Python程序设计

第六讲 函数与模块化

函数的定义与调用

张 华
WHU

函数的定义与调用

- 定义函数
- 调用函数
- 函数的参数
- 参数传递
- 函数的返回值

定义函数

函数定义语法：

```
def 函数名([参数列表]):
    '''注释'''
    函数体
```

说明

- 函数形参不需要声明类型，也不需要指定函数返回值类型。
- 即使该函数不需要接收任何参数，也必须保留一对空的圆括号。
- 括号后面的冒号必不可少。
- 函数体相对于def关键字必须保持一定的空格缩进。
- Python允许嵌套定义函数。

定义函数

函数体中的return语句

return [表达式]

说明

- 一旦执行return语句，将直接结束函数的执行。
- 可以使用return语句返回任意类型的值，函数返回值类型与return语句返回表达式的类型一致。
- 如果函数没有return语句，或者有return语句但是没有执行到，或者执行了不返回任何值的return语句，解释器都会认为该函数以return None结束，即返回空值。

调用函数

通过下面的形式调用函数

函数名([参数表])

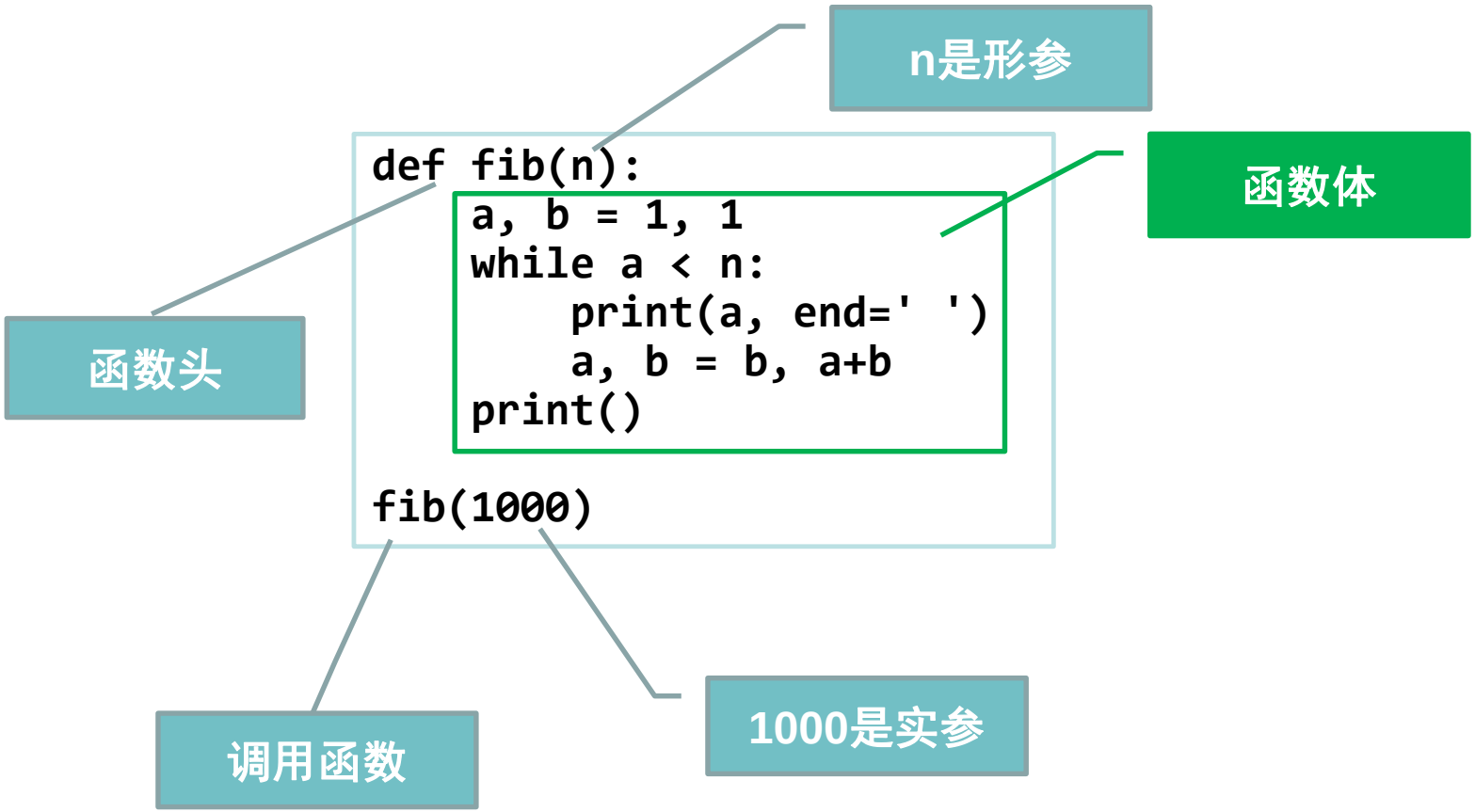
说明

- 函数定义时若有参数，必须在调用时提供实际参数。
- 实际参数个数、位置要与函数定义时的形式参数相对应。
- 函数调用的形式：
  - 直接以语句形式出现
  - 在表达式中出现

函数的定义与调用

示例

- 编写生成斐波那契数列的函数并调用。



函数的注释

- 定义函数时，开头部分的注释并不是必需的，但如果加上注释的话，可以为程序开发人员提供友好的提示。

