

第六章 函数

6.1 函数的概述

6.2 函数的参数和返回值

6.3 函数的调用

6.4 Python内置函数

6.5 实验

6.6 小结

6.7 习题

6.1.1 函数的定义

一个程序可以按不同的功能实现拆分成不同的模块，而函数就是能实现某一部分功能的代码块。

在Python中，定义一个函数要使用**def语句**，依次写出**函数名、括号、括号中的参数和冒号(:)**，然后在缩进块中编写**函数体**，函数的返回值用**return语句**返回。

注意：Python是靠缩进块来标明函数的作用域范围的，缩进块内是函数体，这和其它高级编程语言是有区别的，比如：
C/C++/Java/R语言大括号{ }内的是函数体。

我们以自定义一个求正方形面积的函数`area_of_square`为例，示例代码如下：

```
def area_of_square(x):
```

```
    s = x * x
```

```
    return s
```

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。

6.1.2 全局变量

在函数外面定义的变量称为**全局变量**。全局变量的作用域在整个代码段（文件、模块），在整个程序代码中都能被访问到。在函数内部可以去访问全局变量。如下所示代码：

```
def foodspice(per_price, number):  
    sum_price = per_price * number  
    print('全局变量PER_PRICE_1的值: ', PER_PRICE_1)  
    return sum_price  
  
PER_PRICE_1 = float(input('请输入单价: '))  
NUMBER_1 = float(input('请输入斤数: '))  
SUM_PRICE_1 = foodspice(PER_PRICE_1, NUMBER_1)  
print('蔬菜的价格是: ', SUM_PRICE_1)
```

代码运行结果如下：

请输入单价：21

请输入斤数：7.5

全局变量PER_PRICE_1: 21.0

蔬菜的价格是： 157.5

在上例中，我们在定义的函数foodsprice内部去访问在函数外面定义的全局变量PER_PRICE_1，能得到期望的输入结果21。

在函数内部可以去访问全局变量，但不要去修改全局变量，否则会得不到想要的结果。这是因为在函数内部试图去修改一个全局变量时，**系统会自动创建一个新的同名的局部变量去代替全局变量**，采用**屏蔽 (Shadowing)**的方式，当函数调用结束后函数的栈空间会被释放，数据也会随之释放。

如果要在函数内部去修改全局变量的值，并使之在整个程序生效，采用**关键字global**即可。

试图改变全局变量

```
def foods_price(per_price, number):  
    sum_price = per_price * number # 局部变量  
    PER_PRICE_1 = 9.5 # 试图改变全局变量,  
                        # 但实际上是一个新与全局变量同名的局部变量  
    print('PER_PRICE_1的值: ', PER_PRICE_1)  
    return sum_price
```

```
PER_PRICE_1 = float(input('请输入单价: ')) # 全局变量  
NUMBER_1 = float(input('请输入斤数: '))  
SUM_PRICE_1 = foods_price(PER_PRICE_1, NUMBER_1)  
print('蔬菜的价格是: ', SUM_PRICE_1)  
print('蔬菜的单价是: ', PER_PRICE_1)
```

```
请输入单价: 2.5  
请输入斤数: 3  
PER_PRICE_1的值: 9.5  
蔬菜的价格是: 7.5  
蔬菜的单价是: 2.5
```

用关键字global声明全局变量

```
def foods_price(per_price, number):  
    sum_price = per_price * number # 局部变量  
    global PER_PRICE_1 # 声明PER_PRICE_1是全局变量  
    PER_PRICE_1 = 9.5 # 改变全局变量, 全局有效  
    print('PER_PRICE_1的值: ', PER_PRICE_1)  
    return sum_price
```

```
PER_PRICE_1 = float(input('请输入单价: ')) # 全局变量  
NUMBER_1 = float(input('请输入斤数: '))  
SUM_PRICE_1 = foods_price(PER_PRICE_1, NUMBER_1)  
print('蔬菜的价格是: ', SUM_PRICE_1)  
print('蔬菜的单价是: ', PER_PRICE_1)
```

请输入单价: 2.5

请输入斤数: 3

PER_PRICE_1的值: 9.5

蔬菜的价格是: 7.5

蔬菜的单价是: 9.5

6.1.3 局部变量

在函数内部定义的参数和变量称为局部变量，超出了这个函数的作用域局部变量是无效的，它的作用域仅在函数内部。如下所示代码：

```
def foodspice(per_price, number):  
    sum_price = per_price * number  
  
    return sum_price  
  
PER_PRICE_1 = float(input('请输入单价: '))  
  
NUMBER_1 = float(input('请输入斤数: '))  
  
SUM_PRICE_1 = foodspice(PER_PRICE_1,NUMBER_1)  
  
print('蔬菜的价格是: ',SUM_PRICE_1)  
  
print('局部变量sum_price的值: ',sum_price)
```

代码运行结果如下：

请输入单价：12

请输入斤数：1.56

蔬菜的价格是：18.72

Traceback (most recent call last):

File "G:/6_1_3.py", line 9, in <module> print('局部变量sum_price的值：
' ,sum_price) NameError: name 'sum_price' is not defined

在上例中，我们试图在函数作用域外访问函数内的局部变量sum_price，程序运行到此处时报出了NameError的异常，提示变量sum_price没有定义。

6.1.4 定义匿名函数 (Anonymous Functions)

Lambda expressions (sometimes called **lambda forms**) are used to create **anonymous functions**.

```
lambda_expr ::= "lambda" [parameter_list] ":" expression
```

The expression **lambda parameters: expression** yields a function object. The unnamed object behaves like a function object defined with:

```
def <lambda>(parameters):  
    return expression
```

匿名函数在语法上
限于单个表达式

例如:

```
f1 = (lambda x, y: x + y) # 定义一个具有2个参数x, y的匿名函数
```

```
f2 = (lambda x: 0 if x < 'a' else 1) # 定义一个具有1个参数x的匿名函数
```

```
print(f1(2,5)) # 调用匿名函数
```

```
print(f2('b')) # 调用匿名函数
```

运行结果:

7

1

6.1.4 定义匿名函数 (Anonymous Functions)

有些函数在执行过程中，需要调用一个指定函数来完成特定任务，这个指定函数则是通过参数传入的，例如sorted、map、filter、reduce函数。

匿名函数更常见的一个用法就是**它作为参数**传递一个这样的函数调用。

对元组列表中的元素按元组中的第2个元素的降序排列

```
b = [('Japan', 35), ('USA', 50), ('China', 80), ('UK', 20)]
```

```
result = sorted(b, key=lambda x: x[1], reverse=True)
```

```
print(result)
```

参数key需要传入一个函数对象

运行结果：

```
[('China', 80), ('USA', 50), ('Japan', 35), ('UK', 20)]
```

第六章 函数

6.1 函数的概述

6.2 函数的参数和返回值

6.3 函数的调用

6.4 Python内置函数

6.5 实验

6.6 小结

6.7 习题

函数的参数就是使得函数个性化的一个实例。代码如下所示：

```
>>> def MyFirstFunction(name_city):  
        print('我喜欢的城市:' + name_city)
```

运行结果如下：

```
>>> MyFirstFunction('南京')
```

我喜欢的城市:南京

```
>>> MyFirstFunction('上海')
```

我喜欢的城市:上海

在上例中，对函数MyFirstFunction的形参name_city赋予不同的实参“南京”、“上海”后，函数就输出不同的结果。

函数有了参数之后，函数的输出结果变得可变了，如果需要多个参数，用逗号 “,” (英文状态下输入) 隔开即可。

在Python中对函数参数的数量没有限制，但是定义函数参数的个数不宜太多，一般2~3个即可。在定义函数时，一般要把函数参数的意义注释清楚，便于阅读程序。

那什么是**形参和实参**呢？

- 定义函数时声明的参数叫**形式参数**，简称**形参(Parameters)**，即函数定义时小括号 “()” 内的参数，它定义了函数可以接受的名称及参数类型。
- 调用函数时传给函数的参数叫**实际参数**，简称**实参(Arguments)**，它们是调用函数时传给函数的实际值。

例如下面的函数定义：

```
def MyFirstFunction(name_city):  
    print('我喜欢的城市:' + name_city)
```

name_city 是函数的形参(parameters)。

当我们调用MyFirstFunction时，比如：

```
MyFirstFunction('南京')
```

传给函数的值'南京'就是实参(Arguments)。

又如下面的函数定义：

```
def func(foo, bar=True, **kwargs):  
    print(foo)  
    print(bar)  
    for arg in kwargs: # kwargs是一个字典  
        print(arg, kwargs[arg])
```

foo, *bar*, *kwargs* 是函数的形参(parameters)。

当我们调用func时，比如：

```
func(20, bar=False, abc='xyz', efg=123)
```

传给函数的值20, False, 'xyz'和123就是实参(Arguments)。

运行结果：

20

False

abc xyz

efg 123

6.2.1 函数参数类别

在Python中，可将函数形参分为4类：

(1) 位置参数 (Positional Parameters)

又分为必选 (Required) 参数和默认 (Default) 参数。传递实参时，可以按形参顺序直接传递，即传递**位置实参** (Positional Arguments) ，也可以按形参名传递，即传递**关键字实参** (Keyword Arguments) 。

(2) 可变参数

定义时，参数名前**添加1个星号 "*"** ，形如 ***arg**，实参数量可变，但**只能传递位置实参**。

(3 命名关键字参数 (Keyword-only Parameters)

定义时，这些参数与其他参数用星号*分割，**只能传递关键字实参**，且**实参的名字和数量是固定的**，可以有默认值。

(3) 关键字参数 (Keyword Parameter)

