函数

什么是函数

- 函数是一段具有特定功能的、可重用的语句组，用函数名表示，并通过函数名来完成对函数功能的调用

- 函数也可以看作是一段具有名字的子程序，可以在需要的地方调用执行，不需要在每个执行的地方重复编写这些语句

Python存在大量的内置函数，并且允许用户自定义函数

Python内置函数

- int(x), float(x), str(x), bool(x), list(x), tuple(x), set(x), dict(x)
- len(x), max(x), min(x)
- sorted(x [,key] [,reverse])
- print(x1,x2,...)

参数：零个，一个或多个（参数列表），必选/可选，任意长度

函数名：要使用的函数（执行函数功能）

返回值：函数可以有返回值，为一个或多个；也可以没有返回值

内容

- 函数的定义和调用
- 函数的参数及返回值
- 变量的作用域
- 函数的递归调用
- 函数式编程

函数的定义

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

函数定义示例

```
def greet_user(name):
    """向name问好"""
    print("Hello, ", name)
```

```
>>> help(greet_user)
Help on function greet_user in module __main__:

greet_user(name)
    向name问好
```

```
>>> greet_user(
              (name)
              向name问好
```

函数的调用

[返回值=]<函数名>([<参数列表>])

```
def greet_user(name):
    """向name问好"""
    print("Hello, ", name)
```

```
>>> greet_user('Mary')
Hello,  Mary
```

```
>>> x=greet_user('Mary')
Hello,  Mary
>>> x
>>> print(x)
None
```

函数的调用

[返回值=]<函数名>([<参数列表>])

```
def my_add(x,y):
    """x+y"""
    res = x+y
    return res
```

```
add1 = my_add(3,2)
print(add1)
5
```

```
my_add('abc', 'def')
'abcdef'
```

函数的调用

[返回值=]<函数名>([<参数列表>])

```
def multi_res():
    return 1, 2, 3
```

```
res1 = multi_res()
print(res1)
(1, 2, 3)
```

```
r1, r2, r3 = multi_res()
print(r1)
1
print(r2)
2
print(r3)
3
```

函数的前向引用

Python中，不允许在函数未定义之前对其进行引用或调用（称“前向引用”）

```
print(add(1,2))

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

```
Traceback (most recent call last):
  File "C:/python/Ch6_example2.py", line 1, in <module>
    print(add(1,2))
NameError: name 'add' is not defined
```

函数的前向引用

Python中，不允许在函数未定义之前对其进行引用或调用（称“前向引用”），但在函数中可以进行前向引用

```
def add(a,b):
    print("add")
    return add2(a,b)

def add2(a,b):
    print("add2")
    return a + b

print(add(1,2))
```

```
add
add2
3
```

函数的前向引用

语句对函数的调用，必须在函数定义之后，包括语句直接调用的函数中调用的其它函数，也必须在语句执行前进行定义

```
def add(a,b):
    print("add")
    return add2(a,b)

print(add(1,2))

