

这本《21天入门Python》源于我去年在个人公众号连载的Python零基础入门教程，现将其整理成册，以便于读者离线阅读及保存。

目录

- Day 1 一切都要从搭建环境说起
- Day 2 变量，注释，缩进，细数Python优雅风
- Day 3 切片，灵活的字符串
- Day 4 纵经千万次增删改，初心永不变
- Day 5 输入输出，字符串如影随形
- Day 6 跳动的数字，熟悉又陌生
- Day 7 小小运算符，构筑大世界
- Day 8 分支循环，效率加速器
- Day 9 简单的列表，却囊括了世间万物
- Day 10 字典，寻寻觅觅
- Day 11 元组与集合
- Day 12 函数，化繁为简
- Day 13 高阶函数，匿名函数
- Day 14 排序，不止于升降
- Day 15 异常处理
- Day 16 面向对象，站在更高的角度来思考
- Day 17 对象属性与类属性，私有属性与私有方法
- Day 18 封装、继承与多态，面向对象的三大特性
- Day 19 模块和包，站在巨人的肩膀上
- Day 20 动手打造流水账记录器，实践是检验真理的唯一标准
- Day 21 一切都要从搭建环境说起（新的开始）



今天你将学到以下内容：

- 1.为什么需要搭建环境
- 2.如何搭建环境
- 3.初探Python

”

开篇

在使用 `Python` 进行编程之前，首先需要 `搭建 Python 环境`。如果你是第一次接触 `搭建 Python 环境` 这个名词，不用担心，这个过程非常简单，你只要跟着后面的 `环境搭建` 步骤一步一步的操作就可以了。

为什么需要搭建环境

温馨提示：这一部分属于扩展的知识，与环境搭建部分是独立的，对于非计算机专业的同学来说，看不太懂也没关系，直接跳转到下一部分阅读也无妨哦

在介绍环境搭建步骤之前，还是简要说一下为什么需要搭建环境。

计算机所能识别的，其实只有两个数字，即二进制中的 `0` 和 `1`，由 `0` 和 `1` 的各种组合编写出来的代码被称为 `二进制代码`，一组有意义的二进制代码构成了一条 `指令`，这条指令被称为 `机器语言` 的一条语句。

面对 `机器语言` 中一堆堆的 `0` 和 `1`，不管是读还是写都很让人头大，于是有人在 `机器语言` 的基础上，编写了一些指令，用于替代 `机器语言` 中一堆堆的 `0` 和 `1`，这样便形成了 `汇编语言`。`汇编语言` 的每条指令与 `机器语言` 所实现的某一个功能是一一对应的，但缺点也很明显：符号繁多、无可移植性。

于是前辈们为了能“偷懒”，又开始了进一步的探索。

终于，`高级语言` 问世了！

包括 `Python` 在内的一些编程语言，比如 `C`，`C++` 等，它们都有一个共同的特性，那就是语法规范，符合人类的逻辑认知，并且与计算机硬件关系不大，从而我们可以很容易的专注于在逻辑上实现自己的想法，而无需考虑底层的内存操作如何运作。这些语言都属于同一个大类，那就是今天被广泛使用的 `高级语言`。

现在，回到正题，为什么需要搭建环境？

简单来讲，`Python` 是高级语言，而计算机只能处理包含 `0` 与 `1` 的机器语言，所以，我们需要一种“工具”，它可以把我们书写的 `Python` 代码转换为计算机所能理解并处理的机器语言。而这种“工具”，在搭建环境完成之后，便存在了。这里面包含 `Python解释器` 这个“工具”，它可以将我们书写的每一行代码“翻译”成计算机所能理解的东西。因此，你在使用 `Python` 进行优雅的编程之前，必须先搭建好 `Python` 环境。