定义时，在形参前定义**添加2个星号 "*"** ，形如****kwargs**，**只能传递关键字实参**，但实参的名字和数量是可变的。。

6.2.2 位置参数 (Positional Parameters)

(1) 必选参数 (Required Parameters)

形参列表中，只有参数名、没有赋值、也不带星号的参数就是必选参数。

例如：

```
def Sub(x, y):  
    return x-y
```

函数调用时，必须传入实参，可以是位置实参，也可以是关键字实参，实参数量与形参相同。

调用函数时，传入参数值按照从左到右的位置顺序依次赋给形参，这样的实参称为**位置实参**(Positional Arguments)。例如：

```
>>> Sub(100,30)
```

```
70
```

6.2.2 位置参数 (Positional Parameters)

上例中，Sub(x, y)函数有两个形参x和y，**函数调用Sub(100,30)中传入的两个值100和30是位置参数**，它们按照函数定时的形参位置顺序依次赋给x和y，得到的两数相减的结果是70。

如果交换了参数的位置，就会得到不同的结果，如上例中交换参数后的运行结果如下：

```
>>> Sub(30,100)
```

```
-70
```

从上面的运行结果可以看出，交换了参数顺序后的运行结果是-70，而不是我们期望的结果70。

- **解包参数列表**：元组或列表可以使用 * 运算符来提供位置参数。

Sub(100,30) \Leftrightarrow Sub(*(100,30)) \Leftrightarrow Sub(*[100,30])

6.2.2 位置参数 (Positional Parameters)

在函数调用时，也可以通过形参名（关键字）指定需要赋值的形参，形式是kw=value，这样的实参称为**关键字实参**(Keyword Arguments)。

关键字参数是通过关键字来确认参数的，所以可以不用按照函数定义时的顺序传递参数。

通常在调用一个函数的时候，如果参数有多个，常常容易混淆参数的顺序。在Python中引入关键字参数就可解决这个问题。

如下所示代码：

```
>>> def subtraction(x, y):  
    return (x - y)
```

6.2.2 位置参数 (Positional Parameters)

运行结果如下：

```
>>> subtraction(34,11)
```

```
23
```

```
>>> subtraction(11,34)
```

```
-23
```

```
>>> subtraction(y=11, x=34)
```

```
23
```

解包参数列表：

字典可以使用 `**` 运算符来提供关键字实参

```
subtraction(**{'y': 11, 'x': 34})
```



```
subtraction(x=34, y=11)
```



在上例中，调用函数subtraction时：

- 第1次调用函数subtraction时，给2个参数顺序赋值34、11时得到的结果是23；
- 第2次调用时，交换了2个赋值参数的顺序，得到的结果是-23；
- 第3次调用该函数时，**传递了关键字实参**，改变参数顺序，不影响计算结果。
- **解包参数列表：**字典可以使用 `**` 运算符来提供**关键字实参**。

6.2.2 位置参数 (Positional Parameters)

位置实参和关键字实参可以混合使用。

但是，所有位置实参必须在左边，关键字实参在右边，即关键字实参后面必须都是关键字实参。例如：

```
>>> subtraction(34,y=11)
```

```
23
```

注意，下面的调用都是无效的：

- `subtraction(x=34,11)` # 关键字实参后面出现了位置实参
- `subtraction(34, x=11)` # 给同样的参数x传了两个值
- `subtraction(a=34, y=11)` # 函数中不存在形参a

6.2.3 默认值参数 (Default Parameter)

在定义函数时可以给参数赋一个初值，这样的参数称为**默认值参数**。没有默认值的参数是必选参数，默认值参数则是可选的。

应用默认值参数的意义在于：

- (1) 常用的参数值不必显式指定，函数就会自动去找它的初值，使用默认值来代替，从而简化调用；
- (2) 如果某个参数具有默认值，则当函数调用忘记了给该参数赋值时，函数调用不会出现错误。

默认参数值在函数定义时便已确定。

Default parameter values are evaluated from left to right when the function definition is executed.

6.2.3 默认值参数 (Default Parameter)

```
def subtraction(x=99,y=45):  
    return (x - y)
```

在定义函数时分别给2个参数x, y赋了初值99和45。

运行结果如下:

```
>>> subtraction()
```

```
54
```

```
>>> subtraction(46)
```

```
1
```

```
>>> subtraction(46,12)
```

```
34
```

3次调用:

- 第1次调用时没有赋值, 程序就引用了2个参数的默认值99, 45, 返回的结果是54;
- 第2次调用时, 给第1个参数赋值为46, 程序就引用了第2个参数的默认值45, 返回的结果是1;
- 第3次调用时, 给2个参数分别赋值为46和12, 程序就没有引用函数定义的默认值, 返回的结果是34。

Default parameter values are evaluated from left to right when the function definition is executed.

6.2.3 默认值参数 (Default Parameter)

```
def func(a, b=5, c=10):  
    print('a is', a, 'and b is', b, 'and c is', c)
```

func(3, 7) # c采用默认值10	→	a is 3 and b is 7 and c is 10
func(25, c=24) # b采用默认值5	→	a is 25 and b is 5 and c is 24
func(c=50, a=100) # b采用默认值5	→	a is 100 and b is 5 and c is 50
func(3) # b采用默认值5, c采用默认值10	→	a is 3 and b is 5 and c is 10

func(c=50, b=100) # 错误, 参数a的值未指定

func(c=50) # 错误, 参数a的值未指定

func() # 错误, 参数a的值未指定

```
def func(a=4, b, c=10): # 错误  
    print('a is', a, 'and b is', b, 'and c is', c)
```

```
def func(a=4, b=5, c): # 错误  
    print('a is', a, 'and b is', b, 'and c is', c)
```

Default parameter values are evaluated from left to right when the function definition is executed.

所有默认值参数必须定义在列表右端 (即默认值参数后不能跟无默认值参数)

Python函数默认参数的小陷阱

```
def foo(a1, args=[]):  
    print("args before = %s" % (args))  
    args.insert(0, 10)  
    args.insert(0, 99)  
    print("args = %s " % (args))
```

```
foo('a')  
foo('b')
```

运行结果:

```
args before = []  
args = [99, 10]  
args before = [99, 10]  
args = [99, 10, 99, 10]
```

照通常的理解，第二次调用的args应该为默认值[]，但为什么会变成上一次的结果呢？

Python函数默认参数的一个小陷阱

Default parameter values are evaluated from left to right when the function definition is executed. This means that the expression is evaluated once, when the function is defined, and that the same “pre-computed” value is used for each call. This is especially important to understand when a default parameter is a **mutable object**, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default value is in effect modified. This is generally not what was intended. A way around this is to use **None** as the default, and explicitly test for it in the body of the function, e.g.:

```
def whats_on_the_telly(penguin=None):  
    if penguin is None:  
        penguin = []  
    penguin.append("property of the zoo")  
    return penguin
```

至此，原因已经清楚了：默认参数的默认值在函数定义时就确定了，而参数默认值是可变对象（例如list, dict, set），函数体内修改了原来的默认值，而python会将修改后的值一直保留，并作为下次函数调用时的参数默认值。

Python函数默认参数的一个小陷阱

修改程序如下：

```
def foo(a1, args=[]):  
    print("args before")  
    args.insert(0, 99)  
    args.insert(0, 10)  
    print("args = %s" % (args))
```

默认实参值可变
替换可变默认实参 Alt+Shift+Enter

```
foo('a')  
foo('b')
```

```
def foo(a1, args=None):  
    if args is None:  
        args = []  
    print("args before = %s" % (args))  
    args.insert(0, 10)  
    args.insert(0, 99)  
    print("args = %s" % (args))
```

```
foo('a')  
foo('b')
```

通常不建议默认值为可变对象，而是不可变的整数、浮点数、字符串等。

运行结果：

```
args before = []  
args = [99, 10]  
args before = []  
args = [99, 10]
```

6.2.4 可变参数

在定义函数参数时，如果不能确定究竟需要传多少个实参，则只要在参数前面加上一个星号 “*”，形如 ***args**，这样的参数称为**可变参数**。

例如，系统函数 print 中的 value 参数就是可变参数。

给可变参数传递实参时，只能传递**位置实参**（Positional Arguments）。

当声明一个形如 ***args** 的星号形参时，且从该参数直到结束的所有位置实参都将被收集成一个名为 args 的**元组**。如果没有传递位置实参给 args，则 args 为空元组，即 args=()。

这个收集的过程称为参数打包（Packing），可变参数又称**收集参数**。

```
def print(*values: object,
        sep: str | None = ...,
        end: str | None = ...,
        file: SupportsWrite[str] | None = ...,
        flush: bool = ...) -> None
```

6.2.4 可变参数

例如：

```
def var_par(*args):  
    print('第三个参数是：', args[2])  
    print('可变参数的长度是：', len(args))
```

```
var_par('xx科技股份有限公司', 345, 9, 9.8, 2.37, 'Python')
```

`args=('xx科技股份有限公司', 345, 9, 9.8, 2.37, 'Python')`

输出：

第三个参数是： 9

可变参数的长度是： 6

```
var_par('xx科技股份有限公司', 345, 'Python')
```

`args=('xx科技股份有限公司', 345, 'Python')`

输出：

第三个参数是： Python

可变参数的长度是： 3

```
var_par(*('xx科技股份有限公司', 345, 'Python'))
```

```
var_par(*['xx科技股份有限公司', 345, 'Python'])
```