def add2(a,b):
    print("add2")
    return a + b
```

```
add
Traceback (most recent call last):
  File "C:/python/Ch6_example4.py", line 5, in <module>
    print(add(1,2))
  File "C:/python/Ch6_example4.py", line 3, in add
    return add2(a,b)
NameError: name 'add2' is not defined
```

函数的前向引用

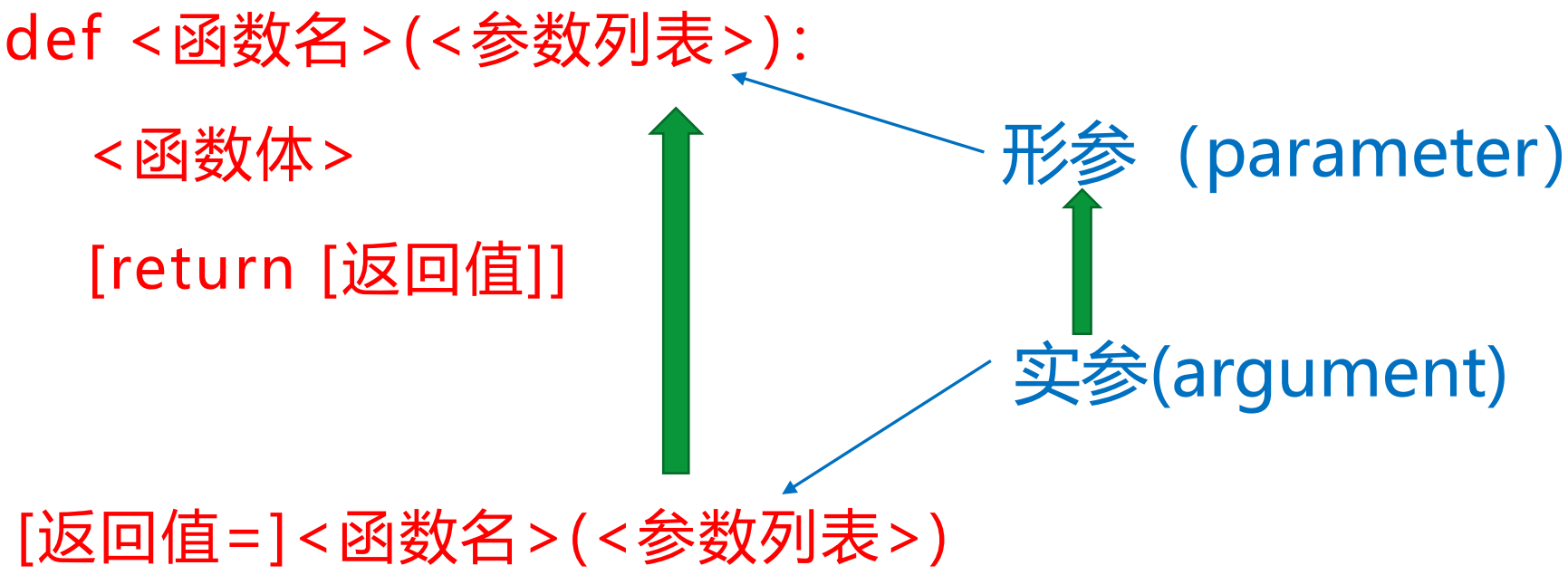
python代码从上往下执行，遇到函数会在内存中为其划分空间，并将函数作为字符串存入，但不会运行，直到函数被调用才会执行。所以在有外部语句调用函数前，定义的函数之间并无先后之分

建议无论是在语句中还是在函数中调用函数，都尽量不要使用前向引用

内容

- 函数的定义和调用
- 函数的参数及返回值
- 变量的作用域
- 函数的递归调用
- 函数式编程

函数的参数：形参和实参


def <函数名>(<参数列表>):
<函数体>
[return [返回值]]
形参（parameter）
实参(argument)
[返回值=]<函数名>(<参数列表>)

函数的参数：参数传递

函数调用时将实参传递给形参，本质上是变量赋值

```
def func1(n):
    print(n)


a = 10
func1(a)
```

n = a

```
def func1(n):
    n = 20
    print(n)


a = 10
func1(a)
```

20

参数传递的方式

对于有多个参数的函数，函数调用时，必须将每个实参一对一的传递给某一个形参。按照何种方式建立这种一对一的传递关系，即参数传递的方式

- 按位置传递
- 按名称传递

参数传递的方式：按位置传递

当一个函数有多个参数时，实参默认按照位置顺序传递给形参

```
def describe_pet(animal_type, pet_name):
    """显示宠物的信息"""
    print("\nI have a", animal_type)
    print("His/Her name is", pet_name)


describe_pet('cat', 'Garfield')
```

```
I have a cat
His/Her name is Garfield
```

参数传递的方式：按位置传递

当一个函数有多个参数时，实参默认按照位置顺序传递给形参

