

# Table of Contents

软件测试Python课程	1.1
函数	1.2
函数基础	1.2.1
变量进阶	1.2.2
函数进阶	1.2.3
匿名函数	1.2.4

传智播客-黑马程序员

# 软件测试Python课程

本阶段课程不仅可以帮助我们进入Python语言世界，同时也是后续UI自动化测试、接口自动化测试等课程阶段的语言基础。

Life is short, you need Python! -- 人生苦短，我用Python！

## 课程大纲

序号	章节	知识点
1	Python基础	1. 认识Python 2. Python环境搭建 3. PyCharm 4. 注释、变量、变量类型、输入输出、运算符
2	流程控制结构	1. 判断语句 2. 循环
3	数据序列	1. 字符串 2. 列表 3. 元组 4. 字典
4	函数	1. 函数基础 2. 变量进阶 3. 函数进阶 4. 匿名函数
5	面向对象	1. 面向对象编程介绍 2. 类和对象 3. 面向对象基础语法 4. 封装、继承、多态 5. 类属性和类方法
6	异常、模块、文件操作	1. 异常 2. 模块和包 3. 文件操作
7	UnitTest框架	1. UnitTest基本使用 2. UnitTest断言 3. 参数化 4. 生成HTML测试报告

## 课程目标

1. 掌握如何搭建Python开发环境；
2. 掌握Python基础语法, 具备基础的编程能力；
3. 建立编程思维以及面向对象程序设计思想；
4. 掌握如何通过UnitTest编写测试脚本，并生成HTML测试报告。

佐智播客-黑马程序员

# 函数

## 目标

1. 掌握函数的概念和作用
2. 掌握函数的定义及调用方法
3. 函数中参数的作用及用法
4. 函数返回值的作用及用法
5. 理解变量的引用
6. 理解可变和不可变类型
7. 掌握局部变量和全局变量
8. 掌握匿名函数的语法和应用场景

# 函数基础

## 目标

1. 掌握函数的概念和作用
2. 掌握函数的基本使用
3. 掌握函数的参数
4. 掌握函数的返回值
5. 掌握函数的嵌套调用

## 1. 函数的快速体验

### 1.1 函数介绍

- 所谓函数，就是把 具有独立功能的代码块 组织为一个小模块，在需要的时候 调用
- 函数的使用包含两个步骤：
  1. 定义函数 —— 封装 独立的功能
  2. 调用函数 —— 享受 封装 的成果
- 函数的作用，在开发程序时，使用函数可以提高编写的效率以及代码的 重用

### 1.2 快速体验

演练步骤：

1. 新建一个用来实现计算操作的文件 `calculate.py`
2. 添加一个实现加法操作的函数 `add(x, y)`
3. 新建另外一个文件，使用 `import` 导入并且调用函数

示例代码：

```
# calculate.py
def add(x, y):
    return x + y
```

```
# test_calculate.py
import calculate

result = calculate.add(1, 2)
print("result=", result) # result= 3
```

## 2. 函数基本使用

### 2.1 函数的定义

定义函数的格式如下：

```
def 函数名():
```

函数封装的代码

.....

1. `def` 是英文 `define` 的缩写
2. 函数名称 应该能够表达 函数封装代码 的功能，方便后续的调用
3. 函数名称 的命名应该 符合 标识符的命名规则
  - o 可以由 字母、下划线 和 数字 组成
  - o 不能以数字开头
  - o 不能与关键字重名

## 2.2 函数调用

调用函数很简单的，通过 `函数名()` 即可完成对函数的调用

## 2.3 第一个函数演练

需求：

1. 编写一个打招呼 `say_hello` 的函数，封装三行打招呼的代码
2. 在函数下方调用打招呼的代码

```
name = "小明"

# 解释器知道这里定义了一个函数
def say_hello():
    print("hello 1")
    print("hello 2")
    print("hello 3")

print(name)
# 只有在调用函数时，之前定义的函数才会被执行
# 函数执行完成之后，会重新回到之前的程序中，继续执行后续的代码
say_hello()

print(name)
```

注意：

- 定义好函数之后，只表示这个函数封装了一段代码而已
- 如果不主动调用函数，函数是不会主动执行的

## 思考

- 能否将 函数调用 放在 函数定义 的上方？
  - o 不能！
  - o 因为在 使用函数名 调用函数之前，必须要保证 `Python` 已经知道函数的存在
  - o 否则控制台会提示 `NameError: name 'say_hello' is not defined` (名称错误: **say\_hello** 这个名字没有被定义)