解包参数列表：元组或列表可以使用 * 运算符来提供位置实参

6.2.4 可变参数

例如：

```
def var_par(*args):  
    print('第三个参数是：', args[2])  
    print('可变参数的长度是：', len(args))  
  
var_par('xx科技股份', 345, 9, 9.8, 2.37, 'Python')
```

`args=('xx科技股份', 345, 9, 9.8, 2.37, 'Python')`

调试不难发现，这些实参确实作为一个元组传递给了可变参数param

```
def var_par(*args):  
    args: ('xx科技股份', 345, 9, 9.8, 2.37, 'Python')  
    print('第三个参数是：', args[2])  
    print('可变参数的长度是：', len(args))
```

```
var_par('xx科技股份', 345, 9, 9.8, 2.37, 'Python')
```


6.2.4 可变参数

```
def total(*args):
```

```
    """可变参数"""
```

```
    s = 0
```

```
    for x in args:
```

```
        s += x
```

```
    return s
```

```
# 未传入参数
```

```
print(total()) # 输出: 0
```

```
# 直接传入若干个位置参数
```

```
print(total(1, 4)) # 输出: 5
```

```
# 将若干位置参数装入列表, 然后向函数中传入该列表, 并在列表前加*
```

```
a=[1, 4, 7]
```

```
print(total(*a)) # 输出: 12
```

```
# 将若干位置参数装入元组, 然后向函数中传入该元组, 并在元组前加*
```

```
print(total(*(1, 4, 7, 6))) # 输出: 18
```

注意区别: 如果total形参改为 **def total(args)**

此时args不是可变参数, 调用时只能传入一个实参, 且这个实参必须是可迭代对象, 例如

```
total([1, 4, 7])
```

6.2.4 可变参数

如果可变参数后面还有其它参数，在参数传递时**可变参数后的参数必须传递关键字实参**，或者在定义函数参数时要给它赋默认值，否则会出错。

如下所示代码：

```
def var_par(*args, str1):  
    print('第三个参数是：', args[2])  
    print('可变参数的长度是：', len(args))
```

6.2.4 可变参数

运行结果如下：

```
>>> var_par('xx科技股份',345,9,9.8,2.37,'Python','函数')
```

```
TypeError: var_par() missing 1 required keyword-only argument: 'str1'
```

```
>>> var_par('xx科技股份',345,9,9.8,2.37,'Python',str1='函数')
```

第三个参数是： 9

可变参数的长度是： 6

在上例中，在定义函数var_par()时分别定义了1个可变参数args，1个必选参数str1：

- 在第1次调用该函数时，由于没有将可变参数后面的必选参数没有传递关键字实参，解释器把所有实参都收集为元组传递给args，必选参数未传值，导致报错。
- 在第2次调用该函数时，将可变参数后的必选参数传递了关键字实参(str1='函数')，程序运行正常。

6.2.4 可变参数

可变参数后的参数赋予默认值，如下所示代码：

```
def var_par(*param,str1='可变函数'):  
    print('可变参数后的参数是：',str1);  
    print('可变参数的长度是：',len(param));
```

运行结果如下：

```
>>> var_par('xx科技股份',345,9,9.8,2.37,'Python')
```

可变参数后的参数是： 可变函数

可变参数的长度是： 6

在上例中，在定义函数var_par()时分别定义了1个可变参数param，1个默认参数str1，在调用该函数时，没有给默认参数传值，程序运行仍然正常，程序引用了函数的参数默认值。

6.2.5 命名关键字参数 (Keyword-only Parameters)

只能通过关键字实参传值的固定形参就是**命名关键字参数**。定义时，参数列表中添加星号*项，其后的参数（带**的参数除外）就是命名关键字参数。

例如，json 模块中的dump函数就定义了skipkeys、ensure_ascii 等关键字参数：

```
def dump(obj: Any,
         fp: IO[str],
         *,
         skipkeys: bool = ...,
         ensure_ascii: bool = ...,
         check_circular: bool = ...,
         allow_nan: bool = ...,
         cls: Type[JSONEncoder] | None = ...,
         indent: None | int | str = ...,
         separators: Tuple[str, str] | None = ...,
         default: (Any) -> Any | None = ...,
         sort_keys: bool = ...,
         **kwds: Any) -> None
```

6.2.5 命名关键字参数 (Keyword-only Parameters)

例如，下面代码中的 city 和 job 也是命名关键字参数：

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

```
    print(name, age, city, job)
```

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

```
# TypeError: person() missing 2 required keyword-only arguments: 'city' and 'job'
```

```
person('Jack', 24)
```

```
# TypeError: person() missing 1 required keyword-only argument: 'job'
```

```
person('Jack', 24, city='Beijing')
```

```
# TypeError: person() got an unexpected keyword argument 'zipcode'
```

```
person('Jack', 24, city='Beijing', job='Engineer', zipcode='12345')
```

6.2.5 命名关键字参数 (Keyword-only Parameters)

命名关键字参数可以设置默认值，以便未传递实参时引用它们。

例如：

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

```
person('Jack', 24, city='Shanghai', job='Engineer') # Jack 24 Shanghai  
Engineer
```

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

6.2.6 关键字参数 (Keyword Parameters)

类似于可变参数，如果要传入的关键字实参的数量是可变的，则可以定义关键字参数，形如 ****kwargs**。例如：

```
def person(name, age, **kwargs):  
    print('name:', name, 'age:', age, 'other:', kwargs)
```

又如，json 模块中的 dump 函数的 kwds 参数：

```
def dump(obj: Any, fp: IO[str], *, skipkeys: bool ... **kwds: Any) -> None
```

当声明一个形如 ****kwargs** 的双星号形参时，从此处开始直至结束的所有**关键字实参** (Keyword Arguments) 都将被收集成一个名为 kwargs 的**字典**。如果未传递实参给 kwargs，则 kwargs 为空字典，即 kwargs={}。

与可变参数的实参收集过程类似，关键字实参的收集过程也称为打包。

6.2.6 关键字参数 (Keyword Parameters)

例如：

```
def var_par(**kwargs):  
    for x in kwargs :  
        print(f'key={x}, value={kwargs[x]}')
```

传入1个关键字实参

```
var_par(name='张三')
```



输出：

key=name, value=张三

kwargs={'name': '张三'}

传入3个关键字实参

```
var_par(name='王五', sex='女', age=25)
```



输出：

key=name, value=王五

key=sex, value=女

key=age, value=25



kwargs={'name': '王五', 'sex': '女', 'age': 25}

```
var_par(**{'name': '王五', 'sex': '女', 'age': 25})
```

解包参数列表：字典可以使用 ** 运算符来提供位置关键字实参

6.2.6 关键字参数 (Keyword Parameters)

```
def var_par(**kwargs):    kwargs: {'name': '李四', 'sex': '男'}
    for x in kwargs:
        print(f'key={x}, value={kwargs[x]}')

# 传入1个关键字参数
var_par(name='张三')
# 传入2个关键字参数
var_par(name='李四', sex='男')
```

调试不难发现，这些参数确实作为一个字典传递给了关键字参数 kwargs

注意区别：如果函数var_par的形参改为 `def var_par(kwargs)` 则此时 `kwargs` 不是可变参数，调用时只能传入一个实参，且这个实参必须是字典，例如 `var_par({'name': '王五', 'sex': '女', 'age': 25})`

6.2.6 关键字参数 (Keyword Parameters)

关键字参数有什么用?

它可以扩展函数的功能。

比如，在person函数里，保证能接收到name和age这两个参数，但是，如果调用者愿意提供更多的参数，从kwargs参数也能收到。

试想，你正在做一个用户注册的功能，除了用户名和年龄是必填项外，其他都是可选项，则利用关键字参数来定义这个函数就能满足注册的需求。

```
def person(name, age, **kwargs):  
    print('name:', name, 'age:', age, 'other:', kwargs)
```

person('Jack', 24)

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

person('Jack', 24, city='Beijing', job='Engineer', zipcode='12345')

name: Jack age: 24 other: {}

name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}

name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer', 'zipcode': '12345'}

6.2.6 关键字参数 (Keyword Parameters)

关键字参数与可变参数的异同：

(1) 不同之处：

- 当声明一个形如 `*args` 的星号形参时，从此处开始直到结束的所有**位置实参**都将被收集成一个名为 `args` 的**元组**。
- 当声明一个形如 `**kwargs` 的双星号形参时，从此处开始直至结束的所有**关键字实参**都将被收集成一个名为 `kwargs` 的**字典**。

(2) 相同之处：

- 实参的个数都是可变的，如果未传递实参，则 `args=()`, `kwargs={}`。
- 声明时，这样的参数都只能出现一次。

6.2.6 关键字参数 (Keyword Parameters)

各程序代码的输出结果是什么？

```
def func(*param):  
    print(type(param))  
    print('|'.join(param))
```

```
func('a', 'b', 'c', 'd')
```

```
def func2(*param, joiner='-'):  
    print(type(param))  
    print(joiner.join(param))
```

```
func2('a', 'b', 'c', 'd', joiner='&')
```

```
def func3(a, **kwargs):  
    print(a)  
    print(type(kwargs))  
    for k, v in kwargs.items():  
        print('%s : %s' % (k, v))
```

```
func3(a=1, b=2, c=3, d=4, e='a')
```

