函数基础

本章主要是介绍函数的核心概念与入门知识。

学习目标：

1. 理解函数的作用
2. 掌握函数的定义与调用
3. 理解形参与实参
4. 掌握函数的嵌套使用
5. 理解局部变量与全局变量

函数入门

假如现在我们需要计算3个不同半径的圆的面积。圆的面积公式：面积 = π × 半径²

```python
# 计算半径为2的圆的面积
radius1 = 2
area1 = 3.14159 * radius1 ** 2
print(f"半径为{radius1}的圆面积：{area1:.2f}")

# 计算半径为5的圆的面积
radius2 = 5
area2 = 3.14159 * radius2 ** 2
print(f"半径为{radius2}的圆面积：{area2:.2f}")

# 计算半径为8的圆的面积
radius3 = 8
area3 = 3.14159 * radius3 ** 2
print(f"半径为{radius3}的圆面积：{area3:.2f}")
```

我们不难发现：

1. 在上面的代码中有大量重复的代码
2. 而且维护起来较为困难，比如若需修改 π 的值（如改为 3.14），需逐个修改 3 处代码

在生活中其实还有很多类似的场景。那么如何解决以上的问题呢？ **使用函数**

函数的概念

什么是函数?

函数是一个**被命名的**、独立的、**完成特定功能**的代码片段，其可能给调用它的程序一个**返回值**

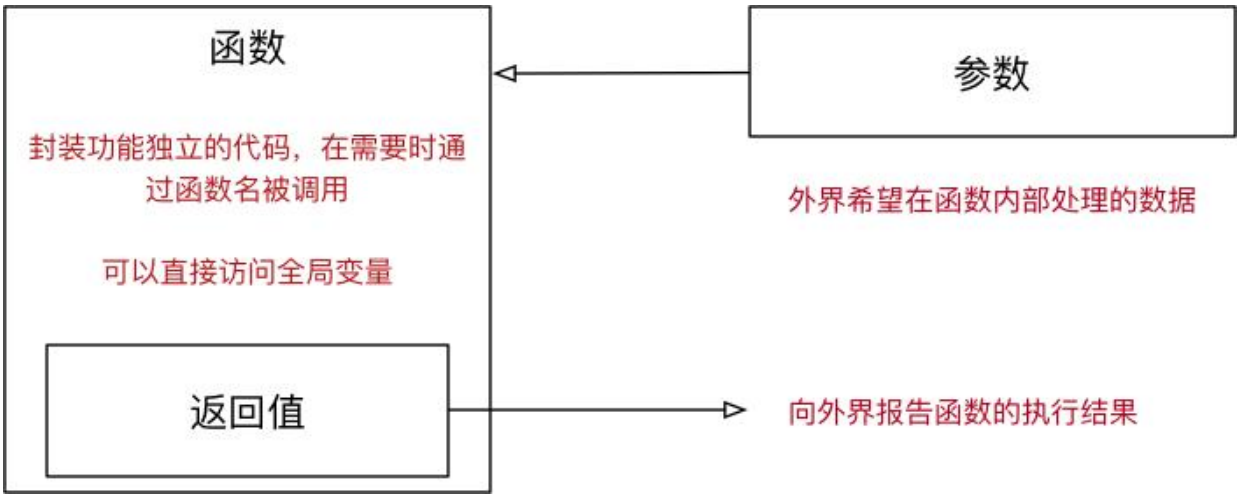
通俗的说，函数就是把一段可以实现某种特定功能的代码封装起来，想要使用这个功能就可以直接调用函数

- 被命名的： 在Python中，大部分函数都是有名函数，都有一个自己的名字
- 完成特定功能的代码段：函数的功能要专一，专门为了完成某个功能而定义
- 返回值：当函数执行完毕后，其可能会返回一个值给函数的调用处

函数的主要作用：**代码复用**。

一个函数中有几个比较重要的概念，我们先来了解一下：

- 函数
- 参数
- 返回值



函数就类似于数学中的方程，传入参数 → 完成特定的功能 → 获取计算的结果。

$$圆的面积方程：y = \pi r^2$$

对于上面这个面积方程，我们只需要传入参数半径r，然后通过方程计算，就可以获取到计算的结果。对于Python中的函数也是一样的。

如果把函数比作一个机器，那么参数就是源材料，返回值就是最终产品，同时根据参数的不同最终产品也会有不同

补充说明：函数也被称为方法。

函数的定义

定义函数的语法

```
def 函数名(形式参数1, 形式参数2...):
    函数体
    return 具体返回的值
```

格式解释：

- def：固定写法，define，定义的意思
- 函数名：类似于变量名，需要符合标识符的命名规则
- 形参列表：表示调用函数，应该传入什么内容
- 函数体：具体的逻辑代码，用于完成特定的功能
- return：结束函数的执行，并且把函数的执行结果进行返回