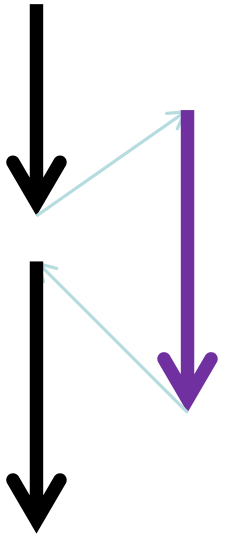
```
>>> def fib(n):
    '''accept an integer n.
       return the numbers less than n in Fibonacci sequence.'''
    a, b = 1, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()


>>> fib(
(n)
accept an integer n.
return the numbers less than n in Fibonacci sequence.
```

用 **fib.__doc__** 表示该注释

函数调用的过程

函数调用的过程

- 当函数调用发生时~
- 调用程序暂停
- 函数形参被赋值为实参(按位置对应)
- 执行函数体
- 控制返回调用点的下一条语句

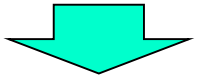


函数的定义与调用举例

案例：生日歌

- 用函数减少重复代码。

```
print("Happy birthday to you!")
print("Happy birthday to you!")
print("Happy birthday, dear Fred.")
print("Happy birthday to you!")
```



```
def happy():
    print("Happy birthday to you!")

happy()
happy()
print("Happy birthday, dear Fred.")
happy()
```