6.2.7 参数组合

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可以组合使用，除了可变参数无法和命名关键字参数混合。**但是，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。**

```
def func(foo, *args, **kwargs):
```

```
    print(foo) # 读取位置参数
```

```
    for arg in p args: # 读取可变数量的位置参数
        print(arg)
```

```
    for arg in kwargs: # 读取可变数量的关键字参数
        print(arg, kwargs[arg])
```

```
func(20, 'Tom', True, abc='xyz', efg=123)
```

foo=20 args=('Tom': True) kwargs={'abc': 'xyz', 'efg': '123'}

输出：

20

Tom

True

abc xyz

efg 123

函数调用时，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

6.2.7 参数组合

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

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

```
>>> 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=66, y=88, z=99)  
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 66, 'y': 88, 'z': 99}  
>>> f2(1, 2, d=99, ext=None)  
a = 1 b = 2 c = 0 d = 99 kw = {'ext': None}
```

6.2.7 参数组合

```
def f1(**kwargs, *args):
```

```
    pass
```

* 形参在 ** 形参后面

Parameter **args** of `new.ch1_ch6.demo_06_解包参数列表.f1`
args: Tuple[Any, ...]

```
def f2(**kwargs, **args):
```

```
    pass
```

```
def f3(*args1, *args2):
```

```
    pass
```

```
def f4(*args, **kwargs):
```

```
    pass
```

注意:

- 可变的位置参数必须位于可变的关键字参数之前。
- 两种可变参数在一个函数定义中都至多各出现一次。

6.2.8 解包 (unpacking) 参数列表

解包参数是可变参数或关键字参数的实参打包的相反过程。

(1) 如果要调用的函数的参数值已经在list或tuple里面了，可以通过解包list或tuple向函数传递位置实参。此时使用*运算符解包，形如*args。

例如，内置的range()函数可以输入两个参数：start和stop，如果它们在一个list或tuple里面，可以通过*操作符解包来传值：

```
print(list(range(3, 6))) # 输出: [3, 4, 5]
args = [3, 6]
print(list(range(*args))) # 输出: [3, 4, 5]
```


total(a[0],a[1], a[2], a[3])
也是可以的，但过于繁琐

又如：

```
def sub(x, y):
    return x - y

print(sub (1, 4)) # 输出: -3
print(sub (*[100, 1])) # 输出: 99
```

```
def total(*param):
    s = 0
    for x in param:
        s += x
    return s
a=(1, 4, 7, 6)
print(total(*a)) # 输出: 18
```



6.2.8 解包 (unpacking) 参数列表

(2) 如果要调用的函数的参数值已经在dict里面了，可以通过解包dict向函数传递关键字实参，此时使用**运算符解包，形如****kwargs**。

例如：

```
def say_hi(name, greeting='Hi', more=''):
    print(greeting, name, more)

say_hi(name='Tom', greeting='Hello', more='good day')

d = {'name': 'Tom', 'greeting': 'Hello', 'more': 'good day'}
say_hi(**d)
```

6.2.9 参数的传递方式：值传递还是引用传递？

python不允许程序员为参数选择采用传值还是传引用，而是统一采用“**传对象引用**”的方式。

- Python对象分为可变对象(list, dict, set等)和不可变对象(number, string, tuple等)，当传递的参数是**可变对象的引用**时，由于可变对象的值可以修改，因此可以通过修改参数值而修改原对象，这类似于C语言中的引用传递；
- 当传递的参数是**不可变对象的引用**时，虽然传递的是引用，参数变量和原变量都指向同一内存地址，但是不可变对象无法修改，所以参数的重新赋值不会影响原对象，而是生成并引用一个深拷贝的对象，这类似于C语言中的值传递。

x是不可变对象

```
>>> x=1
>>> id(x)
140733810611872
>>> x=x+1
>>> id(x)
140733810611904
```

y是可变对象

```
>>> y=[1,3]
>>> id(y)
2401953922112
>>> y.append(5)
>>> y
[1, 3, 5]
>>> id(y)
2401953922112
```

6.2.9 参数的传递方式：值传递还是引用传递？

```
def changeInt(y: int):  
    print('变量y的地址(修改前):', id(y))  
    y = y + 1  
    print('变量y的地址(修改后):', id(y))  
    print("changeInt: y =", y)
```

```
x = 2  
print('    变量x的地址:', id(x))  
changeInt(x)  
print("x =", x)
```

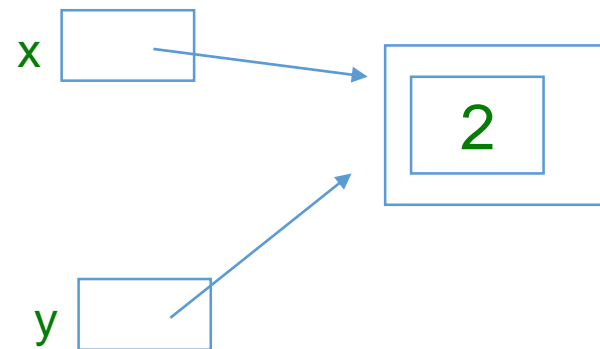
变量x的地址: 140733810611904

变量y的地址(修改前): 140733810611904

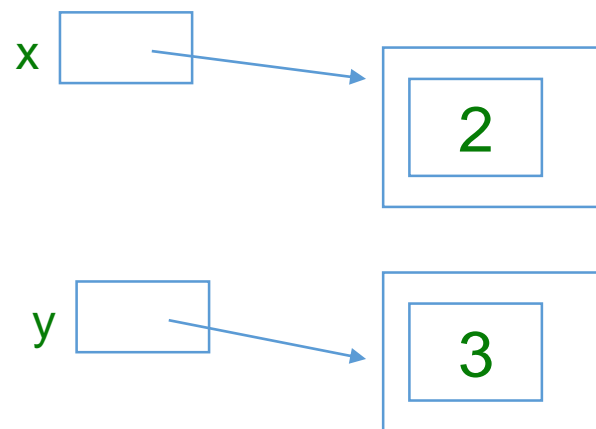
变量y的地址(修改后): 140733810611936

changeInt: y = 3

x = 2



y = y + 1 执行前



y = y + 1 执行后

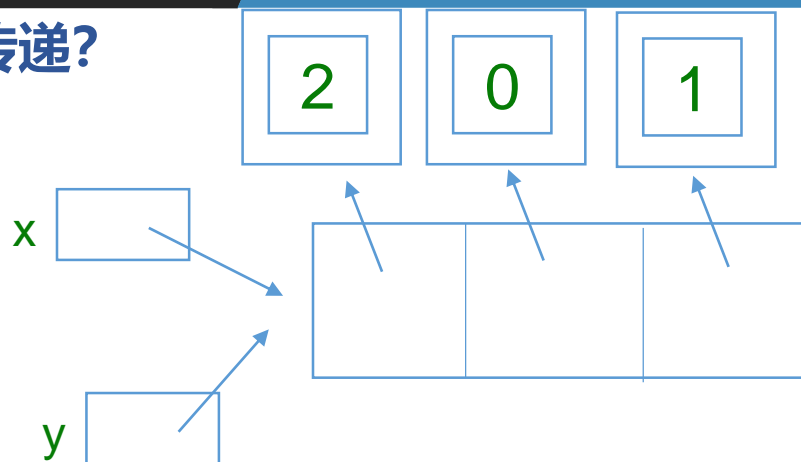
6.2 函数的参数和返回值

6.2.9 参数的传递方式：值传递还是引用传递？

```
def changeList(y: list):  
    print('变量y的地址(修改前):', id(y))  
    y.append(5)  
    print('变量y的地址(修改后):', id(y))  
    print("changeList: y =", y)
```

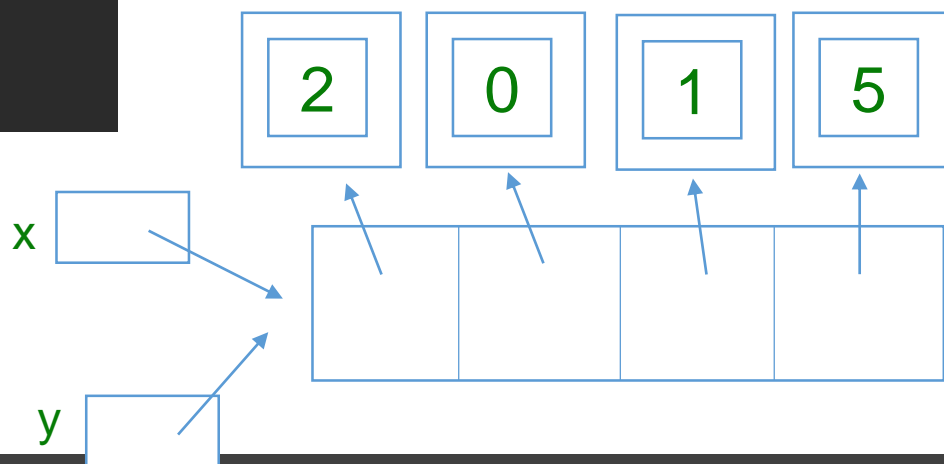
```
x = [2, 0, 1]  
print('    变量x的地址:', id(x))  
changeList(x)  
print("x =", x)
```

变量x的地址: 2344345138112
变量y的地址(修改前): 2344345138112
变量y的地址(修改后): 2344345138112
changeList: y = [2, 0, 1, 5]
x = [2, 0, 1, 5]



y.append(5)执行前

y.append(5)执行后



6.2.10 参数使用规则归纳

Python的函数具有非常灵活的参数形态，既可以实现简单的调用，又可以传入非常复杂的参数。

(1) 默认参数一定要用不可变对象，如果是可变对象，程序运行时会有逻辑错误！

(2) 要注意定义可变参数和关键字参数的语法

- `*args`是可变参数，`args`接收的是一个`tuple`；
- `**kwargs`是关键字参数，`kwargs`接收的是一个`dict`。
- 使用`*args`和`**kwargs`是Python的习惯写法，当然也可以用其他参数名，但最好使用习惯用法。

