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,...)

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)
```

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)
```

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
```

函数的调用

[返回值=]<函数名>([<参数列表>])

```
def multi_res():
    return 1, 2, 3, 4
```

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(my_add(1, 2))

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

```
Traceback (most recent call last):
  File "C:/Users/DW/AppData/Local/Programs/Python/Python310/exp.ch5_1.py", line 1, in <module>
    print(my_add(1, 2))
NameError: name 'my_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 my_add(a, b):
    print('my_add is running...')
    return my_add_2(a, b)

def my_add_2(a, b):
    print('my_add_2 is running...')
    return a + b

print(my_add(1, 2))
```

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 my_add(a, b):
    print('my_add is running...')
    return my_add_2(a, b)

print(my_add(1, 2))

def my_add_2(a, b):
    print('my_add_2 i
    return a + b
```

```
my_add is running...
Traceback (most recent call last):
  File "C:/Users/DW/AppData/Local/Programs/Python/Python310/exp.ch5_3.py", line 5, in <module>
    print(my_add(1, 2))
  File "C:/Users/DW/AppData/Local/Programs/Python/Python310/exp.ch5_3.py", line 3, in my_add
    return my_add_2(a, b)
NameError: name 'my_add_2' is not defined. Did you mean: 'my_add'?
```

函数的前向引用

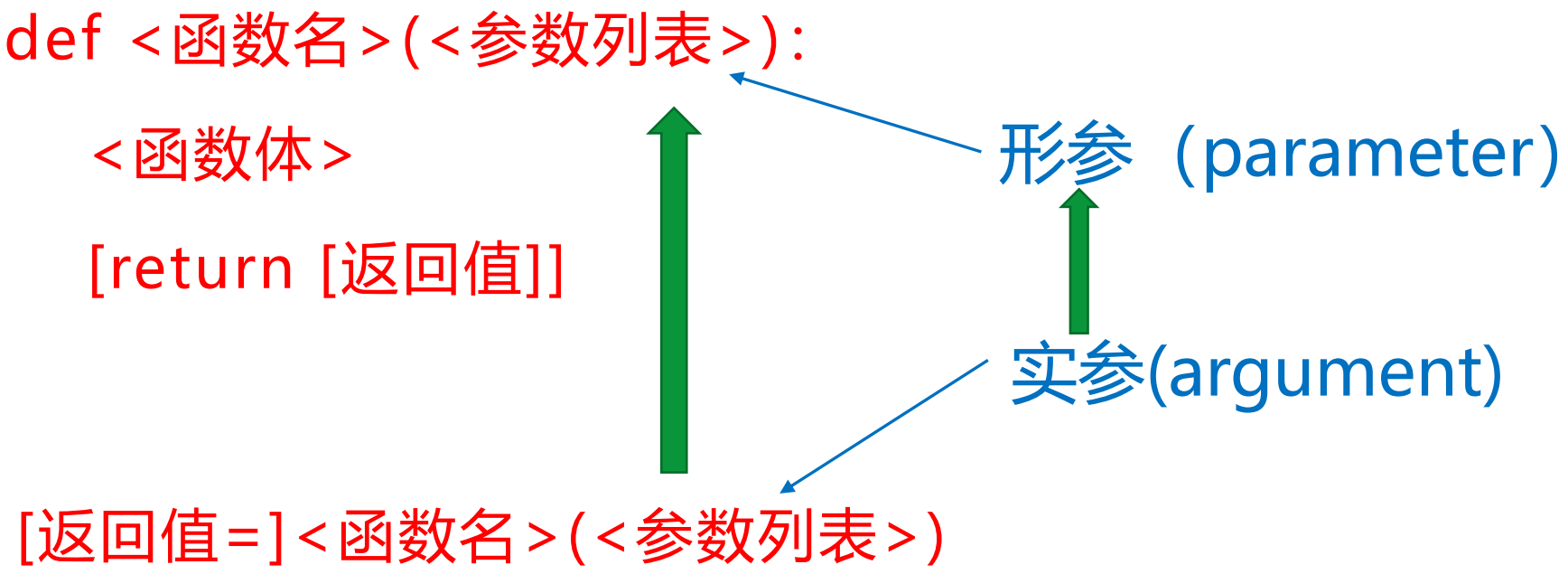
python代码从上往下执行，遇到函数会在内存中为其划分空间，并将函数作为字符串存入，但不会运行，直到函数被调用才会执行。所以在有外部语句调用函数前，定义的函数之间并无先后之分

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)
```

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')
```

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')
```

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')
```

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')
```

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>]...)

参数的默认值

函数的默认值仅仅在第一次定义时，被赋值一次

函数的默认值如果是可变数据类型，则可能出现不可预测的结果

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)
(1, 2, 3)
```

```
>>> def func2(len, *s):
...     print(f'length: {len}, s : {s}')

>>> func2 (5, 'a', 'b', 0, True, -1.5)
length: 5, s : ('a', 'b', 0, True, -1.5)
```

可变数量参数：参数名前使用**

```
>>> def func3(**p):
...     for item in p.items():
...         print(item)
...     print('p: ', p)
```

```
>>> func3 (a = 1, b = 2, c = 3)
('a', 1)
('b', 2)
('c', 3)
p:  {'a': 1, 'b': 2, 'c': 3}
```

函数定义中参数的排列顺序探讨

位置参数 >> 默认值参数 >> *可变参数 >> **可变参数

位置参数 >> *可变参数 >> 默认值参数 >> **可变参数

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 func1(a):
    print("实参传递值: ", a)
    a += a
    print("执行函数后形参的值: ", a)


a = 5
print("实参的值: ", a)
func1(a)
print("调用函数后实参的值: ", a)
```

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("执行函数后形参的值: ", a)


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

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("执行函数后形参的值: ", a)


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

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

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。此规则对于嵌套变量也同样适用

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

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("In:  ", "a =", a, "b =", b, "c =", c, "d =", d)
```

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

In: a = 5 b = 10 c = -2 d = -3

Out: 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)
```

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

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
```

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

```
>>> s1 = list(map(func1, [1, 2, 3, 4, 5]))
>>> s1
[1, 4, 9, 16, 25]
```

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> list(map(func2, a, b))
[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 func3(x, y):
    return x * 10 + y
```

```
>>> from functools import reduce
>>> reduce(func3, [1, 2, 3, 4, 5, 6])
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 func4(x):
    return x % 2 == 1
```

```
>>> s1 = list(filter(func4, [i * 3 for i in range(8)]))
>>> s1
[3, 9, 15, 21]
```

函数式编程示例

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


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

函数式编程示例

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

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

```
>>> s1 = list(map(lambda x: x * x, [1, 2, 3, 4, 5, 6]))
>>> print(s1)
[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

>>> s2 = list(filter (lambda x: x % 3 == 1, [ i *2 for i in range(10)]))
>>> print(s2)
[4, 10, 16]
```

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