[笔记][黑马 Python 之 Python 基础 - 7]

Python

[笔记][黑马 Python 之 Python 基础 - 7]
- 226. 变量的引用-01-变量的引用概念
- 227. 变量的引用-02-调用函数传递实参的引用
- 228. 变量的引用-03-函数返回值传递引用
- 229. 可变类型和不可变类型
- 230. 可变类型和不可变类型-02-列表、字典的修改和赋值
- 231. 可变类型和不可变类型-03-字典的key不能是可变类型
- 232. 局部变量和全局变量-01-基本概念和区别
- 233. 局部变量-01-代码演练
- 234. 局部变量-02-变量的生命周期
- 235-局部变量-03-不同函数内的同名局部变量
- 236. 全局变量-01-基本代码演练
- 237. 全局变量-02-[扩展]PyCharm的单步跟踪技巧
- 238. 全局变量-03-函数内部不允许修改全局变量的值
- 239. 全局变量-04-单步调试确认局部变量的定义
- 240. 全局变量-05-global关键字修改全局变量
- 241. 全局变量-06-全局变量定义的位置及代码结构
- 242. 全局变量-07-全局变量命名的建议
- 243. 函数参数和返回值的作用
- 244. 函数的返回值-01-利用元组返回多个值
- 245. 函数的返回值-02-接受返回元组的方式
- 246. 函数的返回值-03-交换两个变量的值
- 247. 函数的参数-01-在函数内部针对参数赋值不会影响外部实参
- 248. 函数的参数-02-在函数内部使用方法修改可变参数会影响外部实参
- 249. 函数的参数-04-列表使用+=本质上是调用extend方法
- 250. 缺省参数-01-回顾列表的排序方法明确缺省参数的概念及作用
- 251. 缺省参数-02-指定函数缺省参数的默认值
- 252. 缺省参数-03-缺省参数的注意事项
- 253. 多值参数-01-定义及作用
- 254. 多值参数-02-数字累加演练
- 255. 多值参数-03-元组和字典的拆包
- 256. 递归-01-递归的特点及基本代码演练
- 257. 递归-02-递归演练代码的执行流程图
- 258. 递归-03-递归实现数字累加
- 259. 递归-04-数字累加的执行流程图

226. 变量的引用-01-变量的引用概念

目标

- 变量的引用
- 可变和不可变类型
- 局部变量和全局变量

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

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

`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`

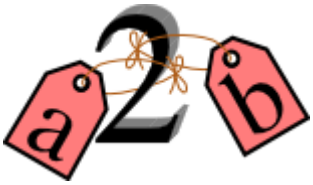
代码	图示
a = 1	

- 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

---

227. 变量的引用-02-调用函数传递实参的引用

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('在函数内部 %d 对应的内存地址是 %d' % (num, id(num)))


# 1. 定义一个数字的变量
a = 10
# 数据的地址本质上就是一个数字
print('a 变量保存数据的内存地址是 %d' % id(a))

# 2. 调用 test 函数，本质上传递的是实参保存数据的引用，而不是实参保存的数据！
test(a)
```

---

228. 变量的引用-03-函数返回值传递引用

```python
def test(num):
    print('在函数内部 %d 对应的内存地址是 %d' % (num, id(num)))
    # 1> 定义一个字符串变量
    result = 'hello'
    print('函数要返回数据的内存地址是 %d' % id(result))
    # 2> 将字符串变量返回，返回的是数据的引用，而不是数据本身
    return result


# 1. 定义一个数字的变量
a = 10
# 数据的地址本质上就是一个数字
print('a 变量保存数据的内存地址是 %d' % id(a))

# 2. 调用 test 函数，本质上传递的是实参保存数据的引用，而不是实参保存的数据！
# 注意：如果函数有返回值，但是没有定义变量接收
# 程序不会报错，但是无法获得返回结果
r = test(a)
print('%s 的内存地址是 %d' % (r, id(r)))
```

229. 可变类型和不可变类型

- **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`

> 可变类型就是可以在原内存地址修改数据，而不用改变内存地址

230. 可变类型和不可变类型-02-列表、字典的修改和赋值

