# 运算符

## 01. 算数运算符

* 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]: '----------------------------------------'

### 1.1 算数运算符的优先级

* 和数学中的运算符的优先级一致，在 Python 中进行数学计算时，同样也是：
  + **先乘除后加减**
  + 同级运算符是 **从左至右** 计算
  + 可以使用 () 调整计算的优先级
* 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 |

* 例如：
  + 2 + 3 \* 5 = 17
  + (2 + 3) \* 5 = 25
  + 2 \* 3 + 5 = 11
  + 2 \* (3 + 5) = 16

## 02. 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**

## 03. 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<br />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<br />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<br />如果 x 为 False，返回 True |

## 04. 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 | c = a + b 将 a + b 的运算结果赋值为 c |
| += | 加法赋值运算符 | c += a 等效于 c = c + a |
| -= | 减法赋值运算符 | c -= a 等效于 c = c - a |
| \*= | 乘法赋值运算符 | c \*= a 等效于 c = c \* a |
| /= | 除法赋值运算符 | c /= a 等效于 c = c / a |
| //= | 取整除赋值运算符 | c //= a 等效于 c = c // a |
| %= | 取 **模** (余数)赋值运算符 | c %= a 等效于 c = c % a |
| \*\*= | 幂赋值运算符 | c \*= a 等效于 c = c \* a |

## 05. 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 |

# 变量的基本使用

**程序就是用来处理数据的，而变量就是用来存储数据的**

## 01. 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
  + = 左边是一个变量名
  + = 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

## 02.变量的类型

* 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)

## 03.不同类型变量之间的计算

### 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

**演练步骤**

1. 定义整数 i = 10
2. 定义浮点数 f = 10.5
3. 定义布尔型 b = True

* 在 iPython 中，使用上述三个变量相互进行算术运算

### 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`

## 04.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
* 例如：去银行取钱，在 ATM 上输入密码
* 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

### 1) 关于函数

* 一个 **提前准备好的功能**(别人或者自己写的代码)，**可以直接使用**，而 **不用关心内部的细节**
* 目前已经学习过的函数

| **函数** | **说明** |
| --- | --- |
| print(x) | 将 x 输出到控制台 |
| type(x) | 查看 x 的变量类型 |

### 2) input 函数实现键盘输入

* 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**
* 语法如下：

字符串变量 = input("提示信息：")

### 3) 类型转换函数

| **函数** | **说明** |
| --- | --- |
| int(x) | 将 x 转换为一个整数 |
| float(x) | 将 x 转换到一个浮点数 |

### 4) 变量输入演练 —— 超市买苹果增强版

**需求**

* **收银员输入** 苹果的价格，单位：**元／斤**
* **收银员输入** 用户购买苹果的重量，单位：**斤**
* 计算并且 **输出** 付款金额

#### 演练方式 1

*# 1. 输入苹果单价*

price\_str = input("请输入苹果价格：")

*# 2. 要求苹果重量*

weight\_str = input("请输入苹果重量：")

*# 3. 计算金额*

*# 1> 将苹果单价转换成小数*

price = float(price\_str)

*# 2> 将苹果重量转换成小数*

weight = float(weight\_str)

*# 3> 计算付款金额*

money = price \* weight

print(money)

**提问**

1. 演练中，针对 **价格** 定义了几个变量？
   * **两个**
   * price\_str 记录用户输入的价格字符串
   * price 记录转换后的价格数值
2. **思考** —— 如果开发中，需要用户通过控制台 输入 **很多个 数字**，针对每一个数字都要定义两个变量，**方便吗**？

#### 演练方式 2 —— 买苹果改进版

1. **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:"))