```
def describe_pet(animal_type, pet_name):
    """显示宠物的信息"""
    print("\nI have a", animal_type)
    print("His/Her name is", pet_name)


describe_pet('cat', 'Garfield')


describe_pet('dog', ‘Neumal')
```

```
I have a cat
His/Her name is Garfield

I have a dog
His/Her name is Neumal
```

参数传递的方式：按位置传递

按照位置传递时，实参的顺序必须和形参相一致

```
def describe_pet(animal_type, pet_name):
    """显示宠物的信息"""
    print("\nI have a", animal_type)
    print("His/Her name is", pet_name)


describe_pet('cat', 'Garfield')


describe_pet('Nermal', 'dog')
```

```
I have a cat
His/Her name is Garfield

I have a Neumal
His/Her name is dog
```

参数传递的方式：按名称传递

直接在实参中将形参名称和实参值关联起来

```
def describe_pet(animal_type, pet_name):
    """显示宠物的信息"""
    print("\nI have a", animal_type)
    print("His/Her name is", pet_name)

describe_pet(animal_type='cat', pet_name='Garfield')

describe_pet(pet_name='Neumal', animal_type='dog')
```

```
I have a cat
His/Her name is Garfield

I have a dog
His/Her name is Neumal
```

参数的默认值

定义函数时，可以给某个形参指定默认值

```
def describe_pet(pet_name,animal_type='dog')
```

参数默认值，在参数列表中指定

调用函数时，对于指定了默认值的形参：

- 如果给形参提供了实参，则使用指定的实参值
- 如果未给形参提供实参，则使用形参的默认值

参数的默认值

```
def describe_pet(pet_name,animal_type='dog'):
    """显示宠物的信息"""
    print("\nI have a", animal_type)
    print("His/Her name is", pet_name)


describe_pet(animal_type='cat', pet_name='Garfield')


describe_pet('Neumal')
```

```
I have a cat
His/Her name is Garfield

I have a dog
His/Her name is Neumal
```

有默认值参数的函数的定义及调用方式

def <函数名>(<必选参数1>, <必选参数2>...,<可选参数1>=<默认值1>[,<可选参数2>=<默认值2>]...[<可选参数n>=<默认值n>):

<函数体>

[return [返回值]]

[返回值=]<函数名>(<必选实参1><必选参数2>...[,<可选参数m>=<实参值m>]...)

```
sorted(list, key = None, reverse = False)
```

可变数量参数

Python函数中可以接受非固定数目的参数

- 通过*将接收到的非固定数目的参数存入元组
- 通过**将接收到的非固定数目的参数存入字典

可变数量参数：参数名前使用*

```
>>> def func1(*a):
        print(a)


>>> func1(1,2,3)                    #按位置传递参数
(1, 2, 3)

>>> func1('a','b',True,0)
('a', 'b', True, 0)
```

可变数量参数：参数名前使用**

```
>>> def func2(**p):
        for i in p.items():
            print(i)
        print(p)


>>> func2(a=1,b=2,c=3)    #按“名称”传递参数
('c', 3)
('b', 2)
('a', 1)
{'c': 3, 'b': 2, 'a': 1}
```

函数定义中参数的排列顺序

位置参数 >> *可变参数 >> 默认值参数 >> **可变参数

```
def func(a,*args,b=10,**kwargs):
    print(a)
    print(args)
    print(b)
    print(kwargs)
func("1st","2nd","3rd", c=10, d=20)
func("1st","2nd","3rd",b='B',c=10, d=20)
```

```
1st
('2nd', '3rd')
10
{'c': 10, 'd': 20}
1st
('2nd', '3rd')
B
{'c': 10, 'd': 20}
```

函数定义中参数的排列顺序

位置参数 >> *可变参数 >> 默认值参数 >> **可变参数

```
def func(a,*args,b=10,**kwargs):
    print(a)
    print(args)
    print(b)
    print(kwargs)
func("1st","2nd","3rd", c=10, b='B', d=20)
func("1st")
```