函数的调用语法

```
函数名(参数)
```

举个例子：现在需要计算3个不同半径的圆的面积

```
# 1. 定义函数
def calculate_area(r):
    area = 3.14159 * r * r
    return area

# 2. 调用函数
area1 = calculate_area(2)            # 计算半径为2的圆的面积
area2 = calculate_area(5)            # 计算半径为2的圆的面积
area3 = calculate_area(8)            # 计算半径为2的圆的面积

# 打印数据
print(f"半径为2的圆的面积:{area1}")
print(f"半径为5的圆的面积:{area2}")
print(f"半径为8的圆的面积:{area3}")
```

注意事项：

1. **函数必须要先定义，再使用**
2. 不同的需求，参数列表不一样，函数的参数列表可有可无，但是括号不能省略

3 调用函数，获取到结果，可以使用变量接收，也可以直接打印结果(本质就是把结果直接作为参数传递给了别的函数)

练一练：

1 定义一个函数，实现求两个数字之和的功能，并且调用。

2 定义一个函数，实现求学生成绩等级的功能，并且求出字典中各个学生的等级

- A：90~100
- B：80~89
- C：70~79
- D：小于70

```python
# 定义字典
dict1 = {"zs":88,"ls":100,"ww":85,'zl':62}

# 定义函数
def score_level(score):
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 70:
        return "C"
    else:
        return "D"

# 使用字典推导式(在value表达式中调用函数)
dict2 = {key:score_level(value) for key,value in dict1.items()}
print(dict2)
```

函数的说明文档

思考：定义一个函数后，程序员如何书写程序能够快速提示这个函数的作用？

回答：注释

思考：如果代码多，我们是不是需要在很多代码中找到这个函数定义的位置才能看到注释？如果想更方便的查看函数的作用怎么办？ 答：函数的说明文档（函数的说明文档也叫函数的文档说明）。

函数的说明文档：函数的说明文档也叫文档说明，就是解释函数的参数，作用，返回值的。

格式：在函数内的第一行，用 三引号的写法来实现即可，可以是双引号(推荐)，也可以单引号.

如何查看说明文档：

- help(函数名)
- 光标放在函数上，按ctrl + Q (更常用)

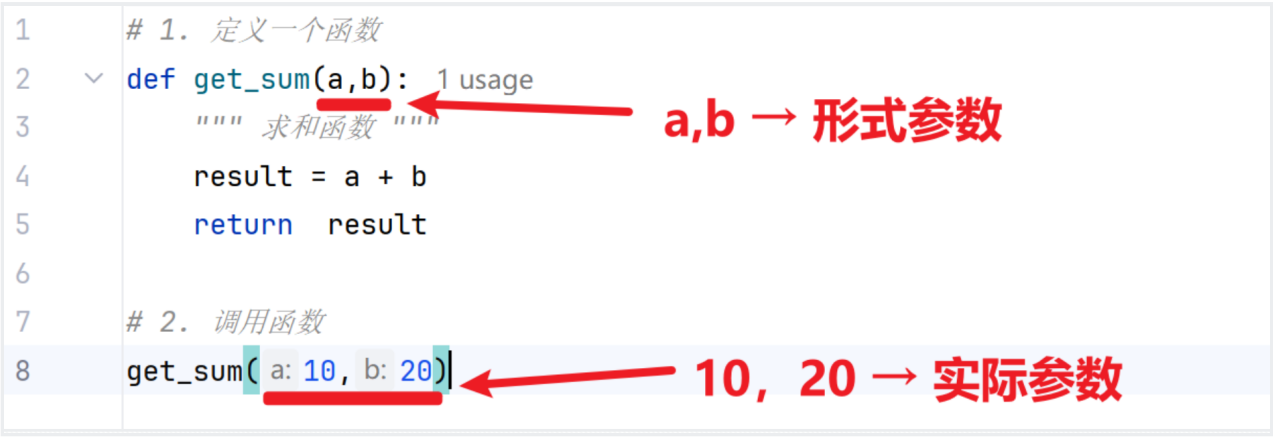
函数的说明文档的定义

```
# 定义函数的说明文档
def 函数名(形参列表):
"""
    说明文档
    参数:
    返回值:
    描述:
    编写人:
    ...
"""

# 查看函数的说明文档
help(函数名)

#####################################################################

# 案例演示
# 定义函数
def sum(a,b):
    """
    :param a:    值1
    :param b:    值2
    :return:     两个数的和
    :auther:     ciggar
    :date:       2025-10-14
    """
    return a+b

# 查看说明文档
help(sum)
```

形参与实参

概念解释：

- **形参**：全称**形式参数**，是**函数定义时**放在括号内的参数，简称形参。用于**规定**函数需要接收的**输入类型**和**数量**
- **实参**：全称**实际参数**，是**函数调用时**传递给函数的具体数值（或变量、表达式），它会“填充”形参的占位符，为函数提供实际的计算数据。



