

## Python基础知识

1. 多文件项目演练
2. python的注释和代码规范
  - 2.1 单行注释
  - 2.2 多行注释
  - 2.3 代码规范
3. 运算符
  - 3.1 算数运算符
  - 3.2 比较（关系）运算符
  - 3.3 逻辑运算符
  - 3.4 赋值运算符
  - 3.5 运算符的优先级
4. 变量的基本使用
  - 4.1 变量定义
  - 4.2 变量的类型
    - 4.3.1 不同类型变量间的计算
    - 4.3.2 变量的输入
  - 4.5 变量的格式化输出
5. 变量的命名
  - 5.1 标识符
  - 5.2 关键字
  - 5.3 变量的命名规则
6. 判断（if）语句
  - 6.1 if 判断语句基本语法
  - 6.2 else 处理条件不满足的情况
  - 6.3 逻辑运算
  - 6.4 elif 语句
  - 6.5 if 的嵌套
7. 综合应用 —— 石头剪刀布  
随机数的处理
8. while 循环基本使用
  - 8.1 while 语句基本语法
  - 8.2 break 和 continue
  - 8.3 while 循环嵌套
9. 字符串中的转义字符

## Python基础2

### 函数基础

1. 函数的快速体验
2. 函数基本使用
  - 2.1 函数的定义
  - 2.2 函数调用
  - 2.3 第一个函数演练
  - 2.4 PyCharm 的调试工具
  - 2.5 函数的文档注释
3. 函数的参数
  - 3.1 函数参数的使用
  - 3.2 参数的作用
  - 3.3 形参和实参
4. 函数的返回值
5. 函数的嵌套调用