## 2.4 PyCharm 的调试工具

- **F8 Step Over** 可以单步执行代码，会把函数调用看作是一行代码直接执行

- **F7 Step Into** 可以单步执行代码，如果是函数，会进入函数内部

## 2.5 函数的文档注释

- 在开发中，如果希望给函数添加注释，应该在 定义函数 的下方，使用 连续的三对引号
- 在 连续的三对引号 之间编写对函数的说明文字
- 在 函数调用 位置，使用快捷键 **CTRL + Q** 可以查看函数的说明信息

注意：因为 函数体相对比较独立，函数定义的上方，应该和其他代码（包括注释）保留 两个空行

## 3. 函数的参数

演练需求

1. 开发一个 `sum_2_num` 的函数
2. 函数能够实现 两个数字的求和 功能

演练代码如下：

```
def sum_2_num():  
  
    num1 = 10  
    num2 = 20  
    result = num1 + num2  
  
    print("%d + %d = %d" % (num1, num2, result))  
  
sum_2_num()
```

思考一下存在什么问题

函数只能处理 固定数值 的相加

如何解决？

- 如果能够把需要计算的数字，在调用函数时，传递到函数内部就好了！

### 3.1 函数参数的使用

- 在函数名的后面的小括号内部填写 参数
- 多个参数之间使用 `,` 分隔

```
def sum_2_num(num1, num2):  
  
    result = num1 + num2  
  
    print("%d + %d = %d" % (num1, num2, result))  
  
sum_2_num(50, 20)
```

### 3.2 参数的作用

- 函数，把 具有独立功能的代码块 组织为一个小模块，在需要的时候 调用
- 函数的参数，增加函数的 通用性，针对 相同的数据处理逻辑，能够 适应更多的数据
  1. 在函数 内部，把参数当做 变量 使用，进行需要的数据处理

2. 函数调用时，按照函数定义的参数顺序，把 希望在函数内部处理的数据，通过参数 传递

### 3.3 形参和实参

- 形参：定义 函数时，小括号中的参数，是用来接收参数用的，在函数内部 作为变量使用
- 实参：调用 函数时，小括号中的参数，是用来把数据传递到 函数内部 用的

## 4. 函数的返回值

- 在程序开发中，有时候，会希望 一个函数执行结束后，告诉调用者一个结果，以便调用者针对具体的结果做后续的处理
- 返回值 是函数 完成工作后，最后 给调用者的 一个结果
- 在函数中使用 `return` 关键字可以返回结果
- 调用函数一方，可以 使用变量 来 接收 函数的返回结果

注意：`return` 表示返回，后续的代码都不会被执行

```
def sum_2_num(num1, num2):  
    """对两个数字的求和"""  
  
    return num1 + num2  
  
# 调用函数，并使用 result 变量接收计算结果  
result = sum_2_num(10, 20)  
  
print("计算结果是 %d" % result)
```

## 5. 函数的嵌套调用

- 一个函数里面 又调用 了 另外一个函数，这就是 函数嵌套调用
- 如果函数 `test2` 中，调用了另外一个函数 `test1`
  - 那么执行到调用 `test1` 函数时，会先把函数 `test1` 中的任务都执行完
  - 才会回到 `test2` 中调用函数 `test1` 的位置，继续执行后续的代码

```
def test1():  
  
    print("*" * 50)  
    print("test 1")  
    print("*" * 50)  
  
def test2():  
  
    print("-" * 50)  
    print("test 2")  
  
    test1()  
  
    print("-" * 50)  
  
test2()
```

函数嵌套的演练 —— 打印分隔线



体会一下工作中 需求是多变的

#### 需求 1

- 定义一个 `print_line` 函数能够打印 `*` 组成的一条分隔线

```
def print_line(char):  
    print("*" * 50)
```

#### 需求 2

- 定义一个函数能够打印 由任意字符组成 的分隔线

```
def print_line(char):  
    print(char * 50)
```

#### 需求 3

- 定义一个函数能够打印 任意重复次数 的分隔线

```
def print_line(char, times):  
    print(char * times)
```