记住一点：

- 形参是在函数定义时出现
- 实参是在函数调用时出现

注意事项：

1 **默认情况下，函数的实参与形参要一一对应，位置也要对应**

2 Python中形参的类型也可以定义，但是仅仅用于提示，解释器不会强制检查

函数的返回值

举个例子：

- 我给儿子10块钱，让他给我买包烟.这个例子中，10块钱是我给儿子的，就相当于调用函数时传递到参数，那么不管这个买烟的过程是什么样的，最终的目标就是得到这包烟，那么这包烟就是返回值
- 定义一个函数去计算数字计算，那么不管数字的计算过程是什么样的，最终这个函数的目的是把计算的结果返回给函数的调用者，计算的结果就是返回值

概念解释

- 概述：函数返回值是**函数执行完成后**，通过 **return** 语句传递给调用者的结果
- 格式：在函数内部通过 `return 返回值` 这个格式即可返回
- 细节：在哪里调用函数，就把返回值返回到哪里

```
# 定义函数
def get_sum(a, b):      # 形参
    """
    该函数用于计算两个整数和
    :param a: 求和计算的第1个整数
    :param b: 求和计算的第2个整数
    :return: 返回求和结果.
    """
    sum = a + b
    return sum


# 调用函数
sum = get_sum3(1, 2)       # 1,2是实参，这里使用sum变量接收函数调用的返回值
print(f'求和结果为: {sum}')
```

函数的嵌套调用

什么是函数的嵌套?

函数的嵌套是指函数的嵌套调用，是指在一个函数里面又调用了另一个函数，**不是**指函数的嵌套定义。

```python
def testB():
    print('---- testB start----')
    print('这里是testB函数执行的代码...(省略)...')
    print('---- testB end----')

def testA():
    print('---- testA start----')
    # 嵌套调用函数B
    testB()
    print('---- testA end----')

# 调用函数
testA()
```

执行结果：

```
D:\soft\python\py313\python.exe D:\back\study\second-project\05-函数与递归\04-函数的嵌套.py
---- testA start----
---- testB start----
这里是testB函数执行的代码...(省略)...
---- testB end----
---- testA end----
```

执行流程：

执行行号的顺序： 13→6→7→8→9→1→2→3→4→9→10→13

如果函数A中，调用了另外一个函数B，那么先把函数B中的任务都执行完毕之后才会回到上次 函数A执行的位置

案例：定义两个函数，一个用于计算三个数的和，另一个用于计算三个数的平均值。

```python
# 1. 定义函数 get_sum(), 用于计算三个整数的和
def get_sum(a, b, c):
    """
    求和函数.
    :param a: 求和操作的第1个整数
    :param b: 第2个整数
```

```
    :param c: 第3个整数
    :return: 3个整数的 和
    """
    sum = a + b + c
    return sum

# 2. get_avg()函数中, 调用get_sum()函数, 并计算它们(三个整数)的平均值.
def get_avg(a, b, c):
    """
    求 平均值 函数
    :param a: 要操作的第1个整数
    :param b: 第2个整数
    :param c: 第3个整数
    :return: 3个整数的平均值.
    """
    sum = get_sum(a, b, c)
    avg = sum // 3
    return avg

# 3.调用,并打印结果
# avg = get_avg(10, 20, 30)
# print(f'平均值为: {avg}')

if __name__ == '__main__':
    avg = get_avg(10, 20, 30)
    print(f'平均值为: {avg}')
```

注意事项：

1. main函数是程序的主入口，所有代码的执行都是从这里开始的
2. main方法可以省略不写(建议写)，Python解释器底层执行的时候，会自动帮我们加
3. 当存在嵌套调用的时候，函数定义的顺序无所谓

变量的作用域

变量的作用域：指的是变量的作用范围（变量在哪里可用，在哪里不可用）

变量主要分为两类：

- 局部变量：是指定义在函数体内部的变量，即只在函数体内部生效
- 全局变量：指的是在函数体内、外都能生效的变量

局部变量

局部变量：在函数体内部定义，用于临时保存数据，当函数调用完成之后，销毁局部变量。

举例

```
def testA():
      a = 100
      print(a)

testA()      # 100
print(a)     # 报错：name 'a' is not defined，函数外部不能访问
```

```
def testB():
    a = 100
    print(a)
    a += 200
    a -= 300
    print(a)
    list1 = [a]
    # ...

def testC():
    pass

# ...
```

局部变量a： 作用域仅仅是在函数内部

全局变量

全局变量：定义在函数外的变量，或者用 global 关键字修饰的变量

举例

```
b = 100

def func2():
    print(f'func2函数, b: {b}')        # 访问全局变量b，并打印变量b存储的数据

def func3():
    print(f'func3函数, b: {b}')        # 访问全局变量b，并打印变量b存储的数据

# 调用函数
func2()          # 100
func3()          # 100
```

global关键字

使用 global关键字 可以在函数内部声明变量为全局变量。

```
a = 100

def testA():
      print(a)

def testB():
      # global 关键字声明a是全局变量
      # 如果这里没有写global a, 那么 a = 200就相当于是在testB中重新定义了一个局部变量
      global a
      a = 200
      print(a)

testA()    # 100
testB()    # 200
print(f'全局变量a = {a}')    # 全局变量a = 200
```