```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

# 修改之后地址不变
# 但是如果重新赋值，比如 a = []，则地址会发生变化
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

---

231. 可变类型和不可变类型-03-字典的key不能是可变类型

字典的 `key` **只能使用不可变类型的数据**
`key` 只能是数字、字符串和元组。开发中最常用的是字符串作为 `key` 。

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` 以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

---

232. 局部变量和全局变量-01-基本概念和区别

- **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！

---

233. 局部变量-01-代码演练

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():
    # 定义一个局部变量
    num = 10
    print('在 demo1 函数内部的变量是 %d' % num)


def demo2():
    # print('%d' % num)
```

```
    pass

# 在函数内部定义的变量，不能在其他位置使用
# print('%d' % num)

demo1()
demo2()
```

---

234. 局部变量-02-变量的生命周期

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**

```
def demo1():
    # 定义一个局部变量
    # 1> 出生：执行了下方的代码之后，才会被创建
    # 2> 死亡：函数执行完成之后
    num = 10
    print('在 demo1 函数内部的变量是 %d' % num)


def demo2():
    # print('%d' % num)
    pass

# 在函数内部定义的变量，不能在其他位置使用
# print('%d' % num)

demo1()
demo2()
```

235-局部变量-03-不同函数内的同名局部变量

```
def demo1():
```

```
    # 定义一个局部变量
    # 1> 出生：执行了下方的代码之后，才会被创建
    # 2> 死亡：函数执行完成之后
    num = 10
    print('在 demo1 函数内部的变量是 %d' % num)


def demo2():
    num = 99
    print('demo2 ==> %d' % num)


# 在函数内部定义的变量，不能在其他位置使用
# print('%d' % num)

demo1()
demo2()
```

---

236. 全局变量-01-基本代码演练

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

```
# 全局变量
num = 10


def demo1():
    print('demo1 ==> %d' % num)


def demo2():
    print('demo2 ==> %d' % num)


demo1()
demo2()
```

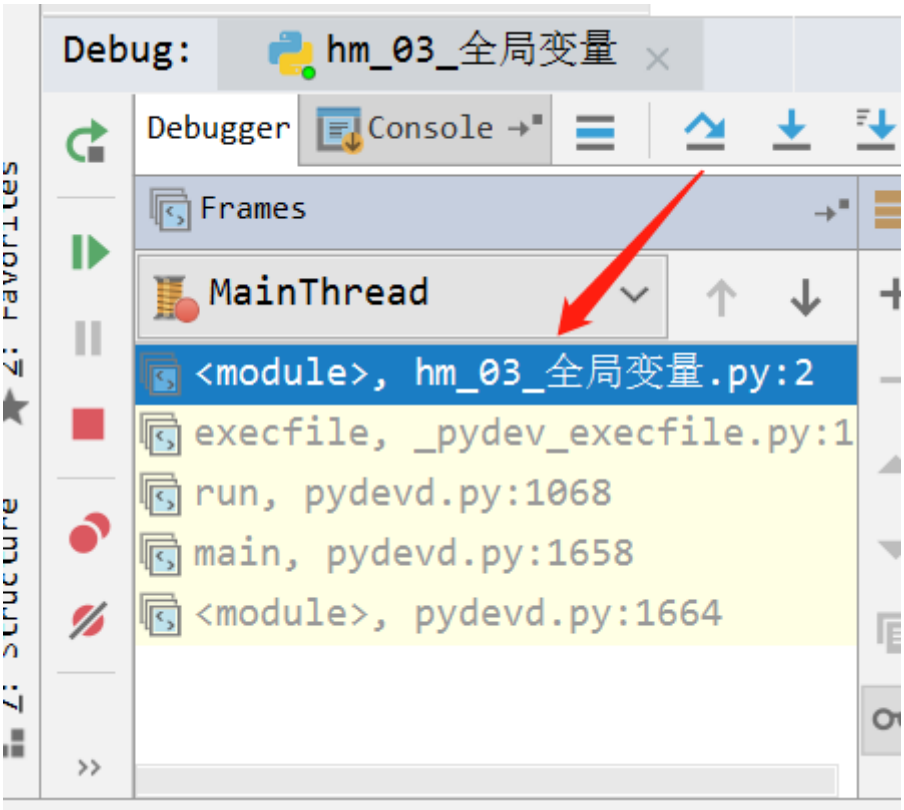
注意：函数执行时，**需要处理变量时** 会：

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！