6.2.10 参数使用规则归纳

(3) 调用函数时如何传入可变参数和关键字参数的语法

- 可变参数既可以直接传入：`func(1, 2, 3)`，又可以先组装`list`或`tuple`，再通过`*args`解包传入：`func(*(1, 2, 3))`；
- 关键字参数既可以直接传入：`func(a=1, b=2)`，又可以先组装`dict`，再通过`**kwargs`解包传入：`func(**{'a': 1, 'b': 2})`。

(4) 命名的关键字参数是为了限制调用者可以传入的参数名，同时可以提供默认值。定义命名的关键字参数不要忘了写分隔符`*`，否则定义的将是位置参数。

(5) 使用多种参数组合定义函数时，**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。**

6.2.11 函数的返回值

有些时候，需要函数返回一些数据来报告函数实现的结果。在函数中用**关键字return**返回指定的值。

如下所示代码：

```
>>> def subtraction(x,y):  
        return (x - y)
```

运行结果如下：

```
>>> print(subtraction(65,23))
```

```
42
```

```
>>> subtraction(34,11)
```

```
23
```


6.2.11 函数的返回值

函数中如果没有用关键字return指定返回值，则返回一个**None对象**。如下所示代码：

```
>>> def test_return():  
        print('Hello First1')
```

运行结果如下：

```
>>> tempt = test_return()  
Hello First1  
>>> tempt  
>>> print(tempt)  
None  
>>> type(tempt)  
<class 'NoneType'>
```

6.2.11 函数的返回值

Python是动态的确定变量类型，变量没有类型，只有名字。Python可以**返回多个值**，它们类型的可以不同。

如下所示代码：

```
>>> def back_test():  
        return ['xx科技',3.67,567]
```

运行结果如下：

```
>>> back_test()  
['xx科技', 3.67, 567]
```

在上例中，Python返回一个**列表数据**，其中有多不同个不同类型的值。

6.2.11 函数的返回值

如下所示代码：

```
>>> def back_test():  
        return 'xx科技',3.67,567
```

运行结果如下：

```
>>> back_test()  
('xx科技', 3.67, 567)
```

在上例中，Python返回多个值是**元组数据**。

6.2.12 函数中使用pass语句

Python pass 是空语句，是为了保持程序结构的完整性。

pass 不做任何事情，一般用做占位语句。

Python 语言 pass 语句语法格式如下：

pass

例如：

输出 Python 的每个字母

for letter **in** 'Python':

if letter == 'h':

pass

 print('这是 pass 块')

 print('当前字母:', letter)

print("Good bye!")

输出：

当前字母 : P

当前字母 : y

当前字母 : t

这是 pass 块

当前字母 : h

当前字母 : o

6.2.12 函数中使用pass语句

pass 一般用于占位置。

在 Python 中有时会看到一个 def 函数:

```
def sample(n_samples):  
    pass
```

该处的 pass 便是占据一个位置，因为如果定义一个空函数程序会报错。当你没有想好函数的内容时，可以用 pass 填充，使程序可以正常运行。这个函数什么也不做，返回None对象。

6.2.13 生成器函数与表达式生成器

列表推导式优点很多，比如运行速度快、编写简单，但是有一点我们不要忘了，它是一次性生成整个列表。如果整个列表非常大，这对内存也同样会造成很大压力，想要实现内存的节约，可以将列表推导式转换为生成器表达式。

python中有两种语言结构可以实现这种思路

- 一个是**生成器函数**，外表看上去像是一个函数，但是没有用return语句一次性的返回整个结果对象 列表，取而代之的是使用**yield语句**一次返回一个结果。
- 另一个是**生成器表达式**，类似于列推导式，但是方括号换成了圆括号，它返回按需产生的一个结果对象，而不是构建一个结果列表。

6.2.13 生成器函数与表达式生成器

```
from collections.abc import Iterator
```

```
def gen_squares(m): # 生成器函数
    for x in range(m):
        y = x ** 2
        print(f'生成第{x}个值: ', end='')
        yield y
    print('生成结束')
```

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

```
    G = gen_squares(3) # 调用生成器函数，得到一个生成器
```

```
    print(isinstance(G, Iterator))
```

```
    print(next(G)) # 调用next函数，从生成器产生一个值
```

```
    print(next(G))
```

```
    print(next(G))
```

```
    # print(next(G)) # 异常：迭代已经结束
```

```
    print('-' * 20)
```

```
    print(list(gen_squares(2))) # 自动迭代并将所有生成的值转换为列表
```

True

生成第0个值: 0

生成第1个值: 1

生成第2个值: 4

生成第0个值: 生成第1个值: 生成结束

[0, 1]

6.2.13 生成器函数与表达式生成器

```
from collections.abc import Iterator
```

```
G = (x**2 for x in range(5)) # 表达式生成器
```

```
print(isinstance(G, Iterator))
```

```
print(next(G)) # 调用next函数，从生成器产生一个值
```

```
print(next(G))
```

```
print(next(G))
```

```
print(next(G))
```

```
print(next(G))
```

```
print('-' * 20)
```

```
print(list(G))
```

True

0

1

4

9

16

[]

List(G)将G从中上次迭代之后的下一位置继续迭代，直到结束

6.2.13 生成器函数与表达式生成器

```
from collections.abc import Iterator
```

```
G = (x**2 for x in range(5)) # 表达式生成器
```

```
print(isinstance(G, Iterator))
```

```
print(next(G)) # 调用next函数, 从生成器产生一个值
```

```
print(next(G))
```

```
print(next(G))
```

```
print('-' * 20)
```

```
print(list(G))
```

True

0

1

4

[9, 16]

List(G)将G从中上次迭代之后的下一位置继续迭代, 直到结束

与生成器函数区别:

生成器函数可以被多次调用以产生多个生成器对象,

而表达式生成器是一次性的,迭代完之后就不能再次使用了, 除非再次定义它

6.2.13 生成器函数与表达式生成器

```
G = (x**2 for x in range(5)) # 表达式生成器
for _ in range(3):
    print(list(G))
```



```
[0, 1, 4, 9, 16]
[]
[]
```

```
for _ in range(3):
    G = (x**2 for x in range(5)) # 表达式生成器
    print(list(G))
```



```
[0, 1, 4, 9, 16]
[0, 1, 4, 9, 16]
[0, 1, 4, 9, 16]
```

```
def gen_squares(): # 生成器函数
    for x in range(5):
        yield x ** 2
```

```
g = gen_squares() # 生成器对象
for _ in range(3):
    print(list(g))
```



```
[0, 1, 4, 9, 16]
[]
[]
```

```
for _ in range(3):
    g = gen_squares() # 生成器对象
    print(list(g))
```



```
[0, 1, 4, 9, 16]
[0, 1, 4, 9, 16]
[0, 1, 4, 9, 16]
```

6.2.14 Python中的参数注解和类型注解

众所周知，Python 是动态类型语言，运行时不需要指定变量类型。2015年9月创始人 Guido van Rossum 在 Python 3.5 引入了一个类型系统，允许开发者指定变量类型—**类型提示 (Type Hints)**。它的主要作用是**方便开发，供IDE 和各种开发工具使用，给使用者提供一个很好的说明，对代码运行不产生影响，运行时会过滤类型信息。**

使用类型提示意味着IDE可以拥有更准确、更智能的建议引擎。

- 函数有了类型提示，则在调用函数时IDE就提示需要传递哪些参数类型及其说明。
- 变量有了类型提示，则在使用对象时，IDE会提示该对象有哪些方法及属性可用。
- 如果用户尝试调用不存在的内容或传递不正确类型的参数，IDE可以立即警告它。

6.2.14 Python中的参数注解和类型注解

(1) 变量类型注解

基本语法：

变量名: 数据类型 = 初值

name: str = 'Tom' # 变量类型注解, 为str类型

age: int = 18 # 变量类型注解, 为int类型

```
13 """
14
15 a: list = [1, 2] # 变量类型注解, 为list类型
16 b: list = (1, 2) # 给出警告: 应为类型 'list', 但实际为 'Tuple[int, int]'
```

应为类型 'list', 但实际为 'Tuple[int, int]'

鼠标移到这里

6.2.14 Python中的参数注解和类型注解

(1) 变量类型注解

针对复合数据类型的高级语法：

变量名: 复合数据类型别名[元素类型] = 初值

此时不能直接用复合数据类型，而是改用其别名，例如 list 的别名是 List。
别名都定义在 typing 模块，使用前需要导入。

from typing **import** List # List为list的别名

a: list[str] = ['a', 'b'] # PyCharm给出警告：无法直接形参化内置 'list'

b: List[str] = ['a', 'b'] # 变量类型注解，为list类型，且元素类型为str

c: List[str] = [1, 2] # PyCharm给出警告：应为类型 'List[str]'，但实际为 'List[int]'

6.2.14 Python中的参数注解和类型注解

(1) 变量类型注解

注意：元组是不可变对象，注解时需要分别指定每个元素的类型。

```
from typing import Tuple # Tuple为tuple的别名
```

```
a: tuple[str, int] = ('Jack', 24) # PyCharm给出警告：无法直接形参化内置'tuple'
```

```
# 变量类型注解，为tuple类型，有两个元素，元素类型为分别为str, int
```

```
b: Tuple[str, int] = ('Jack', 24)
```

```
# PyCharm给出警告：应为类型'Tuple[str, int]'，但实际为'Tuple[str, int, str]'
```

```
c: Tuple[str, int] = ('Jack', 24, 'Beijing')
```