## 6. 使用模块中的函数

### 6.1 第一个模块体验

### 6.2 模块名也是一个标识符

### 6.3 Pyc 文件（了解）

## 变量和函数进阶

### 1. 变量的引用

#### 1.1 引用的概念

#### 1.2 变量引用 的示例

#### 1.3 函数的参数和返回值的传递

### 2. 可变和不可变类型

#### 哈希 (hash)

### 3. 局部变量和全局变量

#### 3.1 局部变量

#### 3.2 全局变量

## 函数进阶

### 4. 使用元组让函数返回多个值

### 5. 函数的参数 进阶

#### 5.1 不可变和可变的参数

#### 5.2 缺省参数

#### 5.3 多值参数

### 6. 函数的递归

#### 6.1 递归函数的特点

#### 6.2 递归案例 —— 计算数字累加

### 7. LINUX 上的 Shebang 符号(#!)

# Python基础知识

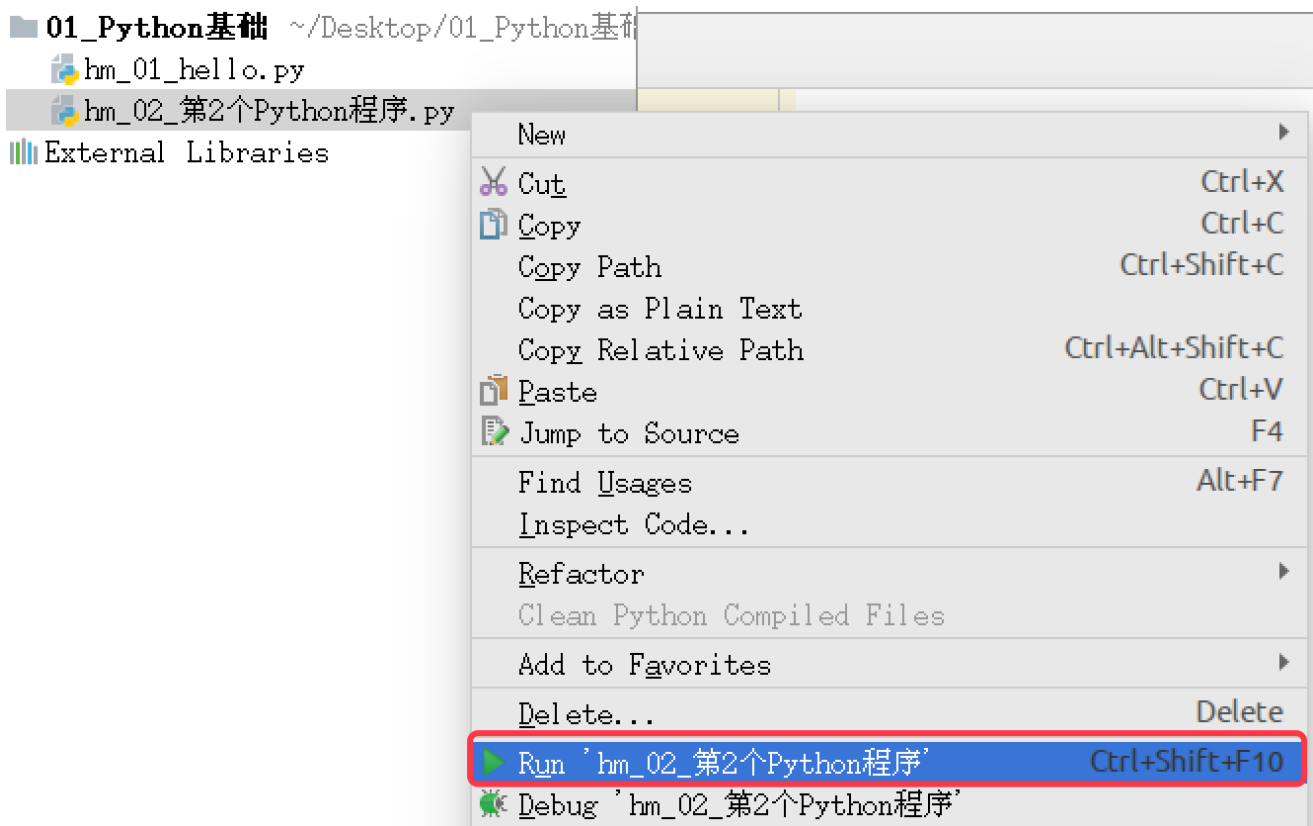
---

## 1. 多文件项目演练

---

- 开发 项目 就是开发一个 专门解决一个复杂业务功能的软件
- 通常每 一个项目 就具有一个 独立专属的目录，用于保存 所有和项目相关的文件
  - 一个项目通常会包含 很多源文件

1. 在 01\_Python基础 项目中新建一个 hm\_02\_第2个Python程序.py
2. 在 hm\_02\_第2个Python程序.py 文件中添加一句 print("hello")
3. 点击右键执行 hm\_02\_第2个Python程序.py



#### 提示

- 在 **PyCharm** 中，要想让哪一个 **Python** 程序能够执行，必须首先通过 鼠标右键的方式执行 一下
- 对于初学者而言，在一个项目中设置多个程序可以执行，是非常方便的，可以方便对不同知识点的练习和测试
- 对于商业项目而言，通常在一个项目中，只有一个 可以直接执行的 **Python** 源程序

## 2. python的注释和代码规范

### 2.1 单行注释

- 以 **#** 开头，**#** 右边的所有内容都被当做说明文字，而不是真正要执行的程序，辅助说明作用

```
# 这是第一个单行注释  
print("hello python")
```

为了保证代码的可读性，**#** 后面建议先添加一个空格，然后再编写相应的说明文字

代码后增加单行注释

- 在程序开发时，可以使用 **#** 在代码的后面（旁边）增加说明性的文字
- 但是，需要注意的是，为了保证代码的可读性，注释和代码之间 至少要有 两个空格

```
print("hello python") # 输出 `hello python`
```

### 2.2 多行注释

- 如果希望编写的 注释信息很多，一行无法显示，就可以使用多行注释
- 要在 Python 程序中使用多行注释，可以用 一对 连续的 三个 引号(单引号和双引号都可以)

```
"""
这是一个多行注释
在多行注释之间，可以写很多很多的内容.....
"""

print("hello python")
```

## 2.3 代码规范

- `Python` 官方提供有一系列 PEP（Python Enhancement Proposals）文档
- 其中第 8 篇文档专门针对 **Python** 的代码格式 给出了建议，也就是俗称的 **PEP 8**
- 文档地址: <https://www.python.org/dev/peps/pep-0008/>
- 谷歌有对应的中文文档: [http://zh-google-styleguide.readthedocs.io/en/latest/google-python-styleguide/python\\_style\\_rules/](http://zh-google-styleguide.readthedocs.io/en/latest/google-python-styleguide/python_style_rules/)

## 3. 运算符

### 3.1 算数运算符

- 算数运算符是 运算符的一种
- 是完成基本的算术运算使用的符号，用来处理四则运算

运算符	描述	实例
+	加	10 + 20 = 30
-	减	10 - 20 = -10
*	乘	10 * 20 = 200
/	除	10 / 20 = 0.5
//	取整除	返回除法的整数部分（商）9 // 2 输出结果 4
%	取余数	返回除法的余数 9 % 2 = 1
**	幂	又称次方、乘方，2 ** 3 = 8

- 在 Python 中 `*` 运算符还可以用于字符串，计算结果就是字符串重复指定次数的结果

```
In [1]: "-" * 50
Out[1]: '-----'
```