```

```

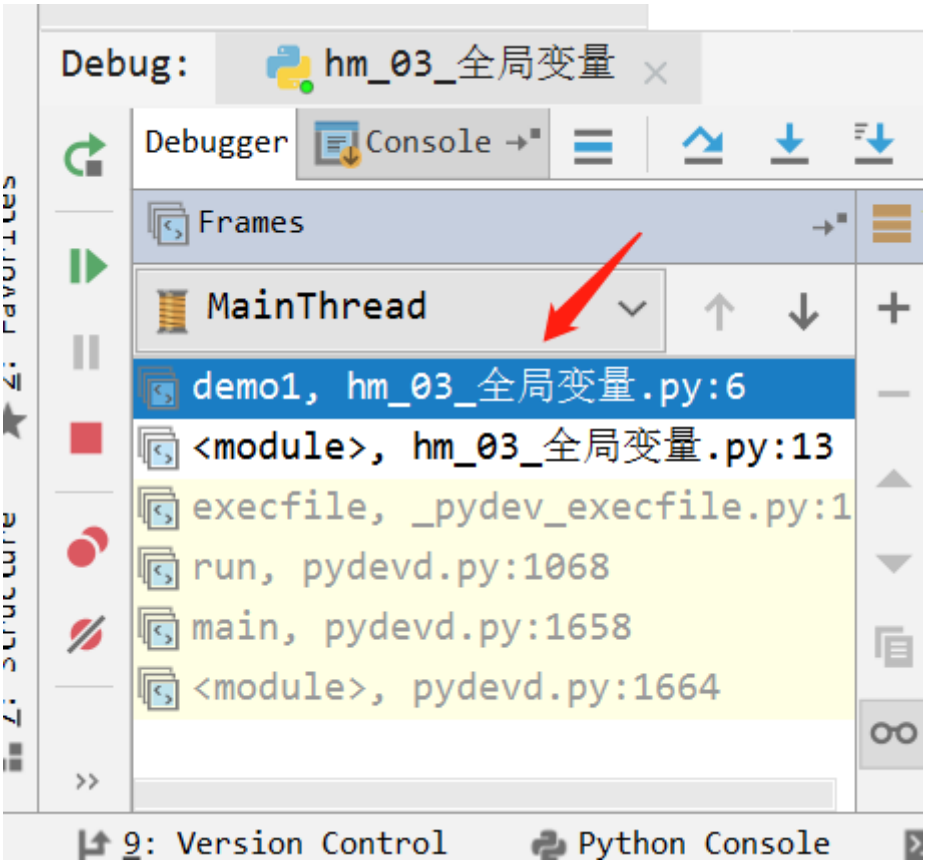
【!】实际上查找顺序是 `LEGB`

237. 全局变量-02-[扩展]PyCharm的单步跟踪技巧

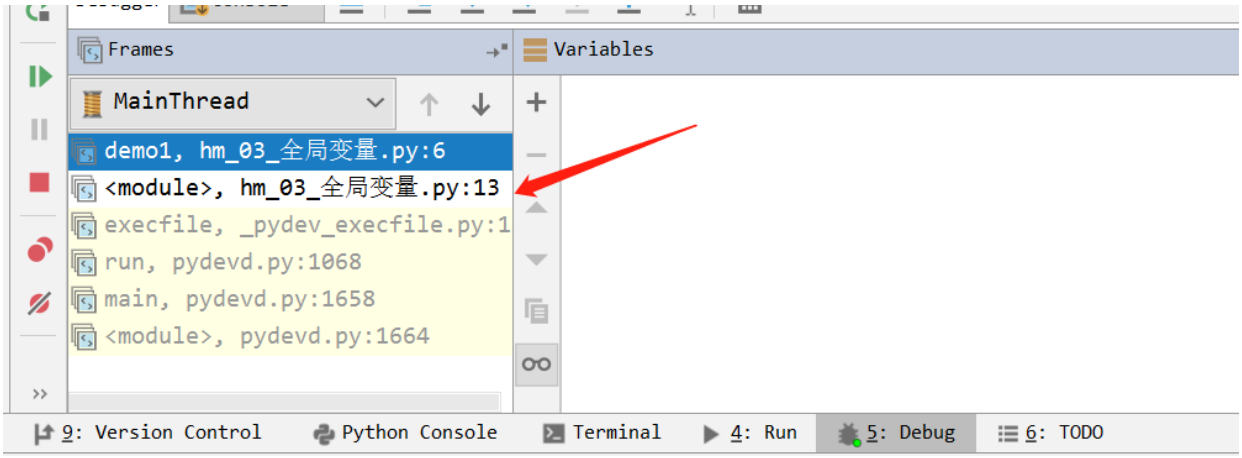


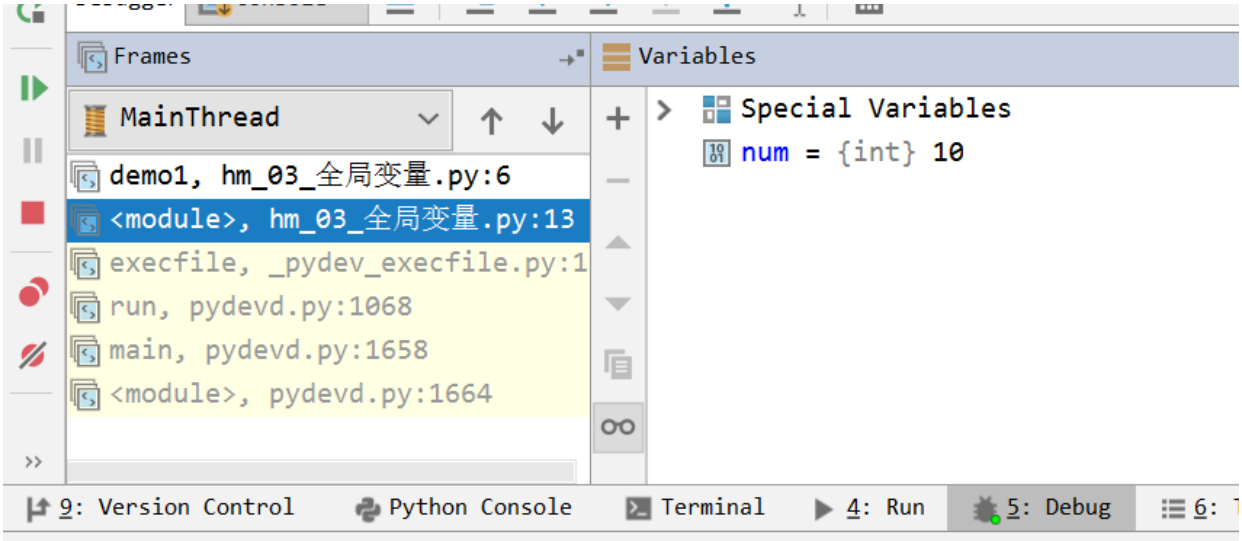
调试窗口的信息意思：
模块，模块名，第几行代码

当使用单步进入 `F7` 进入函数内部执行的时候，左边的窗口信息会保留该函数的代码行数

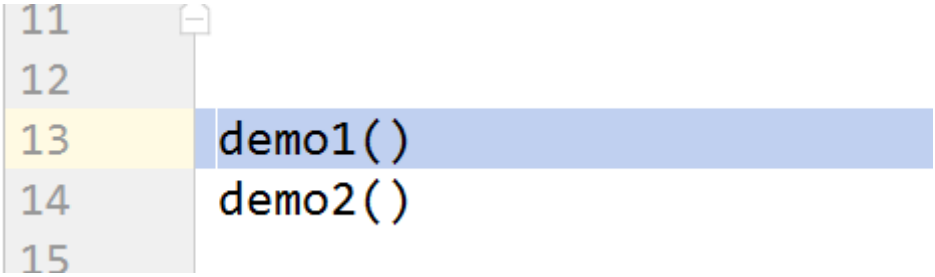


但是在全局变量在函数内部是不会显示的，要显示全局变量，需要点击模块名称





同时编辑区域会有浅蓝色高亮条，表示正在调用该行的函数
（深蓝色高亮条是即将执行该行代码）



---

238. 全局变量-03-函数内部不允许修改全局变量的值

> `Python` 中针对全局变量的一个限制：
> **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