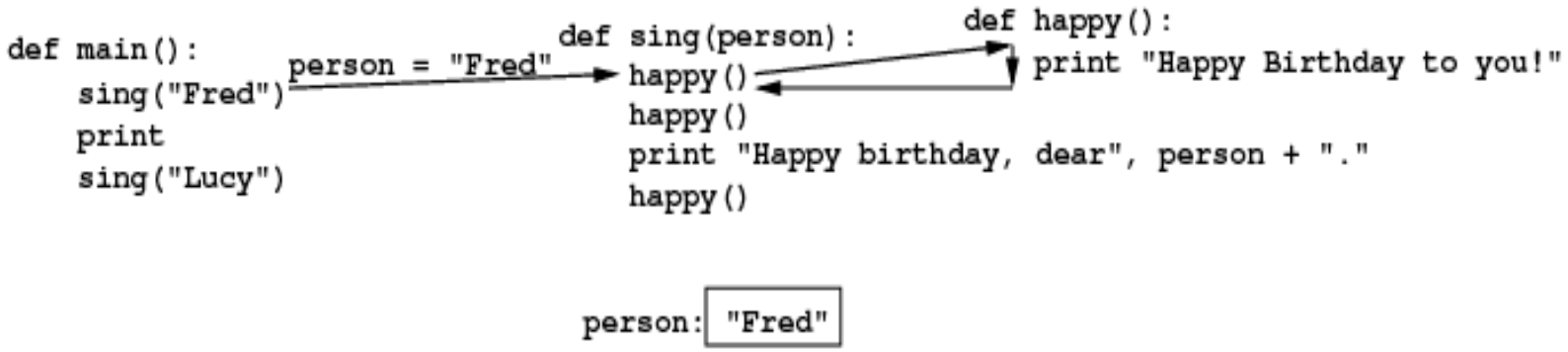
函数的定义与调用举例

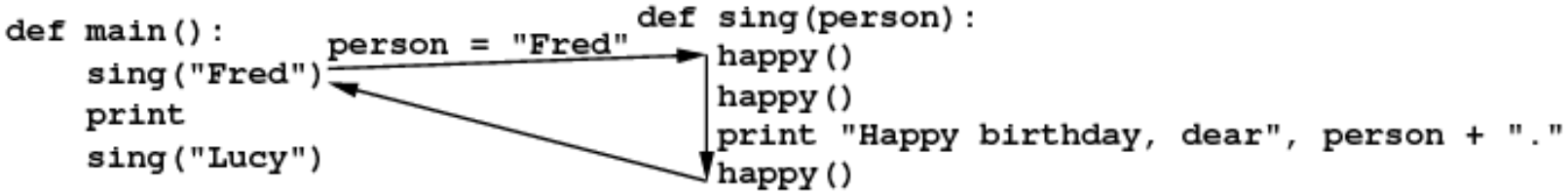
案例：生日歌

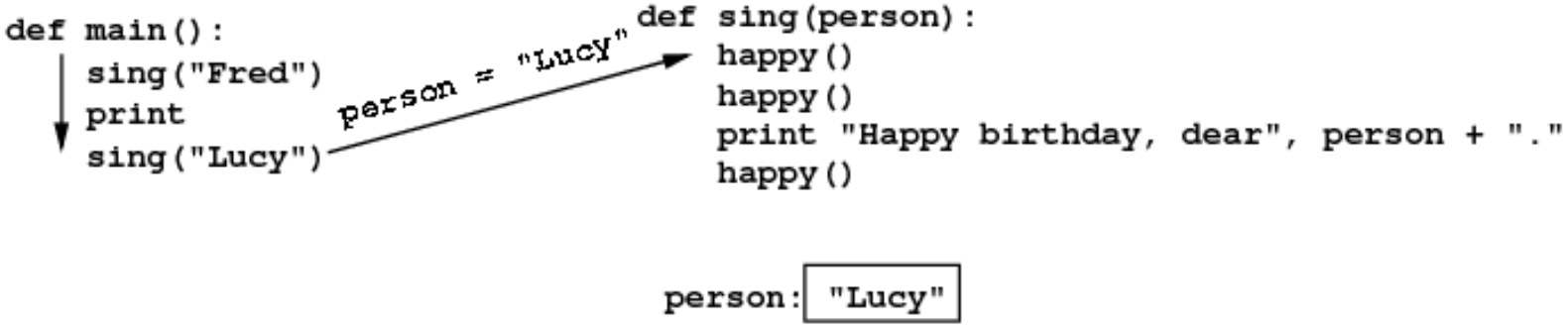
- 用参数增加函数的通用性。

```
def happy():
    print("Happy birthday to you!")

def sing(person):
    happy()
    happy()
    print("Happy birthday, dear", person + ".")
    happy()

def main():
    sing("Fred")
    sing("Lucy")
    sing("Elmer")

main()
```

函数的定义与调用举例


def main():
sing("Fred")
print
sing("Lucy")
person = "Fred"
def sing(person):
happy()
happy()
print "Happy birthday, dear", person + "."
happy()
def happy():
print "Happy Birthday to you!"
person: "Fred"


def main():
sing("Fred")
print
sing("Lucy")
person = "Fred"
def sing(person):
happy()
happy()
print "Happy birthday, dear", person + "."
happy()


def main():
sing("Fred")
print
sing("Lucy")
person = "Lucy"
def sing(person):
happy()
happy()
print "Happy birthday, dear", person + "."
happy()
person: "Lucy"

函数的参数

形式参数与实际参数

- 定义函数时所声明的参数，即为形式参数，简称形参。
- 调用函数时，提供函数所需要的参数的值，即为实际参数，简称实参。

参数传递

- 调用函数时向其传递实参，根据不同的参数类型（位置参数、关键参数、变长参数），将实参的引用传递给形参。
- 定义函数时不需要声明参数类型，解释器会根据实参的类型自动推断形参类型。

函数的参数

示例

形式参数与实际参数

```
def compare(a, b):
    if a > b:
        print(a, '>', b)
    elif a == b:
        print(a, '==', b)
    else:
        print(a, '<', b)
```

```
compare(12, 45)
x, y = 2.3, 0.9
compare(x, y)
compare(10)
```

```
12 < 45
2.3 > 0.9
  File "D:/Course/pycharm/test.py", line 12, in <module>
    compare(10)
TypeError: compare() missing 1 required positional argument: 'b'
```