#### 算数运算符的优先级

- 先乘除后加减，同级运算符是 从左至右 计算，可用 `()` 调整计算的优先级

运算符	描述	优先级
**	幂 (最高优先级)	高
* / % //	乘、除、取余数、取整除	中
+ -	加法、减法	低

### 3.2 比较（关系）运算符

运算符	描述
==	检查两个操作数的值是否 相等，如果是，则条件成立，返回 True
!=	检查两个操作数的值是否 不相等，如果是，则条件成立，返回 True
>	检查左操作数的值是否 大于 右操作数的值，如果是，则条件成立，返回 True
<	检查左操作数的值是否 小于 右操作数的值，如果是，则条件成立，返回 True
>=	检查左操作数的值是否 大于或等于 右操作数的值，如果是，则条件成立，返回 True
<=	检查左操作数的值是否 小于或等于 右操作数的值，如果是，则条件成立，返回 True

Python 2.x 中判断 不等于 还可以使用 `<>` 运算符

`!=` 在 Python 2.x 中同样可以用来判断 不等于

### 3.3 逻辑运算符

运算符	逻辑表达式	描述
and	x and y	只有 x 和 y 的值都为 True，才会返回 True 否则只要 x 或者 y 有一个值为 False，就返回 False
or	x or y	只要 x 或者 y 有一个值为 True，就返回 True 只有 x 和 y 的值都为 False，才会返回 False
not	not x	如果 x 为 True，返回 False 如果 x 为 False，返回 True

### 3.4 赋值运算符

- 在 Python 中，使用 `=` 可以给变量赋值
- 在算术运算时，为了简化代码的编写，Python 还提供了一系列的与 算术运算符 对应的 赋值运算符
- 注意：赋值运算符中间不能使用空格

运算符	描述	实例
=	简单的赋值运算符	<code>c = a + b</code> 将 <code>a + b</code> 的运算结果赋值为 <code>c</code>
+=	加法赋值运算符	<code>c += a</code> 等效于 <code>c = c + a</code>
-=	减法赋值运算符	<code>c -= a</code> 等效于 <code>c = c - a</code>
*=	乘法赋值运算符	<code>c *= a</code> 等效于 <code>c = c * a</code>
/=	除法赋值运算符	<code>c /= a</code> 等效于 <code>c = c / a</code>
//=	取整除赋值运算符	<code>c //= a</code> 等效于 <code>c = c // a</code>
%=	取模(余数)赋值运算符	<code>c %= a</code> 等效于 <code>c = c % a</code>
**=	幂赋值运算符	<code>c = a</code> 等效于 <code>c = c a</code>

### 3.5 运算符的优先级

- 以下表格的算数优先级由高到最低顺序排列

运算符	描述
**	幂(最高优先级)
* / % //	乘、除、取余数、取整除
+ -	加法、减法
<= < > >=	比较运算符
== !=	等于运算符
= %= /= //= -= += *= **=	赋值运算符
not or and	逻辑运算符

## 4.变量的基本使用

### 4.1 变量定义

- 在 Python 中，每个变量 在使用前都必须赋值，变量 赋值以后 该变量 才会被创建
- 等号(=)用来给变量赋值
  - `=` 左边是变量名
  - `=` 右边是存储在变量中的值

变量名 = 值

变量定义之后，后续就可以直接使用了

## 4.2 变量的类型

- 在内存中创建一个变量，会包括：  
名称、保存的数据、存储数据的类型、地址（标示）
- 在 `Python` 中定义变量是 不需要指定类型（在其他很多高级语言中都需要）
- 数据类型可以分为 数字型 和 非数字型
- 数字型
  - 整型 (`int`)
  - 浮点型 (`float`)
  - 布尔型 (`bool`)
    - 真 `True` 非 0 数 —— 非零即真
    - 假 `False` `0`
  - 复数型 (`complex`)
    - 主要用于科学计算，例如：平面场问题、波动问题、电感电容等问题
- 非数字型
  - 字符串
  - 列表
  - 元组
  - 字典

提示：在 `Python 2.x` 中，整数 根据保存数值的长度还分为：

- `int`（整数）
- `long`（长整数）
- 使用 `type` 函数可以查看一个变量的类型

```
In [1]: type(name)
```

### 4.3.1 不同类型变量间的计算

#### 1) 数字型变量之间可以直接计算

- 在 `Python` 中，两个数字型变量是可以直接进行 算数运算的
- 如果变量是 `bool` 型，在计算时
  - `True` 对应的数字是 `1`
  - `False` 对应的数字是 `0`

#### 2) 字符串变量之间使用 `+` 拼接字符串

- 在 `Python` 中，字符串之间可以使用 `+` 拼接生成新的字符串

```
In [1]: first_name = "三"
In [2]: last_name = "张"
In [3]: first_name + last_name
Out[3]: '三张'
```