```
1st
('2nd', '3rd')
B
{'c': 10, 'd': 20}

1st
()
10
{}
```

函数定义中参数的排列顺序

位置参数 >> *可变参数 >> 默认值参数 >> **可变参数

```
def func(a, b=10,*args,**kwargs):
    print(a)
    print(b)
    print(args)
    print(kwargs)
func("1st","2nd","3rd", c=10, d=20)
func("1st","2nd","3rd",b='B',c=10, d=20)
```

```
1st
2nd
('3rd',)
{'c': 10, 'd': 20}
```

```
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in <module>
    func("1st","2nd","3rd",b='B',c=10, d=20)
TypeError: func() got multiple values for argument 'b'
```

参数的传递机制

根据实参类型的不同，参数的传递有两种方式：

- 值传递：实参为不可变数据类型
- 引用传递（地址传递）：实参为可变数据类型

参数传递的机制：值传递

值传递后，若形参的值发生改变，不会改变实参的值

```
def func2(a):
    print("实参传递值:",a)
    a += a
    print("形参的值:",a)


a = 5
print("实参的值:",a)
func2(a)
print("调用函数后实参的值:",a)
```

```
实参的值: 5
实参传递值: 5
形参的值: 10
调用函数后实参的值: 5
```

参数传递的机制：指针传递

指针传递后，若形参的值发生改变，实参的值也会一同改变

```
def func2(a):
    print("实参传递值:",a)
    a[0] = a[1]+a[2]
    print("形参的值:",a)

a = [1,2,3]
print("实参的值:",a)
func2(a)
print("调用函数后实参的值:",a)
```

```
实参的值: [1, 2, 3]
实参传递值: [1, 2, 3]
形参的值: [5, 2, 3]
调用函数后实参的值: [5, 2, 3]
```

列表参数传递：保证列表安全

将列表的全切片作为实参传递给函数

```
def func2(a):
    print("实参传递值:",a)
    a[0] = a[1]+a[2]
    print("形参的值:",a)


a = [1,2,3]
print("实参的值:",a)
func2(a[:])
print("调用函数后实参的值:",a)
```

```
实参的值: [1, 2, 3]
实参传递值: [1, 2, 3]
形参的值: [5, 2, 3]
调用函数后实参的值: [1, 2, 3]
```

return语句和函数的返回值

return语句的功能是结束函数的执行，并将“返回值”作为结果返回

return [返回值]

- 返回值类型可以是常量、变量或复杂的表达式
- return 语句作为函数的出口，可以在函数中多次出现。多个return语句的“返回值”可以不同。在哪个return语句结束函数的执行，函数的返回值就和哪个return语句里面的“返回值”相等

return语句和函数的返回值

- 定义函数时不需要声明函数的返回值类型
- 函数返回值类型与return语句返回的表达式类型一致
- 没有return语句时函数的返回值都为None，即返回空值
- 可以返回元组类型，类似返回多个值

return语句和函数的返回值示例

```
def func1():
    print("Hello World!")

def func2(x,y):
    if x>y:
        return x
    else:
        return y

def func3(x,y):
    return x+y, x-y, x*y, x/y

print("Func1:")
func1()
print("Func2:")
print("Max(4,6)=",func2(4,6))
print("Func3:")
print("Four arithmetic of (5,10)=", func3(5,10))
```

```
Func1:
Hello World!

Func2:
Max(4,6)= 6

Func3:
Four arithmetic of (5,10)= (15, -5, 50, 0.5)
```

内容

- 函数的定义和调用
- 函数的参数及返回值
- 变量的作用域
- 函数的递归调用
- 函数式编程

全局变量和局部变量

- 局部变量是指在函数内部定义的普通变量，仅在函数内部有效，函数执行结束，局部变量就会被删除
- 全局变量一般在函数之外定义，在程序执行的全程有效

Python标识符的作用域：LEGB

标识符的作用域即其声明在程序中可应用的范围

- L(local)：局部作用域，即函数中所定义变量的作用域
- E(enclosing)：嵌套作用域，即嵌套的父级函数的作用域，也就是包含该函数的上级函数的局部作用域
- G(global): 全局变量作用域，即模块作用域
- B(built-in): 系统内建变量的作用域

Python标识符的作用域：LEGB