```
# 全局变量
num = 10


def demo1():
    # 希望修改全局变量的值
    # 在 Python 中，不允许直接修改全局变量的值
    # 如果使用赋值语句，会在函数内部，定义一个局部变量
    num = 99
    print('demo1 ==> %d' % num)


def demo2():
    print('demo2 ==> %d' % num)


demo1()
demo2()
```

239. 全局变量-04-单步调试确认局部变量的定义

240. 全局变量-05-global关键字修改全局变量

如果在函数中需要修改全局变量，需要使用 `global` 进行声明

hm_05_修改全局变量.py

```python
# 全局变量
num = 10


def demo1():
    # 希望修改全局变量的值 - 使用 global 声明一下全局变量即可
    global num
    num = 99
    print('demo1 ==> %d' % num)


def demo2():
    print('demo2 ==> %d' % num)


demo1()
demo2()
```

注意：使用了 `global` 关键字之后，所有的 `num` 都高亮了，因为高亮的 `num` 都是同一个引用，即全局变量。

假如不使用 `global`，只有函数内部的 `num` 才会高亮

hm_05_修改全局变量.py

```python
# 全局变量
num = 10


def demo1():
    # 希望修改全局变量的值 - 使用 global 声明一下全局变量即可
    # global num
    num = 99
    print('demo1 ==> %d' % num)


def demo2():
    print('demo2 ==> %d' % num)


demo1()
demo2()
```

demo1()

```python
# 全局变量
num = 10


def demo1():
    # 希望修改全局变量的值 - 使用 global 声明一下全局变量即可
    # global 关键字会告诉解释器后面的变量是一个全局变量
    # 在使用赋值语句时，就不会创建同名局部变量
    global num
    num = 99
    print('demo1 ==> %d' % num)


def demo2():
    print('demo2 ==> %d' % num)


demo1()
demo2()
```

241. 全局变量-06-全局变量定义的位置及代码结构

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
# 注意：在开发时，应该把模块中的所有全局变量定义在所有函数上方
# 就可以保证所有的函数都能正常访问到每一个全局变量
num = 10
# 再定义一个全局变量
title = 'darkhorse'
# 再定义一个全局变量
name = '小明'


def demo():
    print('%d' % num)
    print('%s' % title)
    print('%s' % name)


demo()
```