3) 字符串变量可以和整数使用 `*` 重复拼接相同的字符串

```
In [1]: "-" * 50
Out[1]: '-----'
```

4) 数字型变量 和 字符串 之间 不能进行其他计算

```
In [1]: first_name = "zhang"
In [2]: x = 10
In [3]: x + first_name
-----
TypeError: unsupported operand type(s) for +: 'int' and 'str'
类型错误: `+` 不支持的操作类型: `int` 和 `str`
```

### 4.3.2 变量的输入

- 所谓 输入，就是 用代码 获取 用户通过 键盘 输入的信息
- 在 Python 中，如果要获取用户在 键盘 上的输入信息，需要使用到 `input` 函数

函数	说明
<code>print(x)</code>	将 x 输出到控制台
<code>type(x)</code>	查看 x 的变量类型

- 在 Python 中可以使用 `input` 函数从键盘等待用户的输入
- 用户输入的 任何内容 Python 都认为是一个 字符串

```
字符串变量 = input("提示信息: ")
```

#### 类型转换函数

函数	说明
<code>int(x)</code>	将 x 转换为一个整数
<code>float(x)</code>	将 x 转换到一个浮点数

定义 一个 浮点变量 接收用户输入的同时，就使用 `float` 函数进行转换



```
price = float(input("请输入价格:"))
```

## 2.5 变量的格式化输出

苹果单价 9.00 元 / 斤，购买了 5.00 斤，需要支付 45.00 元

- 使用 `print` 函数输出 格式化内容
- `%` 被称为 格式化操作符，用于处理字符串格式
  - 包含 `%` 的字符串，被称为 格式化字符串
  - `%` 和不同的 字符 连用，不同类型的数据 需要使用 不同的格式化字符

格式化字符	含义
<code>%s</code>	字符串
<code>%d</code>	有符号十进制整数， <code>%06d</code> 表示输出的整数显示位数，不足的地方使用 <code>0</code> 补全
<code>%f</code>	浮点数， <code>%.2f</code> 表示小数点后只显示两位
<code>%%</code>	输出 <code>%</code>

```
print("格式化字符串" % 变量1)
```

```
print("格式化字符串" % (变量1, 变量2...))
```

```
print("我的名字叫 %s，请多多关照！" % name)
```

```
print("我的学号是 %06d" % student_no)
```

```
print("苹果单价 %.02f 元 / 斤，购买 %.02f 斤，需要支付 %.02f 元" % (price, weight, money))
```

```
print("数据比例是 %.02f%%" % (scale * 100))
```

### 练习 —— 个人名片

#### 需求

- 在控制台依次提示用户输入：姓名、公司、职位、电话、邮箱
- 按照以下格式输出：

```
*****
公司名称

姓名（职位）

电话：电话
邮箱：邮箱
*****
```

#### 代码：

```
"""
在控制台依次提示用户输入：姓名、公司、职位、电话、电子邮箱
"""

name = input("请输入姓名：")
company = input("请输入公司：")
title = input("请输入职位：")
phone = input("请输入电话：")
email = input("请输入邮箱：")

print("*" * 50)
print(company)
print()
print("%s (%s)" % (name, title))
print()
print("电话： %s" % phone)
print("邮箱： %s" % email)
print("*" * 50)
```

## 5. 变量的命名

### 5.1 标识符

标识符就是程序员定义的 变量名、函数名

- 标识符可以由 字母、下划线 和 数字 组成
- 不能以数字开头
- 不能与关键字重名

### 5.2 关键字

- 关键字 就是在 `Python` 内部已经使用的标识符
- 关键字 具有特殊的功能和含义
- 开发者 不允许定义和关键字相同的名字的标识符

通过以下命令可以查看 `Python` 中的关键字

```
In [1]: import keyword
In [2]: print(keyword.kwlist)
```

- `import` 关键字 可以导入一个“工具包”
- 在 `Python` 中不同的工具包，提供有不同的工具

### 5.3 变量的命名规则

命名规则 可以被视为一种 惯例，并无绝对与强制 目的是为了 增加代码的识别和可读性

注意 `Python` 中的 标识符 是 区分大小写的

1. 在定义变量时，`=` 的左右应该各保留一个空格

2. 在 `Python` 中，可以按照以下方式命名

1. 每个单词都使用小写字母
2. 单词与单词之间使用 `_` 下划线 连接
  - 例如: `first_name`、`last_name`、`qq_number`、`qq_password`

驼峰命名法

- 小驼峰式命名法
  - 第一个单词以小写字母开始，后续单词的首字母大写。例如: `firstName`、`lastName`
- 大驼峰式命名法
  - 每一个单词的首字母都采用大写字母。例如: `FirstName`、`LastName`、`CamelCase`