* 改进后的好处：

1. 节约空间，只需要为一个变量分配空间
2. 起名字方便，不需要为中间变量起名字

**提示**

* 如果输入的不是一个数字，程序执行时会出错，有关数据转换的高级话题，后续会讲！

## 05.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

* 在 Python 中可以使用 print 函数将信息输出到控制台
* 如果希望输出文字信息的同时，**一起输出** **数据**，就需要使用到 **格式化操作符**
* % 被称为 **格式化操作符**，专门用于处理字符串中的格式
  + 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** |
| --- | --- |
| %s | 字符串 |
| %d | 有符号十进制整数，%06d 表示输出的整数显示位数，不足的地方使用 0 补全 |
| %f | 浮点数，%.2f 表示小数点后只显示两位 |
| %% | 输出 % |

* 语法格式如下：

print("格式化字符串" % 变量1)

print("格式化字符串" % (变量1, 变量2...))

### 格式化输出演练 —— 基本练习

**需求**

1. 定义字符串变量 name，输出 **我的名字叫 小明，请多多关照！**
2. 定义整数变量 student\_no，输出 **我的学号是 000001**
3. 定义小数 price、weight、money，输出 **苹果单价 9.00 元／斤，购买了 5.00 斤，需要支付 45.00 元**
4. 定义一个小数 scale，输出 **数据比例是 10.00%**

**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

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

实现代码如下：

"""

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)

# 判断（if）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
* 语法格式如下：

## 01. if

**if** 条件:

条件满足执行的代码

……

## 02. if - else

**if** 条件:

条件满足执行的代码

……

**else**:

条件都不满足时，执行的代码

## 03. if - elif - else

**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

……

## 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**

## 04. if 的嵌套

**elif** 的应用场景是：**同时** 判断 **多个条件**，所有的条件是 **平级** 的

* 在开发中，使用 if 进行条件判断，如果希望 **在条件成立的执行语句中** 再 **增加条件判断**，就可以使用 **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
* 语法格式如下：

**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

……

*# 条件 2 不满足的处理*

**else**:

条件 2 不满足时，执行的代码

*# 条件 1 不满足的处理*

**else**:

条件1 不满足时，执行的代码

……

## 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*

# 循环（while）语句

## 01. 程序的三大流程

* 在程序开发中，一共有三种流程方式：
  + **顺序** —— **从上向下**，顺序执行代码
  + **分支** —— 根据条件判断，决定执行代码的 **分支**
  + **循环** —— 让 **特定代码 重复** 执行

## 02. while 语句基本语法

* 循环的作用就是让 **指定的代码** 重复的执行
* while 循环最常用的应用场景就是 **让执行的代码** 按照 **指定的次数** **重复** 执行

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**

## 03. 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

break 和 continue 只针对 **当前所在循环** 有效

### 3.1 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

i = 0

**while** i < 10:

*# break 某一条件满足时，退出循环，不再执行后续重复的代码*

*# i == 3*

**if** i == 3:

break

print(i)

i += 1

print("over")

break 只针对当前所在循环有效

### 3.2 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

i = 0

**while** i < 10:

*# 当 i == 7 时，不希望执行需要重复执行的代码*

**if** i == 7:

*# 在使用 continue 之前，同样应该修改计数器*

*# 否则会出现死循环*

i += 1

continue

*# 重复执行的代码*

print(i)

i += 1

* 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

## 04. 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

## ****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** |
| --- | --- |
| \\ | 反斜杠符号 |
| \' | 单引号 |
| \" | 双引号 |
| \n | 换行 |
| \t | 横向制表符 |
| \r | 回车 |

# 函数基础

## 01. 函数基本使用

### 1.1 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**
   * 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**

### 1.2 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

### 1.3 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)

用 **单步执行 F8 和 F7** 观察以下代码的执行过程

* 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**)

### 1.4 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

### 1.5 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**

## 02. 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！

### 2.1 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)

### 2.2 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

### 2.3 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

## 03. 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)

## 04. 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()

## 05. 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

### 5.1 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** 方便的被 **复用**！

### 5.2 模块名也是一个标识符

* 标示符可以由 **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

### 5.3 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**