#### 需求 4

- 定义一个函数能够打印 **5 行** 的分隔线，分隔线要求符合需求 3

提示：工作中针对需求的变化，应该冷静思考，不要轻易修改之前已经完成的，能够正常执行的函数！

```
def print_line(char, times):  
    print(char * times)  
  
def print_lines(char, times):  
    row = 0  
    while row < 5:  
        print_line(char, times)  
        row += 1
```

# 变量进阶（理解）

## 目标

1. 理解变量的引用
2. 理解可变和不可变类型
3. 掌握局部变量和全局变量

## 1. 变量的引用

- 变量 和 数据 都是保存在 内存 中的
- 在Python中 函数 的 参数传递 以及 返回值 都是靠 引用 传递的

### 1.1 引用的概念

在Python中

- 变量 和 数据 是分开存储的
- 数据 保存在内存中的一个位置
- 变量 中保存着数据在内存中的地址
- 变量 中 记录数据的地址，就叫做 引用
- 使用 `id()` 函数可以查看变量中保存数据所在的 内存地址

注意：如果变量已经被定义，当给一个变量赋值的时候，本质上是 修改了数据的引用

- 变量 不再 对之前的数据引用
- 变量 改为 对新赋值的数据引用

### 1.2 变量引用的示例

在 Python 中，变量的名字类似于 便签纸 贴在 数据 上

代码	描述	图示
<code>a = 1</code>	定义一个整数变量a，并且赋值为1	
<code>a = 2</code>	将变量a赋值为2	

`b = a`      定义一个整数变量**b**，并且将变量**a**的值赋值给**b**



## 1.3 函数的参数和返回值的传递

在Python中，函数的 实参/返回值 都是是靠 引用 来传递来的

```
def test(num):  
  
    print("-" * 50)  
    print("%d 在函数内的内存地址是 %x" % (num, id(num)))  
  
    result = 100  
  
    print("返回值 %d 在内存中的地址是 %x" % (result, id(result)))  
    print("-" * 50)  
  
    return result  
  
a = 10  
print("调用函数前 内存地址是 %x" % id(a))  
  
r = test(a)  
  
print("调用函数后 实参内存地址是 %x" % id(a))  
print("调用函数后 返回值内存地址是 %x" % id(r))
```

## 2. 可变和不可变类型

- 不可变类型，内存中的数据不允许被修改：
  - 数字类型 `int` , `bool` , `float` , `complex` , `long(2.x)`
  - 字符串 `str`
  - 元组 `tuple`
- 可变类型，内存中的数据可以被修改：
  - 列表 `list`
  - 字典 `dict`

```
a = 1  
a = "hello"  
a = [1, 2, 3]  
a = [3, 2, 1]
```

```
demo_list = [1, 2, 3]  
  
print("定义列表后的内存地址 %d" % id(demo_list))  
  
demo_list.append(999)  
demo_list.pop(0)  
demo_list.remove(2)  
demo_list[0] = 10
```

```
print("修改数据后的内存地址 %d" % id(demo_list))

demo_dict = {"name": "小明"}

print("定义字典后的内存地址 %d" % id(demo_dict))

demo_dict["age"] = 18
demo_dict.pop("name")
demo_dict["name"] = "老王"

print("修改数据后的内存地址 %d" % id(demo_dict))
```

注意：字典的 `key` 只能使用不可变类型的数据

注意

1. 可变类型的数据变化，是通过 方法 来实现的
2. 如果给一个可变类型的变量，赋值了一个新的数据，引用会修改
  - 变量 不再 对之前的数据引用
  - 变量 改为 对新赋值的数据引用

## 哈希 (hash)

- Python 中内置有一个名字叫做 `hash(o)` 的函数
  - 接收一个 不可变类型 的数据作为 参数
  - 返回 结果是一个 整数
- 哈希 是一种 算法，其作用就是提取数据的 特征码（指纹）
  - 相同的内容 得到 相同的结果
  - 不同的内容 得到 不同的结果
- 在 Python 中，设置字典的 键值对 时，会首先对 `key` 进行 `hash` 已决定如何在内存中保存字典的数据，以便 后续 对字典的操作：增、删、改、查
  - 键值对的 `key` 必须是不可变类型数据
  - 键值对的 `value` 可以是任意类型的数据

## 3. 局部变量和全局变量

- 局部变量 是在 函数内部 定义的变量，只能在函数内部使用
- 全局变量 是在 函数外部定义 的变量（没有定义在某一个函数内），所有函数 内部 都可以使用这个变量