## 6. 判断（if）语句

### 6.1 if 判断语句基本语法

在 `Python` 中，`if` 语句 就是用来进行判断的，格式如下：

```
if 要判断的条件:
    条件成立时，要做的事情
.....
```

注意：代码的缩进为一个 `tab` 键，或者 **4** 个空格 —— 建议使用空格

- 在 `Python` 开发中，`Tab` 和空格不要混用！

```
age = 18
# if 语句以及缩进部分的代码是一个完整的代码块
if age >= 18:
    print("可以进网吧嗨皮.....")
# 3. 思考！ - 无论条件是否满足都会执行
print("这句代码什么时候执行?")
```

注意：

- `if` 语句以及缩进部分是一个 完整的代码块

### 6.2 else 处理条件不满足的情况

`else`，格式如下：

```
if 要判断的条件:
    条件成立时，要做的事情
.....
else:
    条件不成立时，要做的事情
.....
```

注意：

- `if` 和 `else` 语句以及各自的缩进部分共同是一个完整的代码块

```
age = int(input("今年多大了? "))
# if 语句以及缩进部分的代码是一个完整的语法块
if age >= 18:
    print("可以进网吧嗨皮.....")
else:
    print("你还没长大，应该回家写作业！")
print("这句代码什么时候执行?")
```

## 6.3 逻辑运算

- **and**

条件1 and 条件2

与 / 并且,两个条件同时满足，返回 `True` ,只要有一个不满足，就返回 `False`

- **or**

条件1 or 条件2

或 / 或者,两个条件只要有一个满足，返回 `True` ,两个条件都不满足，返回 `False`

- **not**

not 条件

- 非 / 不是

## 6.4 elif 语句

- 在开发中，使用 `if` 可以判断条件
- 使用 `else` 可以处理条件不成立的情况
- 但是，如果希望再增加一些条件，条件不同，需要执行的代码也不同时，就可以使用 `elif`

```
if 条件1:
    条件1满足执行的代码
    .....
elif 条件2:
    条件2满足时，执行的代码
    .....
elif 条件3:
    条件3满足时，执行的代码
    .....
else:
    以上条件都不满足时，执行的代码
    .....
```

- 对比逻辑运算符的代码

```
if 条件1 and 条件2:
    条件1满足 并且 条件2满足 执行的代码
    .....
```

注意

1. `elif` 和 `else` 都必须和 `if` 联合使用，而不能单独使用
2. 可以将 `if`、`elif` 和 `else` 以及各自缩进的代码，看成一个完整的代码块

## 6.5 `if` 的嵌套

- `if` 的嵌套 的应用场景就是：在之前条件满足的前提下，再增加额外的判断
- `if` 的嵌套 的语法格式，除了缩进之外 和之前的没有区别

```
if 条件 1:
    条件 1 满足执行的代码
    .....
    if 条件 1 基础上的条件 2:
        条件 2 满足时，执行的代码
        .....
    else:
        条件 2 不满足时，执行的代码
else:
    条件1 不满足时，执行的代码
    .....
```

```

has_ticket = True
knife_length = 20

if has_ticket:
    print("有车票，可以开始安检...")
    if knife_length >= 20:
        print("不允许携带 %d 厘米长的刀上车" % knife_length)
    else:
        print("安检通过，祝您旅途愉快.....")
else:
    print("大哥，您要先买票啊")

```

## 7. 综合应用 —— 石头剪刀布

```

# 从控制台输入要出的拳 —— 石头（1） / 剪刀（2） / 布（3）
player = int(input("请出拳 石头（1） / 剪刀（2） / 布（3）："))

# 电脑 随机 出拳 - 假定电脑永远出石头
computer = 1

# 比较胜负
# 如果条件判断的内容太长，可以在最外侧的条件增加一对大括号
# 再在每一个条件之间，使用回车，PyCharm 可以自动增加 8 个空格
if ((player == 1 and computer == 2) or
    (player == 2 and computer == 3) or
    (player == 3 and computer == 1)):

    print("噢耶!!! 电脑弱爆了!!! ")
elif player == computer:
    print("心有灵犀，再来一盘! ")
else:
    print("不行，我要和你决战到天亮! ")

```

## 随机数的处理

- 在 Python 中，要使用随机数，首先需要导入 随机数 的 模块 —— “工具包”

```
import random
```

- 导入模块后，可以直接在 模块名称 后面敲一个 `.`，然后按 `Tab` 键，会提示该模块中包含的所有函数
- `random.randint(a, b)`，返回 `[a, b]` 之间的整数，包含 `a` 和 `b`
- 例如：

```

random.randint(12, 20) # 生成的随机数n: 12 <= n <= 20
random.randint(20, 20) # 结果永远是 20

random.randint(20, 10) # 该语句是错误的，下限必须小于上限

```

## 8. while 循环基本使用

### 8.1 while 语句基本语法

初始条件设置 — 通常是重复执行的 计数器

while 条件(判断 计数器 是否达到 目标次数):

条件满足时, 做的事情1

条件满足时, 做的事情2

条件满足时, 做的事情3

...(省略)...