函数的参数

- 形式参数与局部变量
  - 定义函数时声明的形式参数，等同于函数体中的局部变量，在函数体中的任何位置都可以使用。
  - 局部变量和形式参数变量的区别在于，局部变量在函数体中绑定到某个对象；而形式参数变量则绑定到函数调用代码传递的对应实际参数对象。
- Python参数传递方法是传递对象引用，而不是传递对象的值。

参数传递

传递不可变对象的引用

- 在函数内部直接修改形参的值不会影响实参，而是创建一个新对象。
- 例如：

```
def incOne(x):
    print(id(x), ':', x)
    x += 1
    print(id(x), ':', x)

v = 5
print(id(v))
incOne(v)
print(id(v), ':', v)
```

```
1408003264
1408003264 : 5
1408003296 : 6
1408003264 : 5
```

注意：此时x的地址与v的地址相同

现在x的地址和v的地址不一样了

参数传递

传递可变对象的引用

- 如果传递给函数的实参是可变序列，并且在函数内部使用下标或可变序列自身的方法增加、删除元素或修改元素时，实参也得到相应的修改。
- 例如：

```
>>> def modifyV(v):
...     v[0] += 1
...
>>> x = [7]
>>> modifyV(x)
>>> x
[8]
```

```
>>> def appendV(v, x):
...     v.append(x)
...
>>> a = [1, 2]
>>> appendV(a, 3)
>>> a
[1, 2, 3]
```

参数传递

案例

- 随机混排给定列表的元素值。

```
import random

def shuffle(lst):
    n = len(lst)
    for i in range(n):
        j = random.randint(0, n-1)
        lst[i], lst[j] = lst[j], lst[i]

def test_shuffle():
    v = [1,2,3,4,5]
    shuffle(v)
    print(v)

test_shuffle()
```

参数传递

位置参数

- 函数调用时，实参默认按位置顺序传递给对应的形参。
- 按位置传递的参数称之为位置参数。
- 实参和形参的顺序必须严格一致，并且实参和形参的数量必须相同。

```
>>> def demo(a,b,c):
...     print(a,b,c)
...
>>> demo(1,2,3)
1 2 3
>>> demo(2,3,1)
2 3 1
>>> demo(1,2,3,4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: demo() takes 3 positional arguments but 4 were given
```

参数传递

可选参数

- 在定义函数时，如果希望函数的一些参数是可选的，可以在声明函数时为这些参数指定默认值。
- 调用该函数时，如果没有传入对应的实参值，则函数使用声明时指定的默认参数值。

```
>>> def echo(message, times = 1):
...     print((message+'\n')*times)
...
>>> echo('hello', 2)
hello
hello

>>> echo('Bus is coming')
Bus is coming
```

参数传递

可选参数

- 需要注意的是，在定义带有默认值参数的函数时，任何一个默认值参数右边都不能再出现没有默认值的普通位置参数，否则会提示语法错误。

```
>>> def point(x=0,y=0,z):
...     print(x,y,z)
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```

参数传递

可选参数

注意

- 可以使用“函数名.__defaults__”随时查看函数所有默认值参数的当前值，其返回值为一个元组，其中的元素依次表示每个默认值参数的当前值。
- 多次调用函数并且不为默认值参数传递值时，默认值参数只在定义时进行一次解释和初始化，对于列表、字典这样可变类型的默认值参数，这一点可能会导致很严重的逻辑错误。
- 一般来说，要避免使用列表、字典、集合或其他可变序列作为函数参数默认值，对于上面的函数，更建议使用下面的写法。

```
>>> def demo(newitem, old_list=None):
...     if old_list is None:
...             old_list = []
...     old_list.append(newitem)
...     return old_list
```

参数传递

命名参数（或关键字参数）

- 函数调用时，也可以通过名称（关键字）指定传入的参数，称为命名参数，也称之为关键字参数。
- 使用关键字参数具有三个优点：
  - 参数按名称意义明确；
  - 传递的参数与顺序无关；
  - 如果有多个可选参数，则可以选择指定某个参数值。

```
>>> def demo(a, b, c=10):
...     print(a, b, c)
...
>>> demo(8, 9)
8 9 10
>>> demo(c=7, b=6, a=5)
5 6 7
```

参数传递

可变长度参数

- 主要有两种形式：在形式参数名前加1个*或2个**。
- ***parameter**用来接受多个位置参数并将其放在一个元组中。
- ****parameter**接受多个关键参数并存放到字典中。
- 这类参数必须位于形参列表的最后位置。

参数传递

可变长度参数：传给参数的数量是任意的

*parameter的用法