提示：在其他的开发语言中，大多 不推荐使用全局变量 —— 可变范围太大，导致程序不好维护！

### 3.1 局部变量

- 局部变量 是在 函数内部 定义的变量，只能在函数内部使用
- 函数执行结束后，函数内部的局部变量，会被系统回收
- 不同的函数，可以定义相同的名字的局部变量，但是 彼此之间 不会产生影响

### 局部变量的作用

- 在函数内部使用，临时 保存 函数内部需要使用的数据

```
def demo1():
    num = 10
```

```

print(num)
num = 20
print("修改后 %d" % num)

def demo2():
    num = 100
    print(num)

demo1()
demo2()

print("over")

```

## 局部变量的生命周期

- 所谓 生命周期 就是变量从 被创建 到 被系统回收 的过程
- 局部变量 在 函数执行时 才会被创建
- 函数执行结束后 局部变量 被系统回收
- 局部变量在生命周期 内，可以用来存储 函数内部临时使用到的数据

## 3.2 全局变量

- 全局变量 是在 函数外部定义 的变量，所有函数内部都可以使用这个变量

```

# 定义一个全局变量
num = 10

def demo1():
    print(num)

def demo2():
    print(num)

demo1()
demo2()

print("over")

```

注意：函数执行时，需要处理变量时会：

1. 首先 查找 函数内部 是否存在 指定名称 的局部变量，如果有，直接使用
2. 如果没有，查找 函数外部 是否存在 指定名称 的全局变量，如果有，直接使用
3. 如果还没有，程序报错！

### 1) 函数不能直接修改 全局变量的引用

- 全局变量 是在 函数外部定义 的变量（没有定义在某一个函数内），所有函数 内部 都可以使用这个变量

提示：在其他的开发语言中，大多 不推荐使用全局变量 —— 可变范围太大，导致程序不好维护！

- 在函数内部，可以 通过全局变量的引用获取对应的数据
- 但是，不允许直接修改全局变量的引用 —— 使用赋值语句修改全局变量的值

```

num = 10

def demo1():

```

```

print("demo1" + "-" * 50)
# 只是定义了一个局部变量，不会修改到全局变量，只是变量名相同而已
num = 100
print(num)

def demo2():
    print("demo2" + "-" * 50)
    print(num)

demo1()
demo2()

print("over")

```

注意：只是在函数内部定义了一个局部变量而已，只是变量名相同 —— 在函数内部不能直接修改全局变量的值

## 2) 在函数内部修改全局变量的值

- 如果在函数中需要修改全局变量，需要使用 `global` 进行声明

```

num = 10

def demo1():
    print("demo1" + "-" * 50)
    # global 关键字，告诉 Python 解释器 num 是一个全局变量
    global num
    # 只是定义了一个局部变量，不会修改到全局变量，只是变量名相同而已
    num = 100
    print(num)

def demo2():
    print("demo2" + "-" * 50)
    print(num)

demo1()
demo2()

print("over")

```

## 3) 全局变量定义的位置

- 为了保证所有的函数都能够正确使用到全局变量，应该 将全局变量定义在其他函数的上方

```

a = 10

def demo():
    print("%d" % a)
    print("%d" % b)
    print("%d" % c)

b = 20
demo()
c = 30