处理条件(计数器 + 1)

注意:

- while 语句以及缩进部分是一个 完整的代码块

### 8.2 break 和 continue

break 和 continue 是专门在循环中使用的关键字

- break 某一条件满足时, 退出循环, 不再执行后续重复的代码
- continue 某一条件满足时, 不执行后续重复的代码, 还会循环,

break 和 continue 只针对 当前所在循环 有效

- break
  - 在循环过程中, 如果 某一个条件满足后, 不 再希望 循环继续执行, 可以使用 break 退出循环
- continue
  - 在循环过程中, 如果 某一个条件满足后, 不 希望 执行循环代码, 但是又不希望退出循环, 可以使用 continue
  - 也就是: 在整个循环中, 只有某些条件, 不需要执行循环代码, 而其他条件都需要执行

- 需要注意：使用 `continue` 时，条件处理部分的代码，需要特别注意，不小心会出现 死循环

`continue` 只针对当前所在循环有效

### 8.3 `while` 循环嵌套

- `while` 嵌套就是： `while` 里面还有 `while`

```
while 条件 1:
    条件满足时，做的事情1
    条件满足时，做的事情2
    条件满足时，做的事情3
    ...(省略)...

    while 条件 2:
        条件满足时，做的事情1
        条件满足时，做的事情2
        条件满足时，做的事情3
        ...(省略)...

        处理条件 2

    处理条件 1
```

## 9. 字符串中的转义字符

- `\t` 在控制台输出一个 制表符，协助在输出文本时 垂直方向 保持对齐
- `\n` 在控制台输出一个 换行符

制表符 的功能是在不使用表格的情况下在 垂直方向 按列对齐文本

转义字符	描述
<code>\</code>	反斜杠符号
<code>\'</code>	单引号
<code>\"</code>	双引号
<code>\n</code>	换行
<code>\t</code>	横向制表符
<code>\r</code>	回车

# Python基础2

## 函数基础



# 1. 函数的快速体验

---

- 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

## 2. 函数基本使用

---

### 2.1 函数的定义

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
  - 可以由 字母、下划线 和 数字 组成
  - 不能以数字开头
  - 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
- 1. 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)
```

- 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？
  - 不能！
  - 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
  - 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
```

# 6. 使用模块中的函数

模块是 **Python** 程序架构的一个核心概念

- 模块 就好比是 工具包，要想使用这个工具包中的工具，就需要 导入 **import** 这个模块
- 每一个以扩展名 `.py` 结尾的 `Python` 源代码文件都是一个 模块

- 在模块中定义的 **全局变量**、**函数** 都是模块能够提供给外界直接使用的工具

## 6.1 第一个模块体验

### 步骤

- 新建 `hm_10_分隔线模块.py`
  - 复制 `hm_09_打印多条分隔线.py` 中的内容，最后一行 `print` 代码除外
  - 增加一个字符串变量

```
name = "黑马程序员"
```

- 新建 `hm_10_体验模块.py` 文件，并且编写以下代码：

```
import hm_10_分隔线模块

hm_10_分隔线模块.print_line("-", 80)
print(hm_10_分隔线模块.name)
```

### 体验小结

- 可以在一个 **Python** 文件中定义变量或者函数
- 然后在另外一个文件中使用 `import` 导入这个模块
- 导入之后，就可以使用 `模块名.变量` / `模块名.函数` 的方式，使用这个模块中定义的变量或者函数

模块可以让曾经编写过的代码方便地被复用！

## 6.2 模块名也是一个标识符

- 标识符可以由字母、下划线和数字组成
- 不能以数字开头
- 不能与关键字重名

注意：如果在给 Python 文件起名时，以数字开头是无法在 `PyCharm` 中通过导入这个模块的

## 6.3 Pyc 文件（了解）

`C` 是 `compiled` 编译过的意思

### 操作步骤

1. 浏览程序目录会发现一个 `__pycache__` 的目录
2. 目录下会有一个 `hm_10_分隔线模块.cpython-35.pyc` 文件，`cpython-35` 表示 `Python` 解释器的版本
3. 这个 `pyc` 文件是由 `Python` 解释器将模块的源码转换为字节码
  - `Python` 这样保存字节码是作为一种启动速度的优化