6.2.14 Python中的参数注解和类型注解

(1) 变量类型注解

from typing **import** Dict *# dict为Dict的别名*

PyCharm给出警告: 无法直接形参化内置 'dict'

a: dict[str, int] = {'China': 24, 'USA': 20}

变量类型注解, 为dict类型, 键值对key/value的类型为str/int

b: Dict[str, int] = {'China': 24, 'USA': 20}

PyCharm给出警告: 应为类型 'Dict[str, int]', 但实际为 'Dict[str, str]'

c: Dict[str, int] = {'China': '24'}

6.2.14 Python中的参数注解和类型注解

(2) 函数注解

基本语法:

def 函数名(参数1: 类型, ..., 参数n: 类型) -> 返回类型:

- 函数注解信息保存在__annotations__属性, 是一个字典。

: int 标识了参数应该出现的类型, 为参数注解; -> int 标注返回值为int类型

```
def add(x: int, y: int) -> int:
```

```
    return x + y
```

```
print(add.__annotations__) # 输出函数注解信息
```

```
print(add(3, 5))
```

```
print(add(3.0, 5)) # PyCharm给出警告: 应为类型 'int', 但实际为 'float'
```

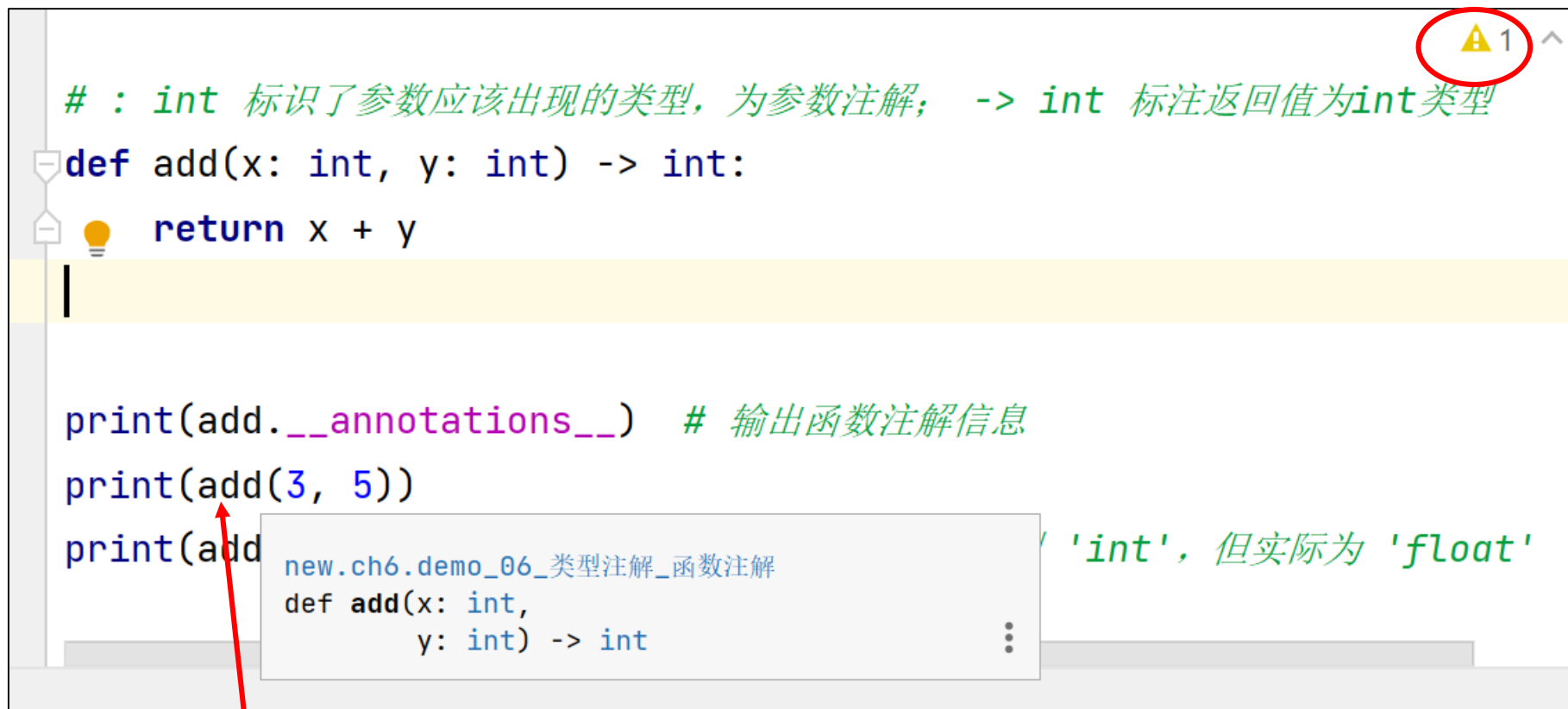
```
{'x': <class 'int'>, 'y': <class 'int'>, 'return': <class 'int'>}
```

```
8
```

```
8.0
```


6.2.14 Python中的参数注解和类型注解

(2) 函数注解



```
# : int 标识了参数应该出现的类型, 为参数注解; -> int 标注返回值为int类型
def add(x: int, y: int) -> int:
    return x + y

print(add.__annotations__) # 输出函数注解信息
print(add(3, 5))
print(add(3, 5.5)) # 'int', 但实际为 'float'
```

new.ch6.demo_06_类型注解_函数注解

```
def add(x: int,
       y: int) -> int
```

鼠标移到这里

6.2.14 Python中的参数注解和类型注解

(2) 函数注解

写在“:”号后面的并不一定是一个类型。

Python把这种写法称为“**annotations**”(标注)，在运行的时候完全不使用它。它是专门设计出来给程序员和自动处理程序看的。任何可被计算出来的东西都可以写在那里。


```
def send_mail(sender: "fish@example.com",
               receiver: "panda@example.com",
               subject: "say hello to you",
               message: "hello",
               attachments: list("type<io.BytesIO>"))
    ) -> bool:
    pass
```

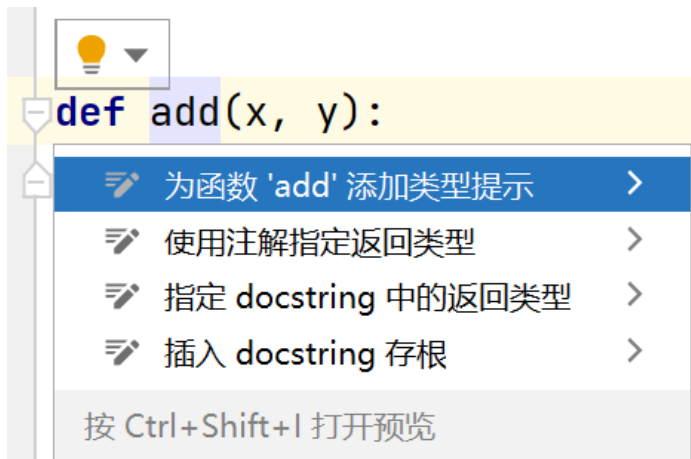
```
print(send_mail.__annotations__) # 输出函数注解
```

```
{'sender': 'fish@example.com', 'receiver': 'panda@example.com',
'subject': 'say hello to you', 'message': 'hello', 'attachments': ['t', 'y', 'p', 'e',
'<', 'i', 'o', '.', 'B', 'y', 't', 'e', 's', 'I', 'O', '>'], 'return': <class 'bool'>}
```

6.2.14 Python中的参数注解和类型注解

(3) PyCharm 中交互式添加函数注解

单击函数名 `add`，1~2 秒后显示小灯泡 ，点击它，从弹出的菜单中选择“为函数 '`add`' 添加类型提示 (type hints)”



```
def add(x: object, y: object) -> object:  
    return x + y
```

参数及返回值的默认类型为 `object`，修改它

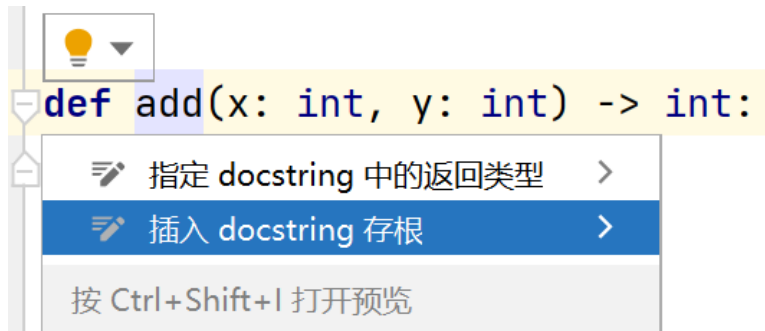


```
def add(x: int, y: int) -> int:  
    return x + y
```

6.2.14 Python中的参数注解和类型注解

(4) PyCharm 中为函数添加文档字符串(docstring)

方法1: 单击函数名add, 1~2秒后显示小灯泡, 点击小灯泡图片, 从弹出的菜单中选择 “插入docstring存根(stub)”



```
def add(x: int, y: int) -> int:
    """
    |
    @param x:
    @param y:
    @return:
    """
    return x + y
```

方法2: 直接输入3个双引号后回车也可自动生成此文档说明框架



```
def add(x: int, y: int) -> int:
    """
    计算两数之和。

    :param x: 第一个操作数
    :param y: 第二个操作数
    :return: 返回两数之和
    """
    return x + y
```

