# 变量进阶（理解）

## 目标

- 变量的引用
- 可变和不可变类型
- 局部变量和全局变量

## 01. 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

- 定义一个整数变量 `a`，并且赋值为 `1`


| 代码 | 图示 | | :---: | :---: | | a = 1 | |

- 将变量 `a` 赋值为 `2`


| 代码 | 图示 | | :---: | :---: | | a = 2 | |

- 定义一个整数变量 `b`，并且将变量 `a` 的值赋值给 `b`


| 代码 | 图示 | | :---: | :---: | | b = a | |

> 变量 `b` 是第 2 个贴在数字 `2` 上的标签

### 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

```python 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))

```

# 02. 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`

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

```python 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，以方便 **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

# 03. 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**

```python 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

```python

# 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

```python 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

```python 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**

```python 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！

**代码结构示意图如下**
```


**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

# 函数进阶

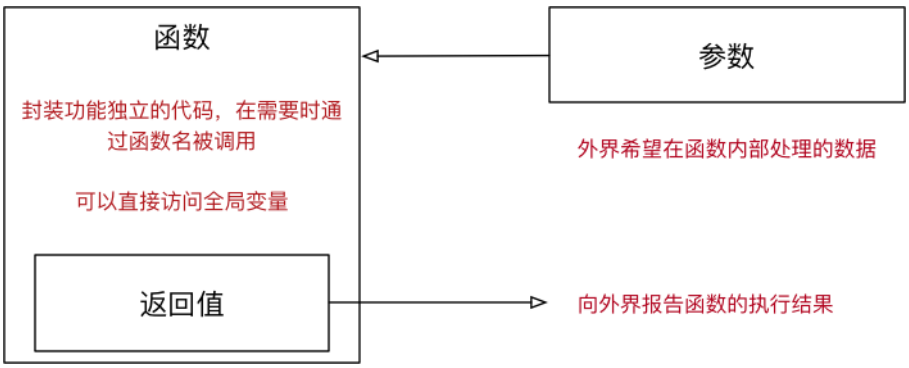
## 目标

- 函数参数和返回值的作用
- 函数的返回值 进阶
- 函数的参数 进阶
- 递归函数

## 01. 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

## 02. 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：

```python 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：

```python 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

`python 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

```python
# 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

```python
# 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

`python a, b = b, a`

## 03. 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**

```python
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**

```python
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

```python
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, gllist) print(glnum) 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

```python 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

```python 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！

`python 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！

```python 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**

```python 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

```python 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** `*`

```python 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)

```

## 04. 函数的递归

> 函数调用自身的 **编程技巧** 称为递归

### 4.1 递归函数的特点

**特点**

- **一个函数 内部 调用自己**
  - 函数内部可以调用其他函数，当然在函数内部也可以调用自己

**代码特点**

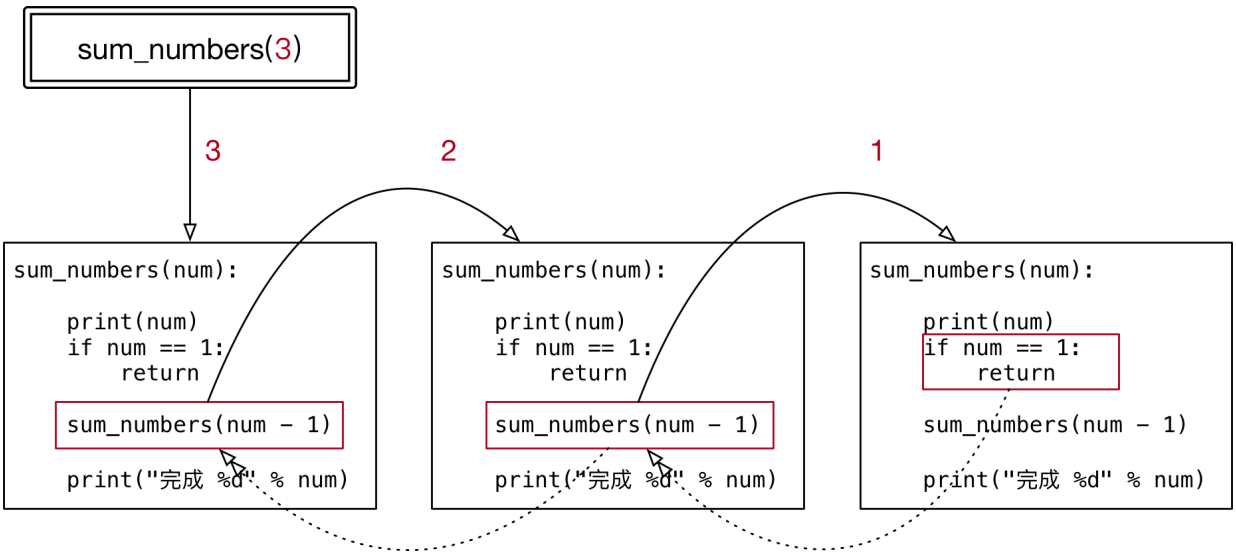
1. 函数内部的 **代码** 是相同的，只是针对 **参数** 不同，**处理的结果不同**
2. 当 **参数满足一个条件** 时，函数不再执行
   - **这个非常重要**，通常被称为递归的出口，否则 **会出现死循环！**

示例代码