```

注意

- 由于全局变量 `c`，是在调用函数之后，才定义的，在执行函数时，变量还没有定义，所以程序会报错！

#### 4) 全局变量命名的建议

- 为了避免局部变量和全局变量出现混淆，在定义全局变量时，有些公司会有一些开发要求，例如：
- 全局变量名前应该增加 `g_` 或者 `gl_` 的前缀

提示：具体的要求格式，各公司要求可能会有些差异

佐智播客-黑马程序员

# 函数进阶

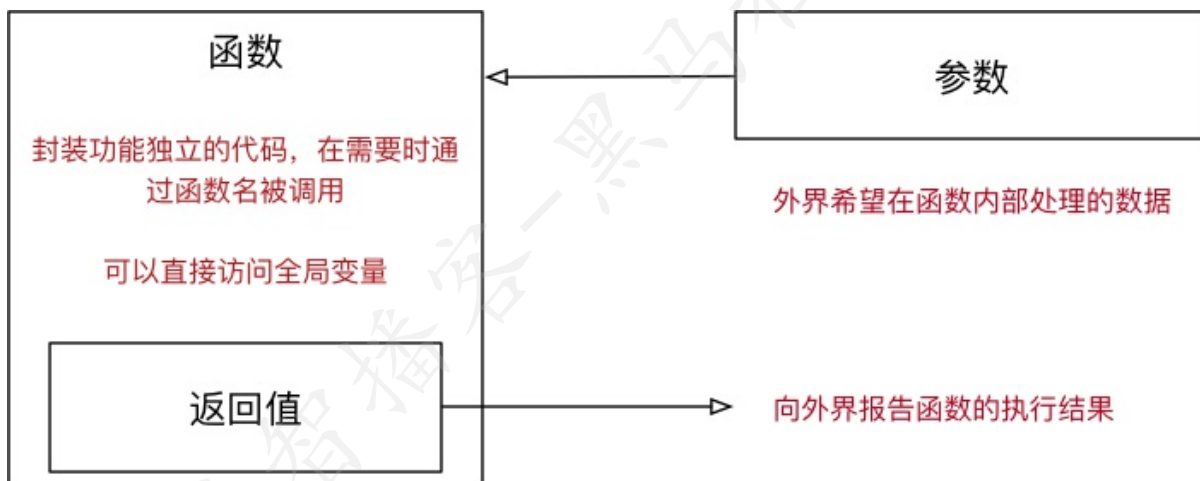
## 目标

1. 函数参数和返回值的作用
2. 函数的返回值 进阶
3. 函数的参数 进阶

## 1. 函数参数和返回值的作用

函数根据 有没有参数 以及 有没有返回值，可以 相互组合，一共有 **4** 种 组合形式

1. 无参数，无返回值
2. 无参数，有返回值
3. 有参数，无返回值
4. 有参数，有返回值



定义函数时，是否接收参数，或者是否返回结果，是根据 实际的功能需求 来决定的！

1. 如果函数 内部处理的数据不确定，就可以将外界的数据以参数传递到函数内部
2. 如果希望一个函数 执行完成后，向外界汇报执行结果，就可以增加函数的返回值

### 1.1 无参数，无返回值

此类函数，不接收参数，也没有返回值，应用场景如下：

1. 只是单纯地做一件事情，例如 显示菜单
2. 在函数内部 针对全局变量进行操作，例如：新建名片，最终结果 记录在全局变量 中

注意：

- 如果全局变量的数据类型是一个 可变类型，在函数内部可以使用 方法 修改全局变量的内容 —— 变量的引用 不会改变
- 在函数内部，使用赋值语句 才会 修改变量的引用



## 1.2 无参数，有返回值

此类函数，不接收参数，但是有返回值，应用场景如下：

- 采集数据，例如 温度计，返回结果就是当前的温度，而不需要传递任何的参数

## 1.3 有参数，无返回值

此类函数，接收参数，没有返回值，应用场景如下：

- 函数内部的代码保持不变，针对 不同的参数 处理 不同的数据
- 例如 名片管理系统 针对 找到的名片 做 修改、删除 操作

## 1.4 有参数，有返回值

此类函数，接收参数，同时有返回值，应用场景如下：

- 函数内部的代码保持不变，针对 不同的参数 处理 不同的数据，并且 返回期望的处理结果
- 例如 名片管理系统 使用 字典默认值 和 提示信息 提示用户输入内容
  - 如果输入，返回输入内容
  - 如果没有输入，返回字典默认值

## 2. 函数的返回值 进阶

- 在程序开发中，有时候，会希望 一个函数执行结束后，告诉调用者一个结果，以便调用者针对具体的结果做后续的处理
- 返回值 是函数 完成工作后，最后 给调用者的 一个结果
- 在函数中使用 `return` 关键字可以返回结果
- 调用函数一方，可以 使用变量 来 接收 函数的返回结果

问题：一个函数执行后能否返回多个结果？

### 示例 —— 温度和湿度测量

- 假设要开发一个函数能够同时返回当前的温度和湿度
- 先完成返回温度的功能如下：

```
def measure():
    """返回当前的温度"""
    print("开始测量...")
    temp = 39
    print("测量结束...")

    return temp

result = measure()
print(result)
```

