于函数的调用者来说，只需要知道如何传递正确的参数，以及函数将返回什么样的值就够了，函数内部的复杂逻辑被包裹封装起来，调用者无需了解。

示例：计算 n 的阶乘，即 n!。

分析：

- 定义函数 factorial
- 函数体中利用 if 条件判断输入是否符合要求
- 利用 for 循环计算 1*2*3……（n-1）*n
- 调用函数 factorial(5)，计算 5 的阶乘

```
#定义函数
def factorial(n):
    result = 1
    if n<= 0 or type(n)==float:
        print('输入有误')
    else:
        for i in range(1, n + 1):
            result = result * i
        print('%d!='%n,result)
#调用函数，求 5 的阶乘
factorial(5)
```

Python 的函数定义非常简单，但函数的参数列表确实非常的灵活。除了正常定义的必需的参数外，还可以使用默认参数、可变参数等等，以下是调用函数时可使用的正式参数类型：

- 必需参数
- 默认参数（也称缺省参数）
- 关键字参数
- 不定长参数（也称为可变参数）

在实际开发中，我们极力推荐使用必需参数，这样使得程序更加规范，可读性更好。其他参数形式的函数做了解即可。

（1）必需参数

必需参数就是函数定义时要求的参数，调用时只要以正确的顺序将参数值传入函数即可，要求数量必需和声明时的一样。如前面我们学习的计算阶乘的示例 factorial(n)，要求调用 factorial(n)时必须传入一个值。

示例：计算 x 的 n 次方。

需求：计算 x 的 n 次方，即 x^n=x*x*x*……（n 个 x 相乘），要求使用函数实现代码的重用性。

分析：若要实现代码的重用性，必须 x 和 n 能够根据实际需要自行定义，所以该函数需要定义两个必需参数，x 和 n

```
#x 的 n 次方
def power(x,n):
    result=1
    while n!=0:
        result = result*x
        n = n-1
    return result
print(power(5,3))#5 的 3 次方
```

（2）默认参数

默认参数，也称为缺省参数，就是在定义函数时，参数具有默认值，这样在调用函数时，如果不传递具体的参数值，函数就讲该参数默认为我们规定的那个值。

例如前面计算 n！的示例，如果将函数定义改为如下形式：

```
#默认参数
def factorial(n=5):
```

那么在调用该函数时，就可以不用必需传递参数进入。即如下调用函数的代码是正确的。

```
#默认参数
factorial()
factorial(4)
```

输出结果为：

5!= 120

4!= 24

（3）关键字参数

关键字参数一般用于多个参数的情况，关键字参数和函数调用关系紧密，函数调用使用关键字参数来确定传入的参数值。简单来讲，就是在传递参数时，指定哪个参数传递哪个值。举个例子就明白了。

例如定义并调用函数 power()，指定关键字 n。

```
#关键字参数，定义函数 power，计算 x 的 n 次方
def power(x,n):
……
#调用函数，计算 5 的 3 次方，如下三种调用均正确，注意观察参数顺序
power(5, 3)
power(5,n=3)
power(n=3,x=5) #注意观察参数顺序
```

只用关键字参数，函数参数的使用时可以不指定顺序。

其实在之前的学习中，可能你已经用到了，例如 print()默认是换行输出，如下面指定关键字 end=’ ---’后，不换行，变成横线。

```
#关键字参数
print(str,end=’--- ’)
```

输出结果为：

我爱 Python---我爱编程

（4）不定长参数

不定长参数，也称为可变参数，顾名思义，就是参数的个数是可变的。一般不定长参数定义的形式是这样的。

【语法】

函数名([正常参数值列表,] *不定长参数)

- 使用星号 * 来定义不定长参数
- 不定长参数以元组的形式存在，存放所有未命名的变量参数

例如：

```
#不定长参数
def change(name,*info):
    print(name,info)
change('高飞',20,'北京')
```

输出结果为：

高飞 (20,'北京')

（5）函数的返回值

在使用函数时，有些场景下需要获得函数的执行结果。这个执行结果就是通过给函数添加返回语句，来实现将函数的执行结果返回给函数调用者的。

1.return 关键字

给函数添加返回值可以在需要返回的地方执行 return 语句。return 语句对于函数来说不是必须的，因此函数可以没有返回值。return 关键字的特点是当执行了 return 语句后，就表示函数已经完成执行了，return 后面的语句就不会执行了。

return 关键字后面接的是该函数要返回的数值。这个数值可以是任意类型的数值。当然 return 后面也可以没有任何数值，此时表示希望终止函数的执行。在一个函数中可以存在多个 return 语句，这些 return 语句表示在不同的条件下终止函数执行，并返回对应的数值。给函数添加返回值的语法，在前面定义函数时大家已经接触到了，如下。