在自动生成的文档字符串框架中输入函数功能说明、参数说明和返回类型说明。
@param 和 @return 为文档说明关键字, 将@删除, 否则报错。

6.2.14 Python中的参数注解和类型注解

(4) PyCharm 中为函数添加文档字符串(docstring)

```
def add(x: int, y: int) -> int:
    """
    计算两数之和。

    :param x: 第一个操作数
    :param y: 第二个操作数
    :return: 返回两数之和
    """
    return x + y
```

```
print(add(3, 5))
```

```
new.ch6.demo_06_类型注解_函数注解
def add(x: int,
        y: int) -> int
```

计算两数之和。

形参: x – 第一个操作数
y – 第二个操作数

返回值: 返回两数之和

⋮

第六章 函数

6.1 函数的概述

6.2 函数的参数和返回值

6.3 函数的调用

6.4 Python内置函数

6.5 实验

6.6 小结

6.7 习题

6.3.1 函数的调用方法

要调用一个函数，需要知道函数的名称和参数。

函数分为**自定义函数**和**内置函数**。

自定义函数需要先定义再调用，内置函数直接调用，有的内置函数是在特定的模块下，这时需要用import命令导入模块后再调用。

可以在交互式命令行通过help(函数名)查看函数的帮助信息。

调用函数的时候，如果传入的参数数量不对，会报 TypeError 的错误，同时Python会明确地告诉你参数的个数。如果传入的参数数量是对的，但参数类型不能被函数所接受，也会报 TypeError 的错误，同时给出错误信息。

函数名其实就是指向一个**函数对象**的引用，可以把函数名赋给一个变量。

6.3.2 嵌套调用

允许在函数内部创建另一个函数，这种函数叫**内嵌函数**或者**内部函数**。**内嵌函数的作用域在其内部**，如果内嵌函数的作用域超出了这个范围就不起作用。如下所示代码：

```
>>> def function_1():  
    print('正在调用function_1()...')  
  
    def function_2():  
        print('正在调用function_2()...')  
  
    function_2()
```


6.3.2 嵌套调用

运行结果如下：

```
>>> function_1()
```

```
正在调用function_1()...
```

```
正在调用function_2()...
```

```
>>> function_2()
```

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

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

```
    function_2()
```

```
NameError: name 'function_2' is not defined
```

6.3.3 使用闭包

闭包是函数式编程的一个重要的语法结构。从表现形式上定义为，如果在一个内部函数里对**一个外部作用域（但不是在全局作用域）的变量进行引用**，那么内部函数就认为是闭包(closure)。如下所示代码：

```
def Fun_sub(a):  
    def Fun_sub2(b):  
        return a-b  
    return Fun_sub2  
  
i = float(input('请输入减数： '))  
j = float(input('请输入被减数： '))
```

6.3.3 使用闭包

```
print(Fun_sub(i)(j))
```

运行结果如下：

请输入减数： 67

请输入被减数： 45

22.0

```
>>> Fun_sub2(23)
```

Traceback (most recent call last):

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

Fun_sub2(23)

6.3.3 使用闭包

NameError: name 'Fun_sub2' is not defined

在上例中，内部函数Fun_sub2()引用了外部作用域的变量a。如果我们在全局范围内直接访问闭包Fun_sub2()，程序会报错，提示闭包函数Fun_sub2()没有定义。

在调用的时候要注意：不能在全局域内访问闭包，否则会出错。在闭包中，外部函数的局部变量对闭包中的局部变量（相当于全局变量和局部变量的关系），在闭包中能访问外部函数的局部变量，但是不能进行修改。

6.3.4 递归调用

递归是算法的范畴，从本质上讲不是Python的语法范围。

函数调用自身的行为是递归。

递归的2个条件：调用函数自身，设置了正确的返回条件。递归即是有进去必须有回来。

Python默认递归深度100层（Python限制）。设置递归的深度的系统函数是：`sys.setrecursionlimit(stepcount)`。参数：`stepcount`设置递归的深度。

递归有危险性：消耗时间和空间，因为递归是基于弹栈和出栈操作。递归忘掉返回使程序崩溃，消耗掉所有内存。

第六章 函数

6.1 函数的概述

6.2 函数的参数和返回值

6.3 函数的调用

6.4 Python内置函数

6.5 实验

6.6 小结

6.7 习题

<u>abs()</u>	<u>divmod()</u>	<u>input()</u>	<u>open()</u>	<u>staticmethod()</u>
<u>all()</u>	<u>enumerate()</u>	<u>int()</u>	<u>ord()</u>	<u>str()</u>
<u>any()</u>	<u>eval()</u>	<u>isinstance()</u>	<u>pow()</u>	<u>sum()</u>
<u>basestring()</u>	<u>execfile()</u>	<u>issubclass()</u>	<u>print()</u>	<u>super()</u>
<u>bin()</u>	<u>file()</u>	<u>iter()</u>	<u>property()</u>	<u>tuple()</u>
<u>bool()</u>	<u>filter()</u>	<u>len()</u>	<u>range()</u>	<u>type()</u>
<u>bytearray()</u>	<u>float()</u>	<u>list()</u>	<u>raw_input()</u>	<u>unichr()</u>
<u>callable()</u>	<u>format()</u>	<u>locals()</u>	<u>reduce()</u>	<u>unicode()</u>
<u>chr()</u>	<u>frozenset()</u>	<u>long()</u>	<u>reload()</u>	<u>vars()</u>
<u>classmethod()</u>	<u>getattr()</u>	<u>map()</u>	<u>repr()</u>	<u>xrange()</u>
<u>cmp()</u>	<u>globals()</u>	<u>max()</u>	<u>reverse()</u>	<u>zip()</u>
<u>compile()</u>	<u>hasattr()</u>	<u>memoryview()</u>	<u>round()</u>	<u>__import__()</u>
<u>complex()</u>	<u>hash()</u>	<u>min()</u>	<u>set()</u>	
<u>delattr()</u>	<u>help()</u>	<u>next()</u>	<u>setattr()</u>	
<u>dict()</u>	<u>hex()</u>	<u>object()</u>	<u>slice()</u>	
<u>dir()</u>	<u>id()</u>	<u>oct()</u>	<u>sorted()</u>	<u>exec</u> 内置表达式

6.4.1 与类型相关的常用内置函数

与类型相关的指，把参数包装或转换为某种类型，这样的内置函数包括：

- `bool()` —布尔型
- `int()` —整形
- `str()` —字符型
- `range()` —不可更改的序列
- `tuple()` —元包型
- `dict()` —字典型
- `list()` —列表型
- `set()` —集合类型
- `frozenset()` # 冻结集合类型，不允许修改
- `zip()` —可迭代对象聚合，(,)
- `complex()` —复数型
- `float()` —浮点型
- `bytes()` —字节型数组
- `bytearray()` —字数数组
- `object()` —无属性的根类
- `slice()` # 返回一个slice对象，其中start, stop, step等都是只读的

6.4.2 与数理统计相关的常用内置函数

有的内置函数可以完成简单的数理统计工作，这样的内置函数包括：

- `abs()` —绝对值
- `min()` —最小值
- `max()` —最大值
- `sum()` —求和
- `pow()` —求次幂
- `all()` —所有元素为true则为true
- `any()` —至少一个元素为true则为true
- `divmod()` —(商, 余数)
- `round()` —四舍五入
- `len()` —参数元素个数

```
>>>a=[2,-5,8,0,9]
>>> all(a)
False
>>> any(a)
True
```

```
>>>a = 101
... b = 5
... # d为商, d为余数
... c, d = divmod(a, b)
... print(c, d)
...
20 1
```

6.4.3 与进制转换相关的常用内置函数

有些内置函数可以帮助我们轻松实现进制转换，这样的内置函数包括：

- chr() —unicode编码
- ord() —chr()反操作
- bin() —转化为0b开头的二进制字符
- oct() — 转化为0o开头的八进制字符
- hex() —转化为0x开头的十六进制字符
- ascii() —可打印表示对象，类似于 repr()

```
>>>ascii(90)
'90'
>>> ascii('abc')
'"abc"'
>>> ascii(True)
'True'
```

```
>>>hex(16)
'0x10'
>>> oct(16)
'0o20'
>>> bin(16)
'0b10000'
```

6.4.4 与面向对象相关的常用内置函数

Python提供与对象属性相关的操作函数，它们为满足Python属性的动态调整提供了可能。

- `setattr(object, name, value)` — 为对象设置属性
- `delattr(object, name)` — 删除命名的属性
- `getattr(object, name)` — 获取属性的取值，如果对象无此属性，会抛异常
- `getattr(object, name, 123)` — 即便无此属性，也不会抛异常，会返回123
- `hasattr(object, name)` — 判断name属性是否属于object
- `isinstance(object, classinfo)` — 判断object是classinfo的实例吗
- `issubclass(class, classinfo)` — 判断class是否为classinfo的子类
- `super()` — 调用父类，方法
- `property()` — 特性相关，`@property`标记为属性
- `type()` — 返回实例的类型
- `vars()` — 返回对象的信息等
- `classmethod()` — 转化方法为类方法
- `staticmethod()` — 方法是静态方法

6.4.5 与迭代器相关的常用内置函数

- reversed() —对列表的元素进行反向排序，返回一个可迭代对象。
- iter() —用来构造迭代器。
- next() —用于遍历迭代器，每次返回迭代器的下一个项目。next() 函数要和生成迭代器的iter() 函数一起使用。

语法：next(iterator[, default])

可选参数default 用于设置在没有下一个元素时返回该默认值，如果不设置，又没有下一个元素则会触发 StopIteration 异常。

- enumerate() —用于构造枚举对象，实现对序列、迭代器等对象中的元素的枚举。也就是将一个可遍历的数据对象(如列表、元组或字符串)组合为一个索引序列，同时列出数据和数据下标，一般用在 for 循环当中。