- 在利用 元组 在返回温度的同时，也能够返回 湿度
- 改造如下：

```
def measure():
    """返回当前的温度"""
```

```
print("开始测量...")
temp = 39
wetness = 10
print("测量结束...")

return (temp, wetness)
```

提示：如果一个函数返回的是元组，括号可以省略

技巧

- 在 Python 中，可以将一个元组使用赋值语句同时赋值给多个变量
- 注意：变量的数量需要和元组中的元素数量保持一致

```
result = temp, wetness = measure()
```

## 面试题 —— 交换两个数字

题目要求

1. 有两个整数变量 `a = 6` , `b = 100`
2. 不使用其他变量，交换两个变量的值

### 解法 1 —— 使用其他变量

```
# 解法 1 - 使用临时变量
c = b
b = a
a = c
```

### 解法 2 —— 不使用临时变量

```
# 解法 2 - 不使用临时变量
a = a + b
b = a - b
a = a - b
```

### 解法 3 —— Python 专有，利用元组

```
a, b = b, a
```

## 3. 函数的参数 进阶

### 3.1. 不可变和可变的参数

问题 1：在函数内部，针对参数使用赋值语句，会不会影响调用函数时传递的实参变量？—— 不会！

- 无论传递的参数是可变还是不可变
  - 只要针对参数使用赋值语句，会在函数内部修改局部变量的引用，不会影响到外部变量的引用

```
def demo(num, num_list):
```

```

print("函数内部")

# 赋值语句
num = 200
num_list = [1, 2, 3]

print(num)
print(num_list)

print("函数代码完成")

gl_num = 99
gl_list = [4, 5, 6]
demo(gl_num, gl_list)
print(gl_num)
print(gl_list)

```

问题 2: 如果传递的参数是 可变类型, 在函数内部, 使用 方法 修改了数据的内容, 同样会影响到外部的数据

```

def mutable(num_list):
    # num_list = [1, 2, 3]
    num_list.extend([1, 2, 3])
    print(num_list)

gl_list = [6, 7, 8]
mutable(gl_list)
print(gl_list)

```

## 面试题 —— +=

- 在 python 中, 列表变量调用 += 本质上是在执行列表变量的 extend 方法, 不会修改变量的引用

```

def demo(num, num_list):
    print("函数内部代码")
    # num = num + num
    num += num
    # num_list.extend(num_list) 由于是调用方法, 所以不会修改变量的引用
    # 函数执行结束后, 外部数据同样会发生变化
    num_list += num_list

    print(num)
    print(num_list)
    print("函数代码完成")

gl_num = 9
gl_list = [1, 2, 3]
demo(gl_num, gl_list)
print(gl_num)
print(gl_list)

```

## 3.2 缺省参数

- 定义函数时, 可以给 某个参数 指定一个默认值, 具有默认值的参数就叫做 缺省参数
- 调用函数时, 如果没有传入 缺省参数 的值, 则在函数内部使用定义函数时指定的 参数默认值
- 函数的缺省参数, 将常见的值设置为参数的缺省值, 从而 简化函数的调用
- 例如: 对列表排序的方法

```
gl_num_list = [6, 3, 9]

# 默认就是升序排序, 因为这种应用需求更多
gl_num_list.sort()
print(gl_num_list)

# 只有当需要降序排序时, 才需要传递 `reverse` 参数
gl_num_list.sort(reverse=True)
print(gl_num_list)
```

## 指定函数的缺省参数

- 在参数后使用赋值语句, 可以指定参数的缺省值

```
def print_info(name, gender=True):

    gender_text = "男生"
    if not gender:
        gender_text = "女生"

    print("%s 是 %s" % (name, gender_text))
```

提示

1. 缺省参数, 需要使用 最常见的值 作为默认值!
2. 如果一个参数的值 不能确定, 则不应该设置默认值, 具体的数值在调用函数时, 由外界传递!