【语法】

```
def 函数名([参数列表]):
    函数体
    [return 函数返回值]
```

前面的案例“计算 x 的 n 次方”中，已经明确的使用到了 return 来返回值，相信大家都已经熟悉了。下面我们再来看个例子。

示例：某公司根据员工在本公司的工龄决定其可享受的年假天数，如下表所示。

某公司年假天数表

工龄	年假天数
小于 5 年	1 天
5 年~10 年	5 天
10 年以上	7 天

定义函数 get_annual_leave()，向函数中传入员工工龄返回其可享有的年假天数并打印。

关键代码如下：

```
#定义函数
def get_annual_leave(seniority):
    if seniority < 5:
        return 1
    elif seniority < 10:
        return 5
    else:
        return 7
#调用函数
seniority = 7
days = get_annual_leave(seniority)
print("工龄是%d年的员工的年假天数是%d"%(seniority,days))
```

输出结果：工龄是 7 年的员工的年假天数是 5

2.yield 关键字

在 Python 里还有一个关键字 yield 也在函数中使用，它用于返回数值。但是 yield 与 return 相比具有不同的特点。

使用 yield 作为返回关键字的函数叫做生成器。生成器是一个可以被迭代的对象，在 Python 中能够使用 for..in…来操作的对象都是可迭代对象，例如之前学过的列表和字符串就是可迭代对象。使用 yield 返回的函数也可以使用 for…in…来操作，但是生成器每次只会被读取一次，也就是使用 for 循环迭代生成器的时候，每次执行到 yield 语句，生成器就会返回一个值，然后当 for 循环继续执行时，返回下一个值。

yield 返回数值有些像一个不终止函数执行的 return 语句。每次执行到它时都会返回一个数值，然后暂停函数（而不是终止），直到迭代器下一次从生成器中取值。

示例：使用 yield 关键字定义一个能够生成 0~3 数字序列的生成器，然后使用 for 循环输出这个数列。

关键代码如下：

```python
#定义两个函数
def generate_sequence_1():
    print("return 0")
    yield 0
    print("return 1")
    yield 1
    print("return 2")
    yield 2
    print("finish")
def generate_sequence_2():
    for i in range(3):
        print("return",i)
        yield i
    print("finish")
#调用函数
print("call generate_sequence_1:")
for i in generate_sequence_1():
    print("print",i)
print("call generate_sequence_2:")
for i in generate_sequence_1():
    print("print",i)
```

输出结果：

call generate_sequence_1:

return 0

print 0

return 1

print 1

return 2

print 2

finish

call generate_sequence_2:

return 0

print 0

return 1

print 1

return 2

print 2

finish

从这个示例可以看出，generate_sequence_1()和 generate_sequence_2()在执行效果上是相同的。只是 generate_sequence_1()中没有使用 for 循环来执行 yield 语句，generate_sequence_2()中使用 for 循环来执行 yield 语句。从输出的结果上可以看出“return”和“print”是交替出现的，并且是先出现“return”再出现与之对应的“print”。也就是当函数执行到 yield 语句后，函数返回了一个值，但是函数并没有被终止，而是暂停了，直到 for 循环继续迭代从生成器中取值时，函数才恢复运行。依此往复，直到所有生成器中的代码执行完毕。

实际上生成器也可以不通过 for 循环来迭代取出其返回的值，使用生成器对象的__next__()方法，就可以依次取出生成器的返回值。再来看这段代码，输出结果是什么？

```
def generate_sequence():
    for i in range(3):
        print("return", i)
        yield i
print("call generate_sequence:")
generate_sequence = generate_sequence()
print("print", generate_sequence.__next__())
print("print", generate_sequence.__next__())
print("print", generate_sequence.__next__())
```

输出结果：

call generate_sequence:

return 0

print 0

return 1

print 1

return 2

print 2

如上代码中使用__next__()方法将生成器中的值不断取出来，达到了迭代的效果。

示例：使用 yield 关键字创建一个生成 n 位的斐波那契数列的函数。

分析：

使用 yield 关键字来创建斐波那契数列，就不需要在函数中构造列表，再将列表作为整体返回；而是使用 yield 关键字依次返回数列中的数值就可以了。

关键代码如下：

```
def yield_gen_fibonacci(n):
    first = 1
    second = 1
    for pos in range(n):
        if pos == 0:
            yield first
        elif pos == 1:
            yield second
        else:
            first,second = second,first+second
            yield second
for item in yield_gen_fibonacci(10):
    print(item, end  = " ")
```

输出结果：

1 1 2 3 5 8 13 21 34 55