语法：enumerate(iterable, start=0)，start表示枚举值（索引号）的起始值，默认从0开始。函数返回一个可迭代对象，其中包含的元素为一个个有枚举值和枚举元素构成的二元组。

```
>>>aList = [123, 'zara', 'abc', 'xyz']
>>>list(reversed(aList))
['xyz', 'abc', 'zara', 123]
>>>aList
[123, 'zara', 'abc', 'xyz']
```

```
>>>obj=iter([1,8])
>>>print(obj)
<list_iterator object at 0x00000293E6714E20>
>>> for x in obj:
...     print(x)
1
8
```

next()函数使用示例：遍历迭代器

```
it = iter([1, 1, 9]) # 构造Iterator对象
```

```
while True: # 循环
```

```
    try:
```

```
        x = next(it) # 获得下一个值
```

```
        print(x)
```

```
    except StopIteration:
```

```
        break # 遇到StopIteration异常就退出循环
```

运行结果：

1

1

9

如果传入第二个参数default, 获取最后一个元素之后, 下一次next返回该默认值, 而不会抛出 StopIteration异常。

```
it = iter([1, 1, 9]) # 构造Iterator对象
```

```
while True: # 循环
```

```
    x = next(it, None) # 获得下一个值, 且默认值为None
```

```
    if x is not None:
```

```
        print(x)
```

```
    else:
```

```
        break
```

运行结果：

1

1

9

enumerate()函数使用示例：对一年春、夏、秋、冬四季构造枚举对象。

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]

>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

```
# 给每一个元素加一个枚举值（索引号）
seasons = ['Spring', 'Summer', 'Fall', 'Winter']
for k, season in enumerate(seasons):
    print(k, season)

for k, season in enumerate(seasons, start=1):
    print(k, season)
```

运行结果：

```
0 Spring
1 Summer
2 Fall
3 Winter
1 Spring
2 Summer
3 Fall
4 Winter
```

6.4.6 高级的常用内置函数

- **map()**

根据提供的函数对指定序列做映射。函数原型为：`map(function, iterable)`

第一个参数 `function` 以 `iterable` 中的每一个元素调用 `function` 函数，返回包含每次 **function 函数返回值** 的可迭代对象。

- **filter()**

用于过滤序列，过滤掉不符合条件的元素，返回由符合条件元素组成的可迭代对象。

函数原型为：`filter(function, iterable)`

该接收两个参数，第一个为**布尔函数**，第二个为序列，序列的每个元素作为参数传递给函数进行判断，然后返回 `True` 或 `False`，最后将返回 `True` 的元素放到可迭代对象中。

- **reduce()**—需要载入 `functools` 模块

对序列中元素按某个函数指定的运算规则进行累积（规约），例如所有元素求和。

函数原型为：`reduce(function, iterable)`

该函数将一个数据集合（列表，元组等）中的所有数据进行下列操作：

用传给 `reduce` 中的函数 `function` (**有两个参数**) 先对集合中的第 1、2 个元素进行操作，得到的结果再与第三个数据用 `function` 函数运算，依次类推，最后得到一个结果。

map()函数使用示例: 对列表的每个元素求平方

```
def f(x):  
    return x**2
```

```
v = [2, 0, 5, 9]
```

第一个参数: 必须是函数, 可以是命名函数, 也可以是匿名函数

第二个参数: 必须是可迭代类型 (可以被for循环的)

```
result = map(f, v) # map() 将f(x)函数的返回值添加到一个可迭代对象中
```

```
print(list(result)) # 可迭代对象必须要转换成列表才能看见
```

也可以简写:

```
result = map(lambda x: x**2, v) # 使用匿名函数
```

```
print(list(result))
```

运行结果:

```
[4, 0, 25, 81]
```

```
[4, 0, 25, 81]
```


filter()函数使用示例: 过滤出列表中的所有奇数

```
def is_odd(n): # 定义布尔函数
    return n%2==1
```

```
v = [2, 0, 5, 9, 4, 3, 17]
```

```
# 第一个参数: 必须是布尔类型的函数
```

```
# 第二个参数: 必须是可迭代类型 (可以被for循环的)
```

```
result = filter(is_odd, v) # 若is_odd(x)函数返回True,
                           # 则filter()将元素x添加到一个可迭代对象
```

```
print(list(result)) # 可迭代对象必须要转换成列表才能看见
```

```
# 也可以采用匿名函数
```

```
result = filter(lambda n: True if n % 2 == 1 else False, v)
```

```
print(list(result))
```

运行结果:

```
[5, 9, 3, 17]
```

```
[5, 9, 3, 17]
```

```
>>> filter(lambda x:x>0,range(5))
```

```
>>> tuple(filter(lambda x:x>0,range(5)))
(1, 2, 3, 4)
```

reduce()函数使用示例：计算列表和：1+2+3+4+5

```
import functools
```

```
def add(x, y): # 定义两个参数的函数  
    return x + y
```

```
v = range(1, 6)  
result = functools.reduce(add, v) # add(x,y)函数先对集合中的第 1、2 个元素求和，  
# 得到的结果再与第3个数据用 add(x,y)函数运算，依次类推，最后得到一个结果  
print(result)
```

```
# 也可以采用匿名函数  
result = functools.reduce(lambda x, y: x + y, v)  
print(result)
```

运行结果：

15

15

6.4.7 其他的常用内置函数

- `sorted()` — 返回一个排序好的列表
- `format()` — 对象格式化
- `input()` — 与标准输入相关
- `print()` — 与标准输出相关
- `help()` — 查看帮助
- `dir()` — 查看对象的方法和属性
- `id()` — 返回一个对象的标识
- `open()` — 打开文件
- `compile()` — 与源码编译相关
- `memoryview()` — 与内存视图相关
- `hash()` — 返回对象的哈希码
- `breakpoint()` — 调试相关
- `exec()` — 动态执行Python代码
- `callable()` — 判断对象是否可调用

第六章 函数

6.1 函数的概述

6.2 函数的参数和返回值

6.3 函数的调用

6.4 Python内置函数

6.5 实验

6.6 小结

6.7 习题

6.4.1 声明和调用函数

6.4.2 在调试窗口中查看变量的值

6.4.3 使用函数参数和返回值

6.4.4 使用闭包和递归函数

6.4.5 使用python的内置函数

第六章 函数

6.1 函数的概述

6.2 函数的参数和返回值

6.3 函数的调用

6.4 Python内置函数

6.5 实验

6.6 小结

6.7 习题

定义函数时，需要确定函数名和参数个数；函数体内用return返回函数结果；函数没有return语句或return语句后面为空时，自动返回None。函数可以同时返回多个值。

默认值参数一定要用不可变对象，如果是可变对象，程序运行时会有逻辑错误。

命名的关键字参数是为了限制调用者可以传入的参数名，同时可以提供默认值。

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。

第六章 函数

6.1 函数的概述

6.2 函数的参数和返回值

6.3 函数的调用

6.4 Python内置函数

6.5 实验

6.6 小结

6.7 习题

习题:

1. 在函数内部可以通过什么关键字来定义全局变量?
2. 如果函数中没有return语句或者return语句不带任何返回值, 那么该函数的返回值是什么?
3. 写函数, 接收一个参数(此参数类型必须是可迭代对象), 将可迭代对象的每个元素以 '_' 相连接, 形成新的字符串, 并返回。
例如, 传入的可迭代对象为[1, '天王', '刘德华']返回的结果为'1_天王_刘德华'。
4. 写函数, 传入n个数, 返回字典{'max':最大值, 'min':最小值}
例如, min_max(2,5,7,8,4) 返回: {'max':8, 'min':2}(此题用到max(), min()内置函数)
5. 写函数, 传入一个参数n, 返回n的阶乘。例如, cal(7) 计算7654321。
6. 写函数, 返回一个扑克牌列表, 里面有52项, 每一项是一个元组。
例如: [('红心', 2), ('草花', 2), ... ('黑桃', 'A')]

习题:

7. 分别使用递归函数，完成：

- ① 求阶乘 $n! = n * (n-1) * (n-2) * \dots * 2 * 1$
- ② 求和 $s(n) = n + (n-1) + (n-2) + \dots + 2 + 1$
- ③ 计算Fibonacci数列 $\text{Fib}(n)$: 1, 1, 2, 3, 5, 8, 13, ..., $\text{Fib}(n-2) + \text{Fib}(n-1)$
- ④ 用辗转相除法求最大公约数 $\text{GCD}(m, n)$
- ⑤ 模拟汉若塔游戏 $\text{Hanoi}(n, \text{start} = 'A', \text{cache} = 'B', \text{target} = 'C')$ ，借助B柱，将n个盘子从A柱移至C柱。n个盘子和3根柱子：A(源)、B(备用)、C(目的)，盘子的大小不同且中间有一孔，可以将盘子“串”在柱子上，每个盘子只能放在比它大的盘子上面。起初，所有盘子在A柱上，问题是将盘子一个一个地从A柱子移动到C柱子。移动过程中，可以使用B柱，但盘子也只能放在比它大的盘子上面。

习题：

8. 使用内置函数map()求字符串列表中每一个字符串的长度。
9. 使用内置函数map()判断整数列表中的每一个数是否为素数(质数)。
- 10.使用内置函数filter()过滤出1~100中平方根是整数的数。
- 11.使用内置函数filter()过滤出列表[8,'Guido van Rossum',5,69,'人生苦短',33,'我用python',2] 中的字符串。
- 12.使用内置函数reduce()找出列表[2, 8, 7, 4, 0, 99, 3]中的最大元素。

感谢聆听