```
x = int(2.9)
g_count = 0

def outer():
    o_count = 1

    def inner():
        i_count = 2
```

#B: 内建

#G: 全局

#E: 嵌套（inner）

#L：局部（outer）

#L: 局部（inner）

Python标识符的作用域：LEGB规则

```
a = 'global'
def outer():
    a = 'enclosed'
    print("in outer():",a)
    def inner():
        a = 'local'
        print("in inner():",a)
    inner()
    print("after inner():",a)
print("outer before:",a)
outer()
print("outer after:",a)
```

#G: 全局

#E: 嵌套

#L: 局部

#L > E > G

```
outer before: global
in outer(): enclosed
in inner(): local
after inner(): enclosed
outer after: global
```

Python标识符的作用域：LEGB规则

```
def len(in_var):
    print('called my len() function:')
    len_no_l = 0
    for i in in_var:
        if i != 'l':
            len_no_l += 1
    return l


lstr = len('Hello, World!')
print("Input variable is of length", lstr)
```

#G: 全局

```
called my len() function:
Input variable is of length 10
```

#G > B

Python标识符的作用域：LEGB规则

L → E → G →B

Python解析标识符的顺序依次为局部作用域、嵌套作用域、全局作用域和内置作用域。这就是所谓的LEGB法则

在局部变量中使用全局变量

在函数内部，如果不修改全局变量（该变量也没有被局部变量覆盖），只是读取全局变量的值，则可以正常使用全局变量。此规则对于嵌套作用域也同样适用

```
a = 1                          #全局作用域
def func():
    b = 10                     #嵌套作用域
    def func1():
        print(a)               #局部作用域
        print(b)               #局部作用域
    func1()
func()
```

```
1
10
```

在局部作用域中为全局变量赋值

在函数内部可以为全局变量或嵌套变量赋值

- 如果一个变量已在全局作用域中定义，在函数内需要为这个变量赋值，可以在函数内使用global关键字将其声明为全局变量
- 如果在函数内需要对已定义的嵌套变量赋值，可在函数内使用nonlocal将其声明为嵌套变量

在局部作用域定义新的全局变量

在函数内部可以定义新的全局变量或嵌套变量

- 如果一个变量在函数外没有定义，在函数内部同样也可以使用global关键字直接将一个变量定义为全局变量，或使用nonlocal关键字将其定义为嵌套变量。该函数执行后，将增加一个新的全局变量或嵌套变量

在局部作用域中为同名全局变量赋值

在函数内的任意位置，如果有为变量赋值的语句，则在整个函数内该变量都为局部变量，且在这条赋值语句之前不能有引用变量值的操作，否则会引起代码异常，除非在引用前使用global声明该变量为全局变量

```
x=3
def f():
    print(x)


f()
3
```

```
x=3
def f():
    print(x)
    x=5
    print(x)
```

```
f()
Traceback (most recent call last):
  File "<pyshell#85>", line 1, in <module>
    f()
  File "<pyshell#84>", line 2, in f
    print(x)
UnboundLocalError: local variable 'x' referenced before assignment
```

在局部作用域中为同名全局变量赋值

在函数内的任意位置，如果有为变量赋值的语句，则在整个函数内该变量都为局部变量，且在这条赋值语句之前不能有引用变量值的操作，否则会引起代码异常，除非在引用前使用global声明该变量为全局变量

```
x=3
def f():
    global x
    print(x)
    x=5
    print(x)
```

```
f()
3
5

print(x)
5
```

变量作用域转换示例

```
def func():
    def func1():
        nonlocal a          #a：嵌套
        a = 5
    def func2():
        global b            #b：全局
        b = 10
    a,c = -1, -2            #a,c：局部
    global d                #d：全局
    d = -3
    func1()
    func2()
    print("a=",a,"b=",b,"c=",c, "d=",d)
```