对比与总结

1. 局部变量：定义在函数内或者函数形参列表中的 变量
2. 全局变量：定义在函数外的变量，或者用 global 关键字修饰的变量
3. 他们有什么区别？(面试题)
   - 定义的位置不同
   - 在内存中的位置不同
     - 全局变量：在堆中
     - 局部变量：在栈帧中
   - 生命周期不同
     - 局部变量：随着函数的调用而存在，随着函数的调用完毕而消失
     - 全局变量：随着 `.py` 文件加载而存在， `.py` 文件从内存中移除而消失

注意事项：

1. 程序在查找变量的时候，遵循**就近原则**。局部位置有就直接使用，没有就去 全局位置找，有就使用，没有就报错
2. main函数中定义的变量属于全局变量（但是对其他模块不可见）
3. 全局变量的定义，建议放在函数定义之前。因为如果在执行函数时，全局变量还有定义，那么程序会报错
4. 全局变量的命名，为了和局部变量区分，建议使用 `g_` 或者 `gl_` 或者 `global_` 这样的前缀

多函数程序执行流程

一般在实际开发过程中，一个程序往往由多个函数组成，有两点需要注意

- **函数的返回值可以作为（另一个函数的）参数进行传递**
- **函数可以作为（另一个函数的）参数进行传递**

案例1： 函数的返回值作为另一个函数的参数进行传递

```python
# 演示 1. 函数的返回值可以作为 (另一个函数的)参数进行传递.
def fun1():
    return 100       # 返回1个结果 100

def fun2(num):       # 需要1个参数
    print(num)

# 调用fun1()函数, 获取返回值.
a = fun1()           # 相当于 a = 100

# 调用fun2()函数.
fun2(a)

# 合并版.
fun2(fun1())         # 函数的返回值可以作为 (另一个函数的)参数进行传递.
```

案例2：函数可以作为（另一个函数的）参数进行传递

```python
# 定义加法函数
def get_sum(a, b):
    return a + b

# 定义减法函数
def get_substract(a, b):
    return a - b

# 定义计算函数
def calculate(a, b, fn):
    """
    自定义函数, 模拟计算器, 传入什么 函数(对象), 就做什么操作.
    :param a: 要操作的第1个整数
    :param b: 要操作的第2个整数
    :param fn: 具体的操作规则
    :return: 计算结果.
    """
    return fn(a, b)


# 把函数作为calculate的参数进行传递
```

```
print(calculate(10, 20, get_sum))
print(calculate(10, 20, get_substract))
```

总结

1 函数的概念

一个被命名的，独立的，具有特定功能的代码片段，并且会给调用它的程序一个返回值

2 函数的定义

```
def 函数名(形参列表):
    函数体
    ...
    return 返回值
```

3 函数的说明文档

用于对函数的功能，参数，返回值等进行解释说明的一种特殊的注释，使用三引号定义在函数的第一行

4 形参与实参

形参：在函数定义时声明的参数，用于规定函数需要接收的输入类型和数量

实参：在函数调用时传入的具体数值，为函数提供实际的计算数据

5 返回值的概念

函数返回值是**函数执行完成后**，通过 **return** 语句传递给调用者的结果

6 函数的嵌套调用

在函数内部需要某些数据的时候，可以通过调用另外一个函数得到

7 变量的作用域

- 局部变量：定义在函数内部，作用区间也在函数内部
- 全局变量：定义在函数外部，或者是使用global关键字定义，在函数内外都能生效

函数进阶

在本章节将介绍一些函数的高级用法。

学习目标：

1 掌握函数返回多个返回值

2 **重点掌握函数的多种参数**

- 位置参数
- 关键字参数
- 缺省参数
- 不定长参数

3. **掌握拆包的用法**
4. 知道可变类型和不可变类型有哪些
5. 理解引用的概念，掌握Python的引用传递
6. 了解匿名函数的用法

函数的返回值进阶

思考： 如果一个函数，有两个return返回语句，那么程序如何执行？

```
def return_num():
    return 1
    return 2

result = return_num()
print(result)                # 1
```

结果：只执行了第一个return，原因是因为return可以退出当前函数，导致return下方的代码不执行

那么，如果一个函数需要有多个返回值，该如何书写代码呢?

```
def return_num():
    return 1,2

result = return_num()
print(result)                # (1,2)
print(type(result))          # <class 'tuple'>
```

注意：

1. return a，b写法，用于返回多个结果（这里只是2个）
2. 返回多个结果的时候，默认是元组类型
3. return返回多个值的时候，单个值的类型没有限制，可以是列表、元组、字典等等

函数的多种参数

由于在使用方式上的不同，函数有如下4中常见的参数：

- 位置参数
- 关键字参数
- 不定长参数
- 缺省参数

以上的四种常见用法，都需要重点掌握。

位置参数

位置参数是指： **调用函数时根据函数定义的参数的位置来传递参数**，这是默认的传值方式

“一一对应”包括：

- 个数一致
- 顺序一致

关键字参数

关键字参数：**函数调用时通过“键=值”形式传递参数**

作用：可以让函数更加清晰、容易使用，同时也清除了参数的顺序需求

```
1  def print_info(name,age,gender):
2      print(f"您的名字是:{name}")
3      print(f"您的年龄是:{age}")
4      print(f"您的性别是:{gender}")
5
6  print_info(age=30,gender='女',name='Alice')
7  print_info('bob',gender='男',age=18)
```