242. 全局变量-07-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

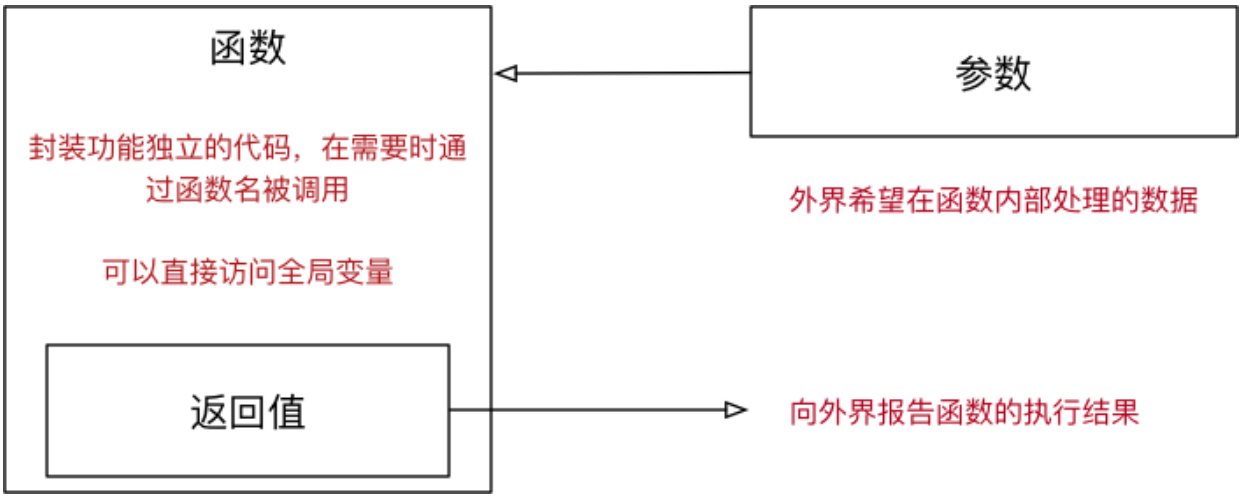
```
# 注意：在开发时，应该把模块中的所有全局变量定义在所有函数上方
# 就可以保证所有的函数都能正常访问到每一个全局变量
gl_num = 10
# 再定义一个全局变量
gl_title = 'darkhorse'
# 再定义一个全局变量
gl_name = '小明'


def demo():
    # 如果局部变量和全局变量同名
    # PyCharm会在局部变量下方显示灰色波浪线
    num = 99
    print('%d' % num)
    print('%s' % gl_title)
    print('%s' % gl_name)


demo()
```

---

243. 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

---

244. 函数的返回值-01-利用元组返回多个值

- 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

> 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

```
def measure():
    """测量温度和湿度"""

    print('测量开始...')
    temp = 39
    humidity = 50
    print('测量结束...')

    # 元组-可以包含多个数据
    # 因此可以使用元组让函数一次返回多个值
    return (temp, humidity)
    # 也可以省略括号
    # return temp, humidity


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

---

245. 函数的返回值-02-接受返回元组的方式

```
def measure():
    """测量温度和湿度"""

    print('测量开始...')
    temp = 39
    humidity = 50
    print('测量结束...')

    # 元组-可以包含多个数据
    # 因此可以使用元组让函数一次返回多个值
    return (temp, humidity)
    # 也可以省略括号
    # return temp, humidity


# result 是一个元组
result = measure()
print(result)

# 需要单独的处理温度或者湿度
print(result[0])
print(result[1])

# 但是准确的指定索引非常不方便
# 如果函数返回的类型是元组
# 同时希望单独的处理元组中的元素
# 可以使用多个变量，一次接受函数的返回结果
# 注意：使用多个变量接收结果时，变量的个数应该和元组中元素的个数保持一致
gl_temp, gl_humidity = measure()

print(gl_temp)
print(gl_humidity)
```

> 【!】这个叫做元组的**拆包 / 解包**

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()
```

246. 函数的返回值-03-交换两个变量的值

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
```

---

247. 函数的参数-01-在函数内部针对参数赋值不会影响外部实参

```
```

> 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('函数内部的代码')
    # 在函数内部，针对参数使用赋值语句，不会修改到外部的实参变量
    num = 100
    num_list = [1, 2, 3]
    print(num)
    print(num_list)
    print('函数执行完成')


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

---

248. 函数的参数-02-在函数内部使用方法修改可变参数会影响外部实参

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 demo(num_list):
    print('函数内部的代码')
    # 使用方法修改列表的内容
    num_list.append(9)
    print(num_list)
    print('函数执行完成')


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

---

249. 函数的参数-04-列表使用+=本质上是调用extend方法

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（即数据会发生变化）