```
a,b,c = 1,2,3       #a,b,c：全局
func()
print("a=",a,"b=",b,"c=",c, "d=",d)
```

```
a= 5 b= 10 c= -2 d= -3
a= 1 b= 10 c= 3 d= -3
```

内容

- 函数的定义和调用
- 函数的参数及返回值
- 变量的作用域
- 函数的递归调用
- 函数式编程

函数的递归调用

函数直接或间接调用自身的情况叫递归调用。Python支持函数的递归调用

```
def fib(n):
    if n == 0:
        return 0
    if n == 1 or n == 2:
        return 1
    else:
        return fib(n-1)+fib(n-2)
```

```
import time
for i in range(51):
    start= time.time()
    print("fib({0})={1}".format(i,fib(i)), end='   ')
    end = time.time()
    print("Runtime:",end-start)
```

fib(0)=0 Runtime: 0.0311698913574218 75
fib(1)=1 Runtime: 0.0156302452087402 34
fib(2)=1 Runtime: 0.0
fib(3)=2 Runtime: 0.0
fib(4)=3 Runtime: 0.0156297683715820 3
fib(5)=5 Runtime: 0.0
fib(6)=8 Runtime: 0.0156211853027343 75
fib(7)=13 Runtime: 0.0
fib(8)=21 Runtime: 0.0
fib(9)=34 Runtime: 0.0156121253967285 16
fib(10)=55 Runtime: 0.0
fib(11)=89 Runtime: 0.0
fib(12)=144 Runtime: 0.0156214237213134 77
fib(13)=233 Runtime: 0.0
fib(14)=377 Runtime: 0.0156099796295166 02
fib(15)=610 Runtime: 0.0
fib(16)=987 Runtime: 0.0155978202819824 22
fib(17)=1597 Runtime: 0.0
fib(18)=2584 Runtime: 0.0156538486480712 9
fib(19)=4181 Runtime: 0.0
fib(20)=6765 Runtime: 0.0
fib(21)=10946 Runtime: 0.0156261920928955 08
fib(22)=17711 Runtime: 0.0156211853027343 75
fib(23)=28657 Runtime: 0.0156209468841552 73
fib(24)=46368 Runtime: 0.0156219005584716 8
fib(25)=75025 Runtime: 0.0312139987945556 64

fib(26)=121393 Runtime: 0.125049591064453 12
fib(27)=196418 Runtime: 0.125056982040405 27
fib(28)=317811 Runtime: 0.125005245208740 23
fib(29)=514229 Runtime: 0.250045776367187 5
fib(30)=832040 Runtime: 0.312426328659057 6
fib(31)=1346269 Runtime: 0.499834133148193 4
fib(32)=2178309 Runtime: 0.796660184860229 5
fib(33)=3524578 Runtime: 1.280987977981567 4
fib(34)=5702887 Runtime: 2.249550580978393 6
fib(35)=9227465 Runtime: 3.499188184738159
fib(36)=14930352 Runtime: 5.639337062835693
fib(37)=24157817 Runtime: 9.122957944869995
fib(38)=39088169 Runtime: 14.762172222137451
fib(39)=63245986 Runtime: 23.619548797607422
fib(40)=102334155 Runtime: 37.866204023361206
fib(41)=165580141 Runtime: 62.110527753829956
fib(42)=267914296 Runtime: 102.47608232498169
fib(43)=433494437 Runtime: 165.97538423538208
fib(44)=701408733 Runtime: 272.22408652305603
fib(45)=1134903170 Runtime: 443.3947858810425
fib(46)=1836311903 Runtime: 722.3663275241852
fib(47)=2971215073 Runtime: 1175.5923311710358
fib(48)=4807526976 Runtime: 1908.2281227111816
fib(49)=7778742049 Runtime: 3101.1473326683044
fib(50)=12586269025 Runtime: 5006.79790854454

函数的递归调用

```
def fib(n):
    a,b = 0,1
    for i in range(n):
        a,b = b, a+b
    return a
```