注意事项：

- 关键字参数一般用于参数列表比较长的场景，可以让传值更方便
- 关键字参数可以和位置参数混用，如果有位置参数时，位置参数比较在关键字参数的前面，且关键字参数之间没有先后顺序

缺省参数

缺省参数：**缺省参数也叫默认参数，用于定义函数，为参数提供默认值，调用函数时可不传该默认参数的值**

作用：当调用函数时没有传递参数，就会使用默认是用缺省参数对应的值

```python
# 缺省参数
def print_info(name,age,gender='男'):
    print(f"您的名字是:{name}")
    print(f"您的年龄是:{age}")
    print(f"您的性别是:{gender}")

print_info('Tom',20)              # 可以不传缺省参数的值，使用缺省值
print_info("秀芹",25,'女')        # 也可以传缺省参数的值，使用传入的值
```

注意事项：

1 定义缺省参数的时候，所有的位置参数必须在缺省参数之前

2 函数调用时，如果没有为缺省参数传值，那么使用默认值，否则使用传入的值

不定长参数(多值参数)

不定长参数：**不定长参数也叫可变参数，即参数的个数是可以变化的。**

Python 中，不定长参数用于处理“参数数量不确定”的场景，分为两类：

- `*args`：接收**多个位置参数**，将参数打包为元组（tuple）；
- `**kwargs`：接收**多个关键字关键字参数**，将参数打包为字典典（dict）。

两者的核心区别在于传递方式：

- `*args` 处理“按位置顺序传递的参数”
- `**kwargs` 处理“按 `键=值` 格式传递的参数”。

按位置传递

语法与特点：

- 定义格式：`def 函数名(*args):`（`args`是约定俗成的变量名，可替换为其他名称，但需保留`*`）
- 传递方式：调用函数时，直接传递多个值（无需指定参数名），按位置顺序打包
- 内部存储：`args`是元组，可通过索引访问每个参数（如`args[0]`获取第一个参数）

```python
# 需求：求任意个整数的和
def sum_total(*args):
    """
    求和
    :param args: 个数未知的多个数字
    :return:     所有参数的和
    """

    total = 0
    for num in args:
        total += num
    return total


print(sum_total(1, 2, 3, 4))          # 10
print(sum_total(1, 2, 3))             # 6
print(sum_total(10, 20))              # 30
```

关键字传递

语法与特点：

- 定义格式：`def 函数名(** kwargs):`(`kwargs`是约定俗成的变量名，可替换，需保留`**`)
- 传递方式：调用函数时，传递多个`键=值`格式的参数（必须指定参数名），按`键-值`对打包
- 内部存储：`kwargs`是字典，可通过键名访问对应值(如`kwargs["name"]`获取`name`参数的值)

```python
def print_user_info(** kwargs):
    print("接收的关键字参数（字典）：", kwargs)
    print(f"姓名：{kwargs.get('name')}，年龄：{kwargs.get('age')}，性别：{kwargs.get('gender')}")

# 调用函数，传递多个关键字参数
print_user_info(name="Alice", age=20, gender="女")
print_user_info(name="Bob", age=22, gender="男", hobby="篮球")  # 可传递额外参数
```

注意事项：

1. 实际开发中，`*args` 和 `**kwargs` 通常配合起来使用，用于接收任意数量的关键字参数和位置参数
2. 在它们一起使用的时候，一定需要注意函数声明的时候，`*args`在前，`**kwargs` 在后，否则解释器报错

拆包

拆包：对于函数中的多个返回数据，去掉元组，列表 或者字典 直接获取里面数据的过程。

主要包括：

- 对元组拆包

```python
# 对元组拆包
my_tup = ('张三', 20, 88.5, '湖北武汉')
name, age, score, addr = my_tup
print(f"玩家的名字是:{name}, 年龄是:{age}, 得分是:{score}, 地址是:{addr}")

# 对函数的多个返回值拆包
def sum_substract(a, b):
    return a + b, a - b

# 返回的默认是元组，可以直接对返回的元组进行拆包
total, result = sum_substract(10, 20)
print(f"两个数的和是:{total}")
print(f"两个数的差是:{result}")
```

- 对列表拆包

```python
# 对列表拆包
my_list = [1,3.14,'Cskaoyan',True]
num,pi,name,my_bool=my_list
print(f"num:{my_list}, pi:{pi}, name:{name}, my_bool:{my_bool}")
```

- 对字典拆包

```python
# 对字典拆包
dict1 = {'name':'李云龙','age':20,'gender':'男'}

# 对字典拆包的时候，获取到的是key值
key1,key2,key3 = dict1
print(f"key1:{key1},key2:{key2},key3:{key3}")
```