### 字节码

- `Python` 在解释源程序时是分成两个步骤的
  1. 首先处理源代码，编译生成一个二进制字节码

2. 再对 字节码 进行处理，才会生成 CPU 能够识别的 机器码

- 有了模块的字节码文件之后，下一次运行程序时，如果在 上次保存字节码之后 没有修改过源代码，Python 将会加载 .pyc 文件并跳过编译这个步骤
- 当 `Python` 重编译时，它会自动检查源文件和字节码文件的时间戳
- 如果你又修改了源代码，下次程序运行时，字节码将自动重新创建

提示：有关模块以及模块的其他导入方式，后续课程还会逐渐展开！

模块是 **Python** 程序架构的一个核心概念

## Python基础知识

1. 多文件项目演练
2. python的注释和代码规范
  - 2.1 单行注释
  - 2.2 多行注释
  - 2.3 代码规范
3. 运算符
  - 3.1 算数运算符
  - 3.2 比较（关系）运算符
  - 3.3 逻辑运算符
  - 3.4 赋值运算符
  - 3.5 运算符的优先级
4. 变量的基本使用
  - 4.1 变量定义
  - 4.2 变量的类型
    - 4.3.1 不同类型变量间的计算
    - 4.3.2 变量的输入
  - 2.5 变量的格式化输出
5. 变量的命名
  - 5.1 标识符
  - 5.2 关键字
  - 5.3 变量的命名规则
6. 判断（if）语句
  - 6.1 if 判断语句基本语法
  - 6.2 else 处理条件不满足的情况
  - 6.3 逻辑运算
  - 6.4 elif 语句
  - 6.5 `if` 的嵌套
7. 综合应用 —— 石头剪刀布  
随机数的处理
8. `while` 循环基本使用
  - 8.1 `while` 语句基本语法
  - 8.2 `break` 和 `continue`
  - 8.3 `while` 循环嵌套
9. 字符串中的转义字符

## Python基础2

### 函数基础

1. 函数的快速体验
2. 函数基本使用
  - 2.1 函数的定义
  - 2.2 函数调用
  - 2.3 第一个函数演练

2.4 PyCharm 的调试工具

2.5 函数的文档注释

### 3. 函数的参数

3.1 函数参数的使用

3.2 参数的作用

3.3 形参和实参

### 4. 函数的返回值

### 5. 函数的嵌套调用

### 6. 使用模块中的函数

6.1 第一个模块体验

6.2 模块名也是一个标识符

6.3 Pyc 文件（了解）

## 变量和函数进阶

### 1. 变量的引用

1.1 引用的概念

1.2 变量引用 的示例

1.3 函数的参数和返回值的传递

### 2. 可变和不可变类型

哈希 (hash)

### 3. 局部变量和全局变量

3.1 局部变量

3.2 全局变量

## 函数进阶

### 4. 使用元组让函数返回多个值

### 5. 函数的参数 进阶

5.1 不可变和可变的参数

5.2 缺省参数

5.3 多值参数

### 6. 函数的递归

6.1 递归函数的特点

6.2 递归案例 —— 计算数字累加

### 7. LINUX 上的 Shebang 符号(#!)

# 变量和函数进阶

## 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


- 定义一个整数变量 `a`，并且赋值为 `1`

代码	图示
<code>a = 1</code>	

- 将变量 `a` 赋值为 `2`

代码	图示
<code>a = 2</code>	

- 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
<code>b = a</code>	

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

```
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

## 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！

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

## 4. 使用元组让函数返回多个值

- 利用 元组 同时返回温度和湿度

```
def measure():  
    """返回当前的温度"""  
    temp = 39  
    wetness = 10  
    return (temp, wetness)  
  
temp, wetness = measure()
```

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

- 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
```

## 5. 函数的参数 进阶

### 5.1 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)

```

## 5.2 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)
```

### 1) 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！

### 2) 缺省参数的注意事项

- 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):
```

### 3) 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)
```

## 5.3 多值参数

### 1) 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

### 2) 多值参数案例 —— 计算任意多个数字的和

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))
```

### 3) 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)  
  
# 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)
```