```
>>> def demo(*p):
    print(p)

>>> demo(1,2,3)
(1, 2, 3)
>>> demo(1,2)
(1, 2)
>>> demo(1,2,3,4,5,6,7)
(1, 2, 3, 4, 5, 6, 7)
```

参数传递

可变长度参数

**parameter的用法

```
>>> def demo(**p):
    for item in p.items():
        print(item)

>>> demo(x=1,y=2,z=3)
('y', 2)
('x', 1)
('z', 3)
```

参数传递

- 几种不同类型的参数可以混合使用，但是不建议这样做。

```
>>> def func_4(a, b, c=4, *aa, **bb):
    print(a,b,c)
    print(aa)
    print(bb)

>>> func_4(1,2,3,4,5,6,7,8,9,xx='1',yy='2',zz=3)
(1, 2, 3)
(4, 5, 6, 7, 8, 9)
{'yy': '2', 'xx': '1', 'zz': 3}
>>> func_4(1,2,3,4,5,6,7,xx='1',yy='2',zz=3)
(1, 2, 3)
(4, 5, 6, 7)
{'yy': '2', 'xx': '1', 'zz': 3}
```

参数传递

强制命名参数

- 在带星号的参数后面申明参数会导致强制命名参数。
  - 调用时必须显式使用命名参数传递值，因为按位置传递的参数默认收集为一个元组，传递给前面带星号的可变参数。
- 如果不需要带星的可变参数，只想使用强制命名参数，可以简单地使用一个星号。
  - 例如：**def my_func(*, a, b, c)**

函数的返回值

return语句和函数返回值

- 举例：编写函数，利用return语句返回函数值，求若干数中的最大值。

```
def themax(v1, v2, *values):
    values = (v1, v2) + values
    result = values[0];
    for i in range(1, len(values)):
        if result < values[i]:
            result = values[i]

    return result;


print(themax(2,4,3,1))
```

函数的返回值

包含多个return语句的函数

- return语句可以放置在函数中任何位置，当执行到第一个return语句时，程序返回到调用程序。
- 举例：
  - 先编制一个判断一个数是否为素数的函数，然后编写测试代码，判断并输出1~99中的素数。

```
def is_prime(x):
    if x<2: return False
    n = 2
    while n*n <= x:
        if x%n == 0: return False
        n += 1
    return True

for i in range(1, 100):
    if is_prime(i): print(i, end=' ')
```

函数的返回值

用return返回多个值

- 如果需要返回多个值，则可以返回一个元组。
- 举例：
  - 编写一个函数，返回若干个1-100内的随机数。

```
import random

def random_values(n):
    va = []
    for v in range(n):
        va.append(random.randint(1,100))
    return tuple(va)


for n in random_values(10):
    print(n, end=' ')
```

应用案例1

问题

- 编写函数计算任意位数的黑洞数。
- 黑洞数是指这样的整数：
  - 由这个数字每位上的数字组成的最大数减去每位数字组成的最小数仍然得到这个数自身。
- 例如：
  - **3位黑洞数是495，因为954-459=495**
  - **4位数字是6174，因为7641-1467=6174**

应用案例1

问题：找黑洞数

```
def black_holes(n):
    '''参数n表示数字的位数，例如n=3时返回495，n=4时返回6174'''
    #待测试数范围的起点和结束值
    start = 10**(n-1)
    end = 10**n
    #依次测试每个数
    for i in range(start, end):
        #由这几个数字组成的最大数和最小数
        big = ''.join(sorted(str(i),reverse=True))
        little = ''.join(reversed(big))
        big, little = map(int,(big, little))
        if big-little == i:
            print(i)

black_holes(3)
black_holes(4)
```

应用案例2

问题

- 编写函数，接收一个所有元素值都不相等的整数列表x和一个整数n，要求将值为n的元素作为支点，将列表中所有值小于n的元素全部放到n的前面，所有值大于n的元素放到n的后面。

```
def demo(x, n):
    t1 = [i for i in x if i<n]
    t2 = [i for i in x if i>n]
    return t1 + [n] + t2

print(demo((1,8,3,4,5,2), 4))
```

应用案例3

问题

- 简单“询问-回答”交互中的误答检测和提示。

```
def ask_ok(prompt, complaint='Yes or no, please!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        print(complaint)

ask_ok('Do you really want to quit?')
ask_ok('OK to overwrite the file?', 'Come on, only yes or no!')
```