经典案例：有变量a = 10和b = 20，交换两个变量的值

```python
# 使用拆包交换两个变量的值
# 传统方式（需要临时变量）
a = 10
b = 20
print("交换前：a =", a, "b =", b)  # 交换前：a = 10 b = 20

# 传统交换方式
temp = a
a = b
b = temp
print("传统方式交换后：a =", a, "b =", b)  # 传统方式交换后：a = 20 b = 10

# 拆包方式交换（更简洁）
a, b = 10, 20  # 重新初始化
a, b = b, a    # 核心：利用拆包同时赋值
print("拆包方式交换后：a =", a, "b =", b)  # 拆包方式交换后：a = 20 b = 10

# 原理说明：
# 等号右边的"b, a"会先被打包成一个元组(b, a)
# 然后通过拆包赋值给左边的"a, b"
# 整个过程是先计算右边所有值，再同时赋值给左边变量，因此无需临时变量
```

引用与引用传递

学习目标：

1. 掌握引用的概念
2. 掌握Python中可变类型与不可变类型
3. 掌握Python中不同情况下的引用传递

引用

什么是引用?

引用就像生活中快递柜的取件码。

- 你买的最新款的iPhone放在快递柜里面

  类比： iPhone类比为对象，是实际的数据，快递柜就是内存，我们的所有数据都是放在内存中

- 快递柜给你的是“取件码”而不是手机本身，取件码就是找到手机的地址

  类比：取件码类比为引用，可以理解为是实际数据的地址值

- 你把取件码告诉朋友，朋友用同样的码也能拿到手机

  类比：多个引用，可以指向同一个对象

python中的引用?

变量 ≠ 对象：变量只是 “引用的名字”，对象才是真正的数据；

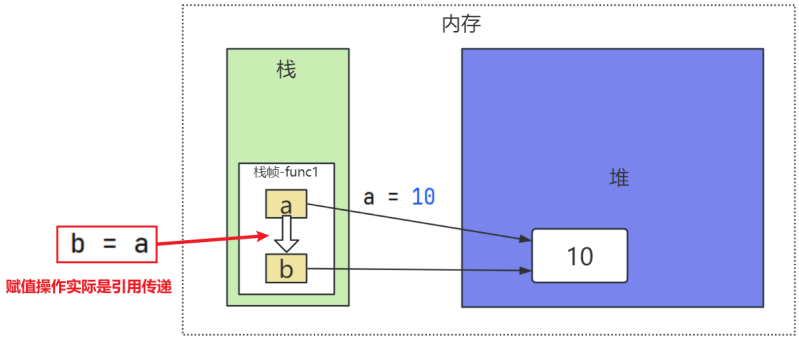
比如 a = 10：

- 10 是**整数对象**（存在内存里）
- a 是**引用**（指向 10 这个对象的 “地址”）

再比如 b = a：

- 不是把 10 复制一份给 b，而是让 b 也指向 10 这个对象（两个引用指向同一个对象）。

```
def fuc1():
    a = 10
    b = a
```



堆：实际存放对象（数据）的一块内存空间，所有对象的仓库
栈：局部引用的临时工作区，函数调用时，Python 会在 “调用栈” 上创建一个栈帧，专门存储函数内的局部变量引用、参数引用、临时计算结果；

***补充说明**：局部变量的引用在栈帧中，全局变量的引用在模块的命名空间（堆）中。*

每次函数调用，CPython（解释器） 会创建一个 `PyFrameObject`：

```
栈帧（frame）：
    - 局部变量表（fast locals）
    - 操作数栈
    - 指令指针
```

📌 **注意：**

> 这个“栈帧”是 **虚拟机层面的栈**，不是 C 语言意义上的 CPU 栈。

验证引用传递

打印a和b的地址值，可以看到地址值是一样的，实际是同一个数据，并不是内存中的两份数据

```python
def fuc1():
    a = 10
    b = a

    print(id(a))            # 140730885489864
    print(id(b))            # 140730885489864

if __name__ == '__main__':
    fuc1()
```

函数中的引用传递

结论：Python中只有引用传递。一句话：一切皆引用

- 函数传参时，**传递的是“指向堆中对象的引用”**，不是对象本身；
- 引用的存储位置会变（比如参数引用存到新栈帧），但堆中对象的位置不变；
- 最终效果由“对象是否可变”决定，核心是“改引用”还是“改堆中对象”。