```
fib(31)=1346269   Runtime: 0.0156035423278808
fib(32)=2178309   Runtime: 0.0156216621398925
fib(33)=3524578   Runtime: 0.0156092643737792
fib(34)=5702887   Runtime: 0.0156221389770507
fib(35)=9227465   Runtime: 0.0
fib(36)=14930352   Runtime: 0.0
fib(37)=24157817   Runtime: 0.0156297683715820
fib(38)=39088169   Runtime: 0.0156221389770507
fib(39)=63245986   Runtime: 0.0156130790710449
fib(40)=102334155   Runtime: 0.0
fib(41)=165580141   Runtime: 0.0
fib(42)=267914296   Runtime: 0.0
fib(43)=433494437   Runtime: 0.0156354904174804
fib(44)=701408733   Runtime: 0.0156216621398925
fib(45)=1134903170   Runtime: 0.0156230926513671
fib(46)=1836311903   Runtime: 0.0
fib(47)=2971215073   Runtime: 0.0
fib(48)=4807526976   Runtime: 0.0
fib(49)=7778742049   Runtime: 0.0156207084655761
fib(50)=12586269025   Runtime: 0.0156254768371582
```

内容

- 函数的定义和调用
- 函数的参数及返回值
- 变量的作用域
- 函数的递归调用
- 函数式编程

函数式编程 (Functional Programming)

接受函数作为参数或者把函数作为返回结果的函数叫高阶函数（Higher-order function）。函数式编程就是指这种高度抽象的编程范式

sorted(list, key = None, reverse = False)

内部高阶函数示例：map(), reduce(), filter()

map(func, *iterables)

将函数func()依次作用于iterables的每个元素，返回结果

```
def func1(x):
    return x*x
```

```
>>> r = map(func1, [1, 2, 3, 4, 5, 6])
>>> print(list(r))
[1, 4, 9, 16, 25, 36]
```

内部高阶函数示例：map(), reduce(), filter()

map(func, *iterables)

```
def multiply(x, y):
    return x*y
```

```
a = [1, 2, 3]
b = [4, 5, 6]
r = map(multiply, a, b)
list(r)
[4, 10, 18]
```

内部高阶函数示例：map(), reduce(), filter()

reduce(func, iterable[, initial])

实现累积计算：将接收两个参数的函数func()作用于iterable，func每次接收两个元素，第一次的两个元素为iterable的前两个元素；其后的第一个元素为上一次func的执行结果，第二个元素为iterable的下一个元素

#reduce(f, [x1,x2,x3,x4]) = f(f(f(x1, x2), x3), x4)

```
def func2(x,y):
    return x*10 + y
```

```
>>> from functools import reduce
>>> r = reduce(func2, [1, 2, 3, 4, 5, 6])
>>> print(r)
123456
```

内部高阶函数示例：map(), reduce(), filter()

filter(func or None, iterable)

将函数func()依次作用于iterable的每个元素，根据返回值是True还是False保留或丢弃该元素

```
def func3(x):
    return x%2 == 1
```

```
>>> r = filter(func3, [1, 2, 3, 4, 5, 6, 15, 18])
>>> print(list(r))
[1, 3, 5, 15]
```

函数式编程示例

```
def func1(x):
    return x*x

def addmulti(x, y, f):
    return f(x)+f(y)
```

```
>>> r = addmulti(3,5,func1)
>>> print(r)
34
```

函数式编程示例

```
def foo():
    def bar():
        print('I am a bar')
    return bar
```

```
res = foo()
res
<function foo.<locals>.bar at 0x000001D31AB96830>
res()
I am a bar
```

```
foo()()
I am a bar
```

```
bar()
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    bar()
NameError: name 'bar' is not defined
```

函数式编程示例