```
def demo(num, num_list):
    print('函数开始')
    num += num
    # 列表变量使用 +
    # 本质上是在调用列表的 extend 方法
    num_list += num_list
    # num_list.extend(num_list)
    print(num)
    print(num_list)
    print('函数完成')


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

---

250. 缺省参数-01-回顾列表的排序方法明确缺省参数的概念及作用

- 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_list = [6, 3, 9]

# 默认按照升序排序 - 升序排序情况可能会多！
# gl_list.sort()

# 如果需要降序排序，需要执行 reverse 参数
gl_list.sort(reverse=True)

print(gl_list)
```

`reverse` 参数就是默认参数，默认值是 `False`

---

251. 缺省参数-02-指定函数缺省参数的默认值

- 在参数后使用赋值语句，可以指定参数的缺省值

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！

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

    :param name: 班上同学的姓名
    :param gender: True 男生 False 女生
    :return:
    """

    gender_text = '男生'
    if not gender:
        gender_text = '女生'
    # 这里其实可以用if else的三元表达式
    # gender_text = '男生' if gender else '女生'

    print('%s 是 %s' % (name, gender_text))


# 假设：班上的同学，男生居多！
# 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')
print_info('小美', False)
```

---

252. 缺省参数-03-缺省参数的注意事项

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！

> 【!】其实如果是按顺序的话，是不用指定参数名的。

如果新加入了一个参数，没有文档字符串，那么可以把光标放在函数名或者参数名上，用 `Alt + Enter`，再一次回车即可。

```
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
    :return:
    """

    gender_text = '男生'
    if not gender:
        gender_text = '女生'
    # 这里其实可以用if else的三元表达式
    # gender_text = '男生' if gender else '女生'

    print('[%s]%s 是 %s' % (title, name, gender_text))


# 假设：班上的同学，男生居多！
# 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')
print_info('小美', gender=False)
```

---

253. 多值参数-01-定义及作用

> 有一种叫法叫做**可变长参数**。

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**

---

254. 多值参数-02-数字累加演练

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
    print(args)
    num = sum(args)
```

```
    # 或者采用循环遍历累加
    # for n in args:
    #     num += n

    return num


result = sum_numbers(1, 2, 3, 4, 5)
# 如果不用*args，使用元组接收函数，不是很直观
# 因为会有两个括号
# result = sum_numbers((1, 2, 3, 4, 5))
print(result)
```

---

255. 多值参数-03-元组和字典的拆包

- 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)


# 元组变量/字典变量
gl_nums = (1, 2, 3)
gl_dict = {'name': '小明', 'age': 18}

# demo(gl_nums, gl_dict)
demo(*gl_nums, **gl_dict)

# 为什么叫做拆包？
# 如果不使用拆包：
# demo(1, 2, 3, name='小明', age=18)
# 拆包语法可以简化元组/字典变量的传递
```