## 缺省参数的注意事项

### 1) 缺省参数的定义位置

- 必须保证 带有默认值的缺省参数 在参数列表末尾
- 所以, 以下定义是错误的!

```
def print_info(name, gender=True, title):
```

### 2) 调用带有多个缺省参数的函数

- 在 调用函数时, 如果有 多个缺省参数, 需要指定参数名, 这样解释器才能够知道参数的对应关系!

```
def print_info(name, title="", gender=True):
    """
    :param title: 职位
    :param name: 班上同学的姓名
    :param gender: True 男生 False 女生
    """
    gender_text = "男生"

    if not gender:
        gender_text = "女生"

    print("%s%s 是 %s" % (title, name, gender_text))

# 提示: 在指定缺省参数的默认值时, 应该使用最常见的值作为默认值!
print_info("小明")
print_info("老王", title="班长")
print_info("小美", gender=False)
```

### 3.3 多值参数（知道）

#### 定义支持多值参数的函数

- 有时可能需要一个函数能够处理的参数个数是不确定的，这个时候，就可以使用多值参数
- python 中有两种多值参数：
  - 参数名前增加一个 \* 可以接收元组
  - 参数名前增加两个 \* 可以接收字典
- 一般在给多值参数命名时，习惯使用以下两个名字
  - \*args —— 存放元组参数，前面有一个 \*
  - \*\*kwargs —— 存放字典参数，前面有两个 \*
- args 是 arguments 的缩写，有变量的含义
- kw 是 keyword 的缩写，kwargs 可以记忆键值对参数

```
def demo(num, *args, **kwargs):  
    print(num)  
    print(args)  
    print(kwargs)  
  
demo(1, 2, 3, 4, 5, name="小明", age=18, gender=True)
```

提示：多值参数的应用会经常出现在网络上一些大牛开发的框架中，知道多值参数，有利于我们能够读懂大牛的代码

#### 多值参数案例 —— 计算任意多个数字的和

需求

1. 定义一个函数 sum\_numbers，可以接收的任意多个整数
2. 功能要求：将传递的所有数字累加并且返回累加结果

```
def sum_numbers(*args):  
    num = 0  
    # 遍历 args 元组顺序求和  
    for n in args:  
        num += n  
  
    return num  
  
print(sum_numbers(1, 2, 3))
```

#### 元组和字典的拆包（知道）

- 在调用带有多值参数的函数时，如果希望：
  - 将一个元组变量，直接传递给 args
  - 将一个字典变量，直接传递给 kwargs
- 就可以使用拆包，简化参数的传递，拆包的方式是：
  - 在元组变量前，增加一个 \*
  - 在字典变量前，增加两个 \*

```
def demo(*args, **kwargs):  
    print(args)  
    print(kwargs)  
  
# 需要将一个元组变量/字典变量传递给函数对应的参数  
gl_nums = (1, 2, 3)  
gl_xiaoming = {"name": "小明", "age": 18}  
  
# 会把 num_tuple 和 xiaoming 作为元组传递个 args  
# demo(gl_nums, gl_xiaoming)  
demo(*gl_nums, **gl_xiaoming)
```

# 匿名函数

## 目标

1. 掌握匿名函数的语法和应用场景

## 1. 匿名函数介绍

- 用lambda关键词能创建小型匿名函数
- 这种函数得名于省略了用def声明函数的标准步骤
- Lambda函数能接收任何数量的参数，但只能返回一个表达式的值

语法格式：

```
lambda [arg1 [,arg2,...,argn]]:expression
```

示例：

```
sum = lambda arg1, arg2: arg1 + arg2

# 调用sum函数
print("result=", sum(10, 20)) # result= 30
print("result=", sum(20, 20)) # result= 40
```

## 2. 应用场景

匿名函数主要用于临时调用一次的场景，更多的是将匿名函数作为其他函数的参数来使用

### 2.1 自己定义函数

```
def cal(x, y, opt):
    print("x=", x)
    print("y=", y)
    result = opt(x, y)
    print("result=", result)

cal(1, 2, lambda a, b: a + b)

# 打印结果
# x= 1
# y= 2
# result= 3
```

### 2.2 作为内置函数的参数

已知用户列表数据：

```
user_list = [  
    {"name": "zhangsan", "age": 18},  
    {"name": "lisi", "age": 19},  
    {"name": "wangwu", "age": 17}  
]
```

需求：按照用户姓名对用户列表数据进行排序

示例代码：

```
user_list = [  
    {"name": "zhangsan", "age": 18},  
    {"name": "lisi", "age": 19},  
    {"name": "wangwu", "age": 17}  
]  
  
# 按照name进行排序  
user_list.sort(key=lambda x: x["name"])  
  
print(user_list)  
# [{'name': 'lisi', 'age': 19}, {'name': 'wangwu', 'age': 17}, {'name': 'zhangsan', 'age': 18}]
```