```python def sum_numbers(num):

```
print(num)

# 递归的出口很重要，否则会出现死循环
if num == 1:
    return

sum_numbers(num - 1)
```

sum_numbers(3)

```




### 4.2 递归案例 —— 计算数字累加

**需求**

1. 定义一个函数 `sum_numbers`
2. 能够接收一个 `num` 的整数参数
3. 计算 1 + 2 + ... num 的结果

```python def sum_numbers(num):

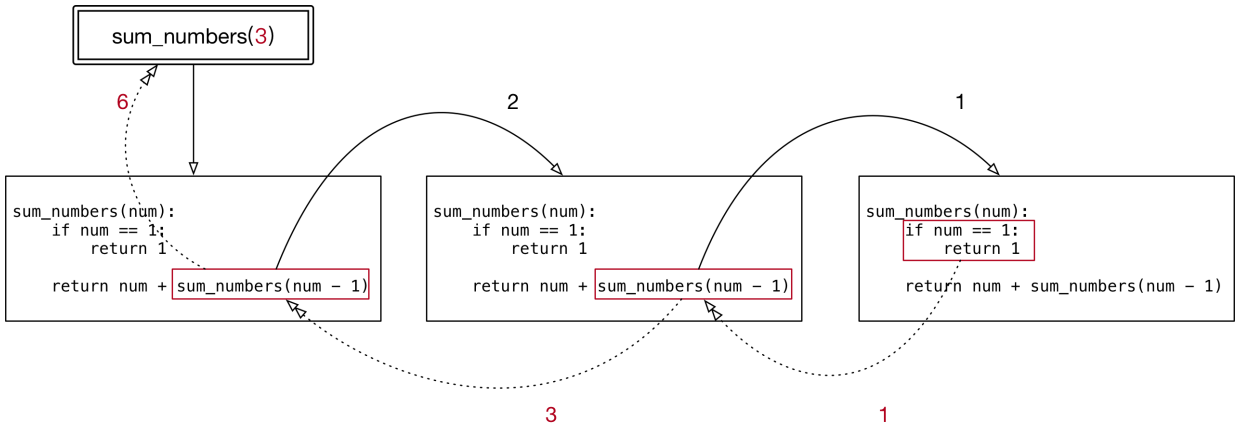
```
if num == 1:
    return 1

# 假设 sum_numbers 能够完成 num - 1 的累加
temp = sum_numbers(num - 1)
```

```
    # 函数内部的核心算法就是 两个数字的相加
    return num + temp
```

print(sum_numbers(2))

```




> 提示：递归是一个 **编程技巧**，初次接触递归会感觉有些吃力！在处理 **不确定的循环条件时，格外的有用，例如：遍历整个文件目录的结构**
```