Python函数

函数

数学定义

- y=f(x)，y是x的函数，x是自变量。y=f(x0, x1, ..., xn)

Python函数

- 由若干语句组成的语句块、函数名称、参数列表构成，它是组织代码的最小单元
- 完成一定的功能

函数的作用

- 结构化编程对代码的最基本的**封装**，一般按照功能组织一段代码
- 封装的目的为了**复用**，减少冗余代码
- 代码更加简洁美观、可读易懂

函数的分类

- 内建函数，如max()、reversed()等
- 库函数，如math.ceil()等
- 自定义函数，使用def关键字定义

函数定义

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

- 函数名就是标识符，命名要求一样
- 语句块必须缩进，约定4个空格
- Python的函数若没有return语句，会隐式返回一个None值
- 定义中的参数列表称为**形式参数**，只是一种符号表达（标识符），简称**形参**

函数调用

- 函数定义，只是声明了一个函数，它不能被执行，需要调用执行
- 调用的方式，就是**函数名后加上小括号**，如有必要在括号内填写上参数
- 调用时写的参数是**实际参数**，是实实在在传入的值，简称**实参**

```python
def add(x, y): # 函数定义
    result = x + y # 函数体
    return result  # 返回值

out = add(4,5) # 函数调用，可能有返回值，使用变量接收这个返回值
print(out) # print函数加上括号也是调用
```

上面代码解释：

- 定义一个函数add，及函数名是add，能接受2个形式参数
- 该函数计算的结果，通过返回值返回，需要return语句
- 调用时，通过函数名add后加2个实际参数，返回值可使用变量接收
- **函数名也是标识符**
- **返回值也是值**
- 定义需要在调用前，也就是说调用时，已经被定义过了，否则抛NameError异常
- 函数是**可调用的对象**，callable(add)返回True

看看这个函数是不是通用的？体会一下Python函数的好处

函数参数

函数在定义是要定义好形式参数，调用时也提供足够的实际参数，一般来说，形参和实参个数要一致（可变参数除外）。

实参传参方式

1、位置传参

定义时def f(x, y, z)， 调用使用 f(1, 3, 5)，按照参数定义顺序传入实参

2、关键字传参

定义时def f(x, y, z)，调用使用 f(x=1, y=3, z=5)，使用形参的名字来传入实参的方式，如果使用了形参名字，那么**传参顺序就可和定义顺序不同**

要求位置参数必须在关键字参数之前传入，位置参数是按位置对应的

```python
def add(x, y):
    print(x)
    print(y)
    print('-' * 30)

add(4, 5)
add(5, 4) # 按顺序对应，反过来x和y值就不同

add(x=[4], y=(5,))
add(y=5.1, x=4.2) # 关键字传参，按名字对应，无所谓顺序

add(4, y=5) # 正确
add(y=5, 4) # 错误传参
```

切记：传参指的是**调用时**传入实参，就2种方式。

下面讲的都是形参定义。

形参缺省值

缺省值也称为默认值，可以在函数定义时，为形参增加一个缺省值。其作用：

- 参数的默认值可以在未传入足够的实参的时候，对没有给定的参数赋值为默认值
- 参数非常多的时候，并不需要用户每次都输入所有的参数，简化函数调用

```
def add(x=4, y=5):
    return x+y

测试调用 add()、add(x=5)、add(y=7)、add(6, 10)、add(6, y=7)、add(x=5, y=6)、
add(y=5, x=6)、add(x=5, 6)、add(y=8, 4)、add(11, x=20)

能否这样定义 def add(x, y=5) 或 def add(x=4,y) ？
```

```
# 定义一个函数login，参数名称为host、port、username、password
def login(host='localhost', port=3306, username='root', password='root'):
    print('mysql://{2}:{3}@{0}:{1}/'.format(host, port, username, password))

login()
login('127.0.0.1')
login('127.0.0.1', 3361, 'wayne', 'wayne')
login('127.0.0.1', username='wayne')
login(username='wayne', password='wayne', host='www.magedu.com')
```

可变参数

需求：写一个函数，可以对多个数累加求和

```
def sum(iterable):
    s = 0
    for x in iterable:
        s += x
    return s

print(sum([1,3,5]))
print(sum(range(4)))
```

上例，传入可迭代对象，并累加每一个元素。

也可以使用可变参数完成上面的函数。

```
def sum(*nums):
    sum = 0
    for x in nums:
        sum += x
    return sum

print(sum(1, 3, 5))
print(sum(1, 2, 3))
```