## 6. 函数的递归

函数调用自身的 编程技巧 称为递归

### 6.1 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

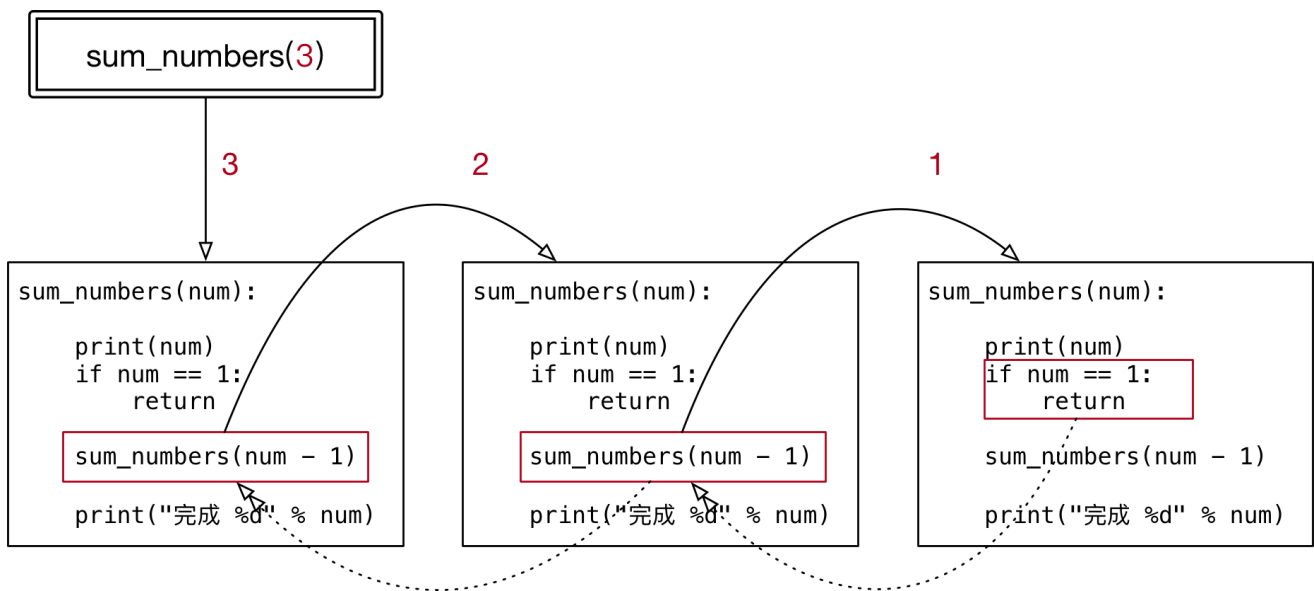
```
def sum_numbers(num):

    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)
```



## 6.2 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  $1 + 2 + \dots + \text{num}$  的结果

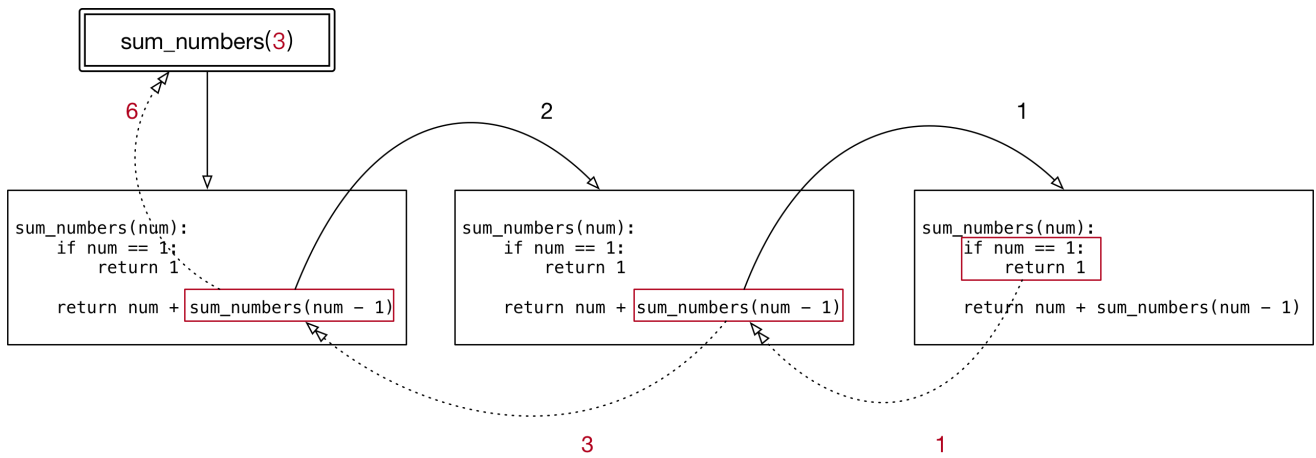
```
def sum_numbers(num):

    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

## 7. LINUX 上的 Shebang 符号(#!)

- `#!` 这个符号叫做 **Shebang** 或者 **Sha-bang**
- **Shebang** 通常在 **Unix** 系统脚本的中 第一行开头 使用
- 指明 执行这个脚本文件 的 解释程序

使用 **Shebang** 的步骤

- 1. 使用 `which` 查询 `python3` 解释器所在路径

```
$ which python3
```

- 2. 修改要运行的 主 **python** 文件，在第一行增加以下内容

```
#!/usr/bin/python3
```

- 3. 修改 主 **python** 文件 的文件权限，增加执行权限

```
$ chmod +x cards_main.py
```

- 4. 在需要时执行程序即可

```
./cards_main.py
```