---

256. 递归-01-递归的特点及基本代码演练

> 函数调用自身的 **编程技巧** 称为递归

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

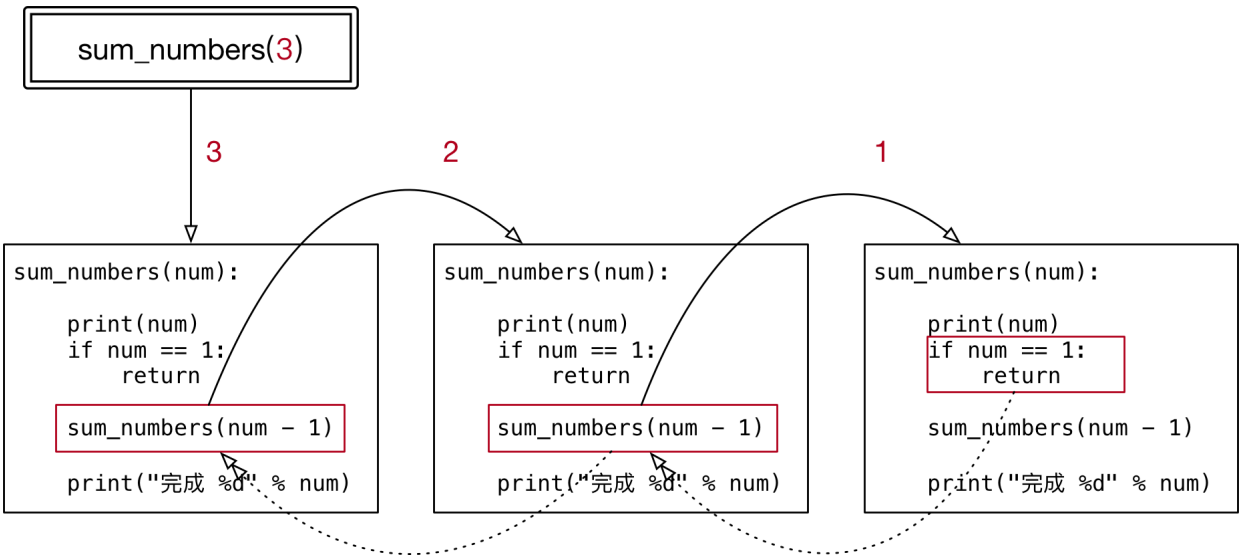
代码特点

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**！

```
def sum_number(num):
    print(num)
    # 递归的出口，当参数满足某个条件时，不再执行函数
    if num == 1:
        return
    # 自己调用自己
    sum_number(num - 1)


sum_number(3)
```

---

257. 递归-02-递归演练代码的执行流程图



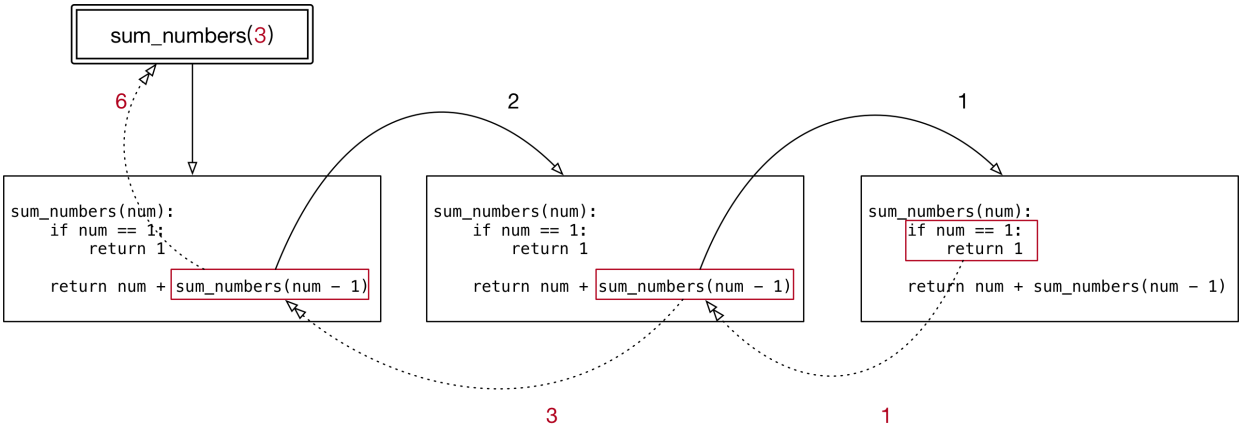
258. 递归-03-递归实现数字累加

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

```
# 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


def sum_numbers(num):
    # 1. 出口
    if num == 1:
        return 1

    # 2. 数字的累加
    # 假设 sum_numbers 能够正确的处理 1 ... num-1 的累加
    temp = sum_numbers(num - 1)
    return num + temp


result = sum_numbers(3)
print(result)
```

259. 递归-04-数字累加的执行流程图



> 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**

完成于 `201810170827`