1、可变位置参数

- 在形参前使用 * 表示该形参是可变位置参数，可以接受多个实参
- 它将收集来的实参组织到一个tuple中

2、可变关键字参数

- 在形参前使用 ** 表示该形参是可变关键字参数，可以接受多个关键字参数
- 它将收集来的实参的名称和值，组织到一个dict中

```
def showconfig(**kwargs):
    for k,v in kwargs.items():
        print('{}={}'.format(k,v), end=', ')

showconfig(host='127.0.0.1', port=8080, username='wayne', password='magedu')
```

混合使用

```
可以定义为下列方式吗？
def showconfig(username, password, **kwargs)
def showconfig(username, *args, **kwargs)
def showconfig(username, password, **kwargs, *args) # ?
```

总结：

- 有可变位置参数和可变关键字参数
- 可变位置参数在形参前使用一个星号*
- 可变关键字参数在形参前使用两个星号**
- 可变位置参数和可变关键字参数都可以收集若干个实参，可变位置参数收集形成一个tuple，可变关键字参数收集形成一个dict
- 混合使用参数的时候，普通参数需要放到参数列表前面，可变参数要放到参数列表的后面，可变位置参数需要在可变关键字参数之前

使用举例

```
def fn(x, y, *args, **kwargs):
    print(x, y, args, kwargs, sep='\n', end='\n\n')

fn(3, 5, 7, 9, 10, a=1, b='abc')
fn(3, 5)
fn(3, 5, 7)
fn(3, 5, a=1, b='abc')
fn(x=1, y=2, z=3)
fn(x=3, y=8, 7, 9, a=1, b='abc') # ?
fn(7, 9, y=5, x=3, a=1, b='abc') # ?
```

fn(x=3, y=8, 7, 9, a=1, b='abc')，错在位置传参必须在关键字传参之前
fn(7, 9, y=5, x=3, a=1, b='abc')，错在7和9已经按照位置传参了，x=3、y=5有重复传参了

keyword-only参数

先看一段代码

```
def fn(*args, x, y, **kwargs):
    print(x, y, args, kwargs, sep='\n', end='\n\n')

fn(3, 5) #
fn(3, 5, 7) #
fn(3, 5, a=1, b='abc') #
fn(3, 5, y=6, x=7, a=1, b='abc')
```

在Python3之后，新增了keyword-only参数。

keyword-only参数：在形参定义时，在一个*星号之后，或一个可变位置参数之后，出现的普通参数，就已经不是普通的参数了，称为keyword-only参数。

```
def fn(*args, x):
    print(x, args, sep='\n', end='\n\n')

fn(3, 5) #
fn(3, 5, 7) #
fn(3, 5, x=7)
```

keyword-only参数，言下之意就是这个参数必须采用关键字传参。

可以认为，上例中，args可变位置参数已经截获了所有位置参数，其后的变量x不可能通过位置传参传入了。

思考：def fn(**kwargs, x) 可以吗?

```
def fn(**kwargs, x):
    print(x, kwargs, sep='\n', end='\n\n')
```

直接语法错误了。

可以认为，kwargs会截获所有关键字传参，就算写了x=5，x也没有机会得到这个值，所以这种语法不存在。

keyword-only参数另一种形式

* 星号后所有的普通参数都成了keyword-only参数。

```
def fn(*, x, y):
    print(x, y)
fn(x=6, y=7)
fn(y=8, x=9)
```

Positional-only参数

Python 3.8 开始，增加了最后一种形参类型的定义：Positional-only参数。（2019年10月发布3.8.0）

```
def fn(a, /):
    print(a, sep='\n')

fn(3)
fn(a=4) # 错误，仅位置参数，不可以使用关键字传参
```

参数的混合使用

```
# 可变位置参数、keyword-only参数、缺省值
def fn(*args, x=5):
    print(x)
    print(args)
fn() # 等价于fn(x=5)
fn(5)
fn(x=6)
fn(1,2,3,x=10)
```

```
# 普通参数、可变位置参数、keyword-only参数、缺省值
def fn(y, *args, x=5):
    print('x={}, y={}'.format(x, y))
    print(args)
fn() #
fn(5)
fn(5, 6)
fn(x=6)  #
fn(1, 2, 3, x=10)
fn(y=17, 2, 3, x=10)  #
fn(1, 2, y=3, x=10)  #
fn(y=20, x=30)
```

```
# 普通参数、缺省值、可变关键字参数
def fn(x=5, **kwargs):
    print('x={}'.format(x))
    print(kwargs)
fn()
fn(5)
fn(x=6)
fn(y=3, x=10)
fn(3, y=10)
fn(y=3, z=20)
```