可变类型：列表、字典、集合

不可变类型：整数、字符串、元组

```python
######################################## 传递不可变对象
########################################

def change_str(s):
    # 1. 函数调用创建栈帧，s接收外部引用（指向堆中"hello"）
    # 2. "world"是新对象（存堆），s的引用被修改为指向"world"（改的是栈帧里的引用）
    s = "world"
    print("函数里的s:", s, id(s))   # id(s)是新对象"world"在堆的地址

my_str = "hello"  # my_str的引用（模块命名空间）指向堆中"hello"
```

```python
change_str(my_str)
# 函数结束：栈帧销毁，s的引用消失；my_str的引用仍指向堆中"hello"
print("函数外的my_str:", my_str, id(my_str))

######################################## 传递可变对象
########################################

def add_item(lst):
    # 1. 函数调用创建栈帧，lst接收外部引用（指向堆中[1,2,3]）
    # 2. append(4)直接修改堆中列表对象的内容（没改lst的引用）
    lst.append(4)
    print("函数里的lst:", lst, id(lst))  # id(lst)还是原来堆中列表的地址

my_list = [1,2,3]  # my_list的引用（模块命名空间）指向堆中列表
add_item(my_list)
# 函数结束：栈帧销毁，lst的引用消失；但堆中列表已被修改
print("函数外的my_list:", my_list, id(my_list))
```

结论

1. **引用是“地址”**：变量存的是引用（指向堆中对象），不是对象本身；
2. **存储分工**：
   - 堆：存所有对象（真正的数据）
   - 栈帧：存函数内局部变量 / 参数的引用（函数结束销毁）
   - 全局引用：存模块命名空间（长期存在）
3. **传参本质**：只传引用（引用从外部到函数栈帧），不改堆中对象地址
4. **效果判断**：
   - **不可变对象：改栈帧里的引用 → 外部不变**
   - **可变对象：改堆中的对象内容 → 外部也变（因为两个引用指向同一块堆空间）**

练一练：

1. 运行代码，思考： my_num 变没变？栈帧和堆发生了什么？

```python
def change_num(n):
    n += 5  # n是栈帧中的引用，指向堆中新对象15
my_num = 10  # 堆中对象10，my_num引用在模块命名空间
change_num(my_num)
print(my_num)  # 输出？
```

2. 运行代码，思考：my_list变没变？栈帧和堆发生了什么？

```python
def clear_list(lst):
    lst.clear()  # 改堆中列表的内容（清空），没改lst的引用
my_list = [1,2,3]  # 堆中列表，my_list引用在模块命名空间
clear_list(my_list)
print(my_list)  # 输出?
```

匿名函数

什么是匿名函数?

- 概述：没有名字的函数，就叫做匿名函数
- 格式：`lambda 参数1, 参数2, ... : 表达式 # 冒号前是参数，冒号后是返回结果的表达式`
- 细节：
  - Python的匿名函数 类似于 Java中的Lambda表达式
  - 匿名函数适用于简单的业务需求，即：函数体只有一行代码的函数
  - 匿名函数的应用场景：
    - 当对方法仅调用一次
    - 匿名函数 可以作为 函数对象 进行传递

需求1：定义函数，用于计算两个整数和

```python
# 普通函数.
def get_sum(a, b):
    return a + b

print(get_sum(10, 20))

# 匿名函数
my_get_sum = lambda a, b :  a + b
print(my_get_sum(11, 22))
```

需求2：定义函数，接收2个整数，分别计算两个整数的：和，差，积，商，最大值，最小值.

```python
# 普通函数
def get_sum(a, b):
    return a + b            # 求和

def get_sub(a, b):
    return a - b            # 差

def get_mul(a, b):
    return a * b            # 积

```

```python
# ...


# 匿名函数
def cal_num(a,b,fn):
    return fn(a,b)

a = 10
b = 20
# 求和
sum_result = cal_num(a,b,lambda a,b:a+b)
sub_result = cal_num(a,b,lambda a,b:a-b)
mul_result = cal_num(a,b,lambda a,b:a*b)
```

递归

大和尚与小和尚的故事。

递归的定义

我们已经学习过方法，使用过函数了。函数体中是可以调用函数的，那么如果在函数体中调用函数自身呢？

我们把方法在运行时调用自身的情况，称之为递归，又叫做递归调用。

使用递归的注意事项

递归的使用有很多限制，尤其要注意以下两点：

1. 合法的递归，除了要有递归体语句外，还要有递归出口。无限制的递归下去，会引发栈溢出错误（）
2. 即便是有出口的递归，递归的深度也不能超过栈空间的大小，否则仍然会报错

案例1：自然数求和

```python
def sum1(n):
    # 递归出口
    if n == 1:
        return 1
    return n + sum1(n-1)
```

递归的思想

案例2：使用递归计算N（N>=1）的阶乘 **(factorial)**

这个代码很好写，参考如下：

```
# 求n的阶乘
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)
```

我们可以总结一下使用递归的两要素：

1. 递归体（方法中自身调用自身方法的那句语句）
2. 递归出口

以上两部分对于一个正常的递归而言都是必须的，在实际使用中，我们只要找到这两个部分就能够写出递归的代码了。

观察这样的一个代码，我们想求n的阶乘，就只需要知道（n - 1）的阶乘的值，（n - 1）阶乘的结果就需要知道（n - 2）阶乘的结果，最终我们知道1的阶乘就是1。

这种将大问题分解为小问题的思想就是递归的思想：

1. 把一个复杂的大规模的问题，分解成若干相似的小规模的子问题。
2. 当子问题规模足够小的时候，就可以直接得到小规模问题的解。
3. 然后把所有的小规模的子问题的解，组合起来，得到要求解的大规模问题的解。