```
def weight(g):
    def cal_mg(m):
        return m*g
    return cal_mg
```

```
w = weight(10)
G1 = w(100)
print(G1)
1000
```

```
w2 = weight(9.8)
G2 = w2(100)
print(G2)
980.0000000000001
```

匿名函数Lambda

Python支持匿名函数，即没有函数名的函数

lambda [arg1 [,arg2, …argN]]: <expression>

参数列表

函数体&返回值

```
def func1(x):
    return x*x
```

```
lambda x: x*x
```

```
func1 = lambda x: x*x
```

匿名函数在函数式编程中的应用

```
>>> r = map(lambda x:x*x, [1,2,3,4,5,6])
>>> print(list(r))
[1, 4, 9, 16, 25, 36]

>>> from functools import reduce
>>> r = reduce(lambda x,y: x*10+y, [1,2,3,4,5,6])
>>> print(r)
123456

>>> r = filter(lambda x: x%2 == 1, [1,2,3,4,5,6,15,18])
>>> print(list(r))
[1, 3, 5, 15]
```

匿名函数在函数式编程中的应用：按姓氏拼音排序

```
>>> students =[ ('20181213001', 'Li', 'M', 21, 'CS'),
                ('20181213002', 'Liu', 'F', 20, 'CS'),
                ('20181213003', 'Wang', 'F', 19, 'IS'),
                ('20181213004', 'Zhang', 'M', 20, 'IS'),
                ('20181213005', 'Chen', 'M', 19, 'IS') ]

>>> s = sorted(students, key=lambda student: student[1])
>>> for st in s: print(st)

('20181213005', 'Chen', 'M', 19, 'IS')
('20181213001', 'Li', 'M', 21, 'CS')
('20181213002', 'Liu', 'F', 20, 'CS')
('20181213003', 'Wang', 'F', 19, 'IS')
('20181213004', 'Zhang', 'M', 20, 'IS')
```

匿名函数在函数式编程中的应用

世界杯小组赛的32支参赛队分为八个小组，每组四队进行比赛。每支球队都必须和其他三支球队进行且只进行一场比赛，胜者得三 分，负者不得分，打平双方各得一分。小组排名规则的前3条如下：

a、积分高者排名靠前； b、小组中总净胜球（总进球数减去总失球数）高者排名靠前； c、小组中总进球数高者排名靠前。

匿名函数在函数式编程中的应用

```
>>> Teams = [ ('Iran', 2, 2, 4),('Morocco', 2, 4, 1),
              ('Portugal', 5, 4, 5),('Spain', 6, 5, 5)]
```

国家名　进球数　失球数　积分

```
>>> ts = sorted(Teams,key = lambda x :(x[3],x[1]-x[2],x[1]),reverse = True)

>>> for t in ts: print(t)

('Spain', 6, 5, 5)
('Portugal', 5, 4, 5)
('Iran', 2, 2, 4)
('Morocco', 2, 4, 1)
```

函数：总结

- 可以在函数定义的开头部分使用三引号增加注释，向用户提示函数说明
- 定义函数时不需要指定其形参类型，而是根据调用函数时传递的实参自动推定
- 在绝大多数情况下，在函数内部直接修改形参的值不会影响实参
- 当实参为可变数据类型时，在函数内部对形参的修改可能会影响实参

函数：总结

- 参数可以按位置传递，也可以按名称传递
- 定义函数时可以为形参设置默认值，默认值参数必须在位置参数之后
- 可以通过在形参名前加*或**的方式接收不定长参数，其中，加*的形参接收到的实参放置到元组中，加**的形参接收到的实参放置到字典中
- 定义函数时不需要指定其返回值的类型，而是由具体执行到的return语句确定返回值类型；若没有return、return无返回值或未执行到return语句，则返回None值

函数：总结

- Python解析标识符作用域的顺序为：LEGB
- 在函数内定义的普通变量只能在函数内部起作用，称为局部变量。当函数运行结束后，该函数的局部变量被自动删除
- 在函数内部可以通过global关键字声明或定义全局变量
- 函数可以调用自身（递归），也可以以函数为参数或返回值（高阶函数）
- lambda表达式可以用来创建只包含一个表达式的匿名函数