参数规则

参数列表参数一般顺序是：positional-only参数、普通参数、缺省参数、可变位置参数、keyword-only参数（可带缺省值）、可变关键字参数。

注意：

- 代码应该易读易懂，而不是为难别人
- 请按照书写习惯定义函数参数

```
def fn(a, b, /, x, y, z=3, *args, m=4, n, **kwargs):
    print(a, b)
    print(x, y, z)
    print(m, n)
    print(args)
    print(kwargs)
    print('-' * 30)


def connect(host='localhost', user='admin', password='admin', port='3306', **kwargs):
    print('mysql://{}:{}@{}:{}/{}'.format(
        user, password, host, port, kwargs.get('db', 'test')
    ))

connect(db='cmdb') # 参数的缺省值把最常用的缺省值都写好了
connect(host='192.168.1.123', db='cmdb')
connect(host='192.168.1.123', db='cmdb', password='mysql')
```

- 定义最常用参数为普通参数，可不提供缺省值，必须由用户提供。注意这些参数的顺序，最常用的先定义
- 将必须使用名称的才能使用的参数，定义为keyword-only参数，要求必须使用关键字传参
- 如果函数有很多参数，无法逐一定义，可使用可变参数。如果需要知道这些参数的意义，则使用可变关键字参数收集

参数解构

```
def add(x, y):
    print(x, y)
    return x + y

add(4, 5)
add((4, 5)) # 可以吗？
t = 4, 5
add(t[0], t[1])
add(*t)
add(*(4, 5))
add(*[4, 5])
add(*{4, 5}) # 注意有顺序吗？
add(*range(4, 6))

add(*{'a':10, 'b':11}) # 可以吗？
add(**{'a':10, 'b':11}) # 可以吗？
add(**{'x':100, 'y':110}) # 可以吗？
```

参数解构：

- 在给函数提供实参的时候，可以在可迭代对象前使用 * 或者 ** 来进行结构的解构，提取出其中所有元素作为函数的实参
- 使用 * 解构成位置传参
- 使用 ** 解构成关键字传参
- 提取出来的元素数目要和参数的要求匹配

```python
def add(*nums):
    result = 0
    for x in nums:
        result += x
    return result

add(1, 2, 3)
add(*[1, 3, 5])
add(*range(5))
```

```python
# 3.8以后，下面就不可以使用字典解构后的关键字传参了
def add(x, y, /): # 仅位置形参
    print(x, y)
    return x + y


add(**{'x':10, 'y':11})
```

函数返回值

先看几个例子

```python
# return语句之后可以执行吗？
def showplus(x):
    print(x)
    return x + 1
    print('~~end~~') # return之后会执行吗？

showplus(5)

# 多条return语句都会执行吗
def showplus(x):
    print(x)
    return x + 1
    return x + 2

showplus(5)

# 下例多个return可以执行吗？
def guess(x):
    if x > 3:
        return "> 3"
    else:
        return "<= 3"

print(guess(10))
```

```
25
26 # 下面函数执行的结果是什么
27 def fn(x):
28     for i in range(x):
29         if i > 3:
30             return i
31     else:
32          print("{} is not greater than 3".format(x))
33
34 print(fn(5)) # 打印什么？
35 print(fn(3)) # 打印什么？
```

总结

- Python函数使用return语句返回“返回值”
- 所有函数都有返回值，如果没有return语句，隐式调用return None
- return 语句并不一定是函数的语句块的最后一条语句
- 一个函数可以存在多个return语句，但是只有一条可以被执行。如果没有一条return语句被执行到，隐式调用return None
- 如果有必要，可以显示调用return None，可以简写为return
- 如果函数执行了return语句，函数就会返回，当前被执行的return语句之后的其它语句就不会被执行了
- 返回值的作用：结束函数调用、返回“返回值”

能够一次返回多个值吗?

```
1 def showvalues():
2     return 1, 3, 5
3
4 showvalues() # 返回了多个值吗？
```

- 函数不能同时返回多个值
- return 1, 3, 5 看似返回多个值，隐式的被python封装成了**一个元组**
- `x, y, z = showlist()` 使用解构提取返回值更为方便