对比一下，下面没有使用递归，正常使用for循环的代码：

循环求n的阶乘

```
def factorial(n):
    result = 1
    for i in range(1,n+1):
        result *= i
    return result
```

不难发现递归的优点是：

1. 递归的代码会非常简洁，这是最直观的。
2. 人在解决问题的时候，都会下意识的分解问题。递归的思想很符合人的思维，用递归求解一个问题的思路很容易被人理解。
3. 接第二条，一旦能够找打分解问题的思路，递归会非常好用。

当然递归的缺点也非常明显：

1. 不用递归时，往往一个方法就能解决问题。而递归会调用多个方法，占用大量栈内存，且明显存在重复计算，效率低。也就是说，**使用递归求解一个问题，时间和空间复杂度都不占优势，既占用空间效率还低。**
2. 栈溢出错误警告！递归很危险，一旦栈溢出是严重错误！

综上，递归是一把伤人亦伤己的利器，实际开发中不要随意使用递归，使用递归要深思熟虑递归的深度和出口，避免栈溢出错误

经典案例

案例1：斐波那契(Feibonacci)数列

1，1，2，3，5，8，13，21.....

求第n个位置的值是多少

```
def get_feibonacci(n):
    if n == 1 or n == 2:
        return 1
    return get_feibonacci(n-1) + get_feibonacci(n-2)
```

案例2：青蛙跳台阶

一只青蛙一次可以跳上一层台阶，也可以跳上两层

求该青蛙跳上n层的台阶总共有多少种跳法（先后次序不同算不同的结果）

```
def judge_floor(n):
    if n == 1:
        return 1
    if n == 2:
        return 2
    return judge_floor(n-1) + judge_floor(n-2)
```

PyCharm的调试(补充)

我们目前已经学完了函数相关的知识点，我们也知道了函数之间是可以互相调用的。函数之间互相调用，层次可能会很深，如果在复杂的工程中，排查问题就变成了一个大麻烦。

为此，PyCharm提供了调试模式（DEBUG模式）供我们使用。

调试模式是 PyCharm 自带的 “代码诊断工具”，能让代码**一步步执行**，方便观察：

- 变量在执行过程中的变化；
- 代码执行的路径（哪行执行了，哪行没执行）；
- 哪里出现了预期外的错误（比如逻辑错误、变量值异常）。

接下来看个案例：

```python
def factorial(n):  1 usage
    result = 1
    for i in range(1, n):  # 这里有错误：range应该到n+1才对
        result *= i
    return result

# 测试：预期5! = 120，实际会得到24（因为循环少执行了一次）
print(factorial(5))
```

以DEBUG模式运行：（一定要添加断点）

1：rerun，重启程序

2：stop，停止程序

3：**resume，放行这个断点，继续执行（F9）**

4：**step over，下一步（F8）**

5：**step into，进入函数（F7）**

6：step into my code，进入我的代码中的函数

7：**step out，跳出函数**（shift+F8）

8：浏览所有的断点

9：禁用所有的断点

10：更多选项

11：显示线程和变量

12：控制台

Python的内存管理原理

- int对象

int清楚写了[-5, 256]这些小整数被定义在了这个对象池里.所以当引用小整数时会自动引用整数对象池里的对象的.

好处：每次去初始化小整数变量，速度很快，无需构造数值对象

Python一个数值对象默认占用的空间是28个字节

- **string对象（可以用于面试）**

string对象也是不可变对象,python有个intern机制,简单说就是维护一个字典,这个字典维护已经创建字符串(key)和它的字符串对象的地址(value),每次创建字符串对象都会和这个字典比较,没有就创建,重复了就用指针进行引用就可以了.

string实现了intern共享？我觉得是一种空间效率和时间效率的妥协。相比于数字，string本身参与的运算要少很多，而且string本身占据的空间也大许多，因此string的主要问题在于不共享带来的空间浪费，所以string实现了很费时间的intern操作。

对于数字情况正好相反。作为一个数字，需要做的运算要比string多太多了，而且大小比string也小很多。如果在计算10000+20000之前先花好久查找重复对象，导致一个1ms完成的加法花了100ms，我肯定想砸电脑的。

- float对象

float类型可以认为每个赋值都是创建一个对象,因为float有点多,所以没必要和int一样了.

- tuple对象

tuple它是不可变对象,理应和int和string一样会做一个缓存,但是书上没有说明,于是看了看源码

Python是一门垃圾收集语言，这意味着当一个变量不再被使用时，Python会将该变量使用的内存释放回操作系统，以供其他进程（变量）使用。然而，对于长度为1~20的元组，即使它们不在被使用，它们的空间也不会立刻还给系统，而是留待未来使用。这意味着当未来需要一个同样大小的新的元组时，我们不再需要向操作系统申请一块内存来存放数据，因为我们已经有了预留的空间。

必须记住的

引用计数为0，对应的对象就会被释放