如何搭建环境

搭建环境，无非就是下载安装软件，配置环境变量等步骤。

如果你跟随接下来的步骤进行操作，可以让搭建过程更简单。

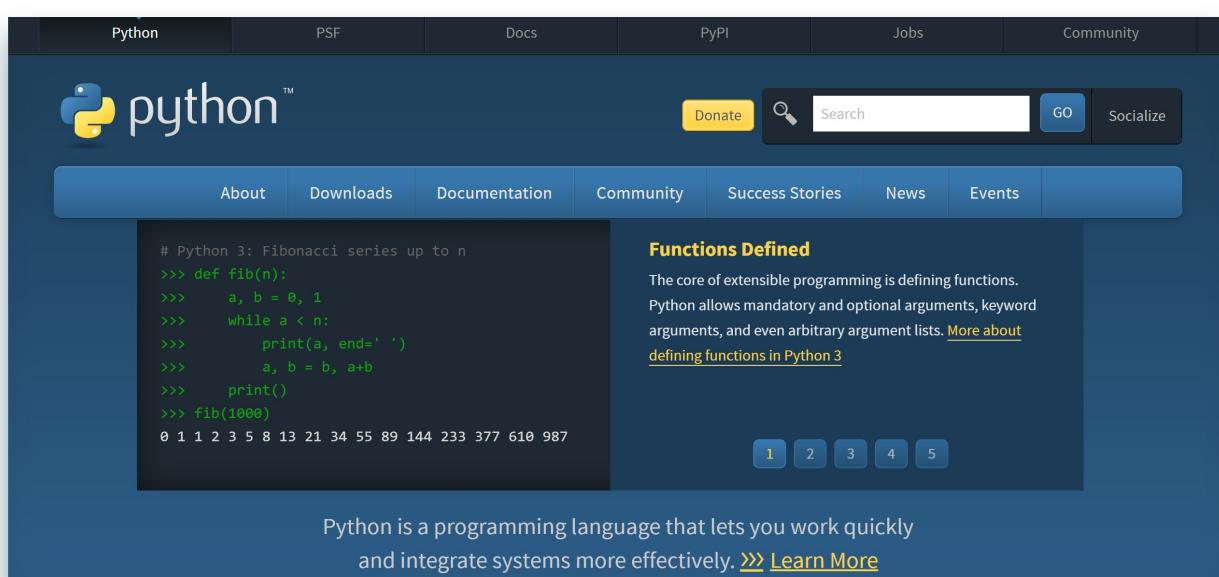
Let's get started~~~

对于初学者而言，最容易上手的便是从 `Python` 官网进行下载以及后续操作，具体步骤如下：

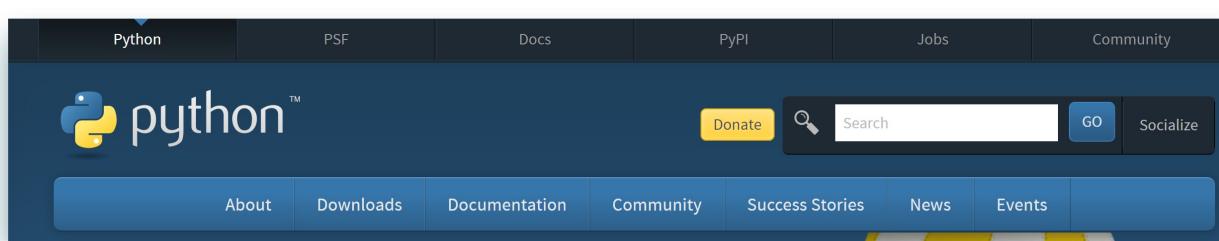
(以下操作均在 `windows` 系统上进行)

1. 访问 `Python` 官网 <https://www.python.org/>

进入到这个界面

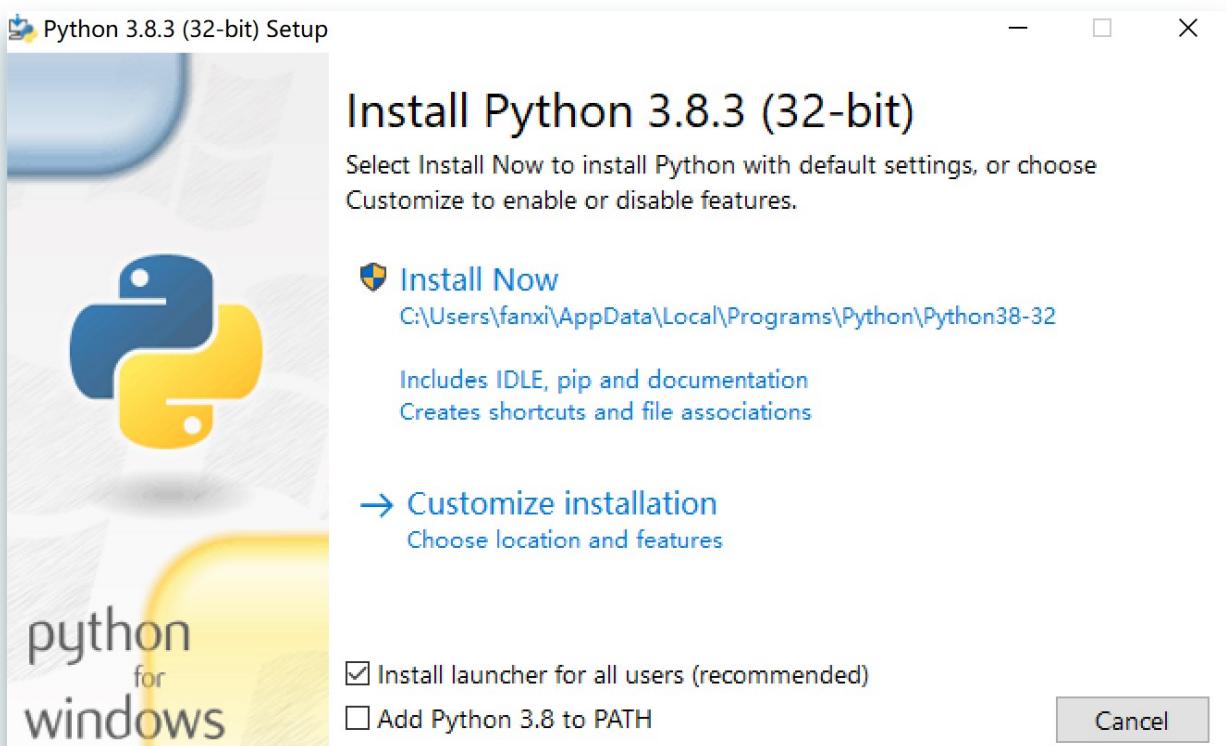


2. 点击 `Downloads`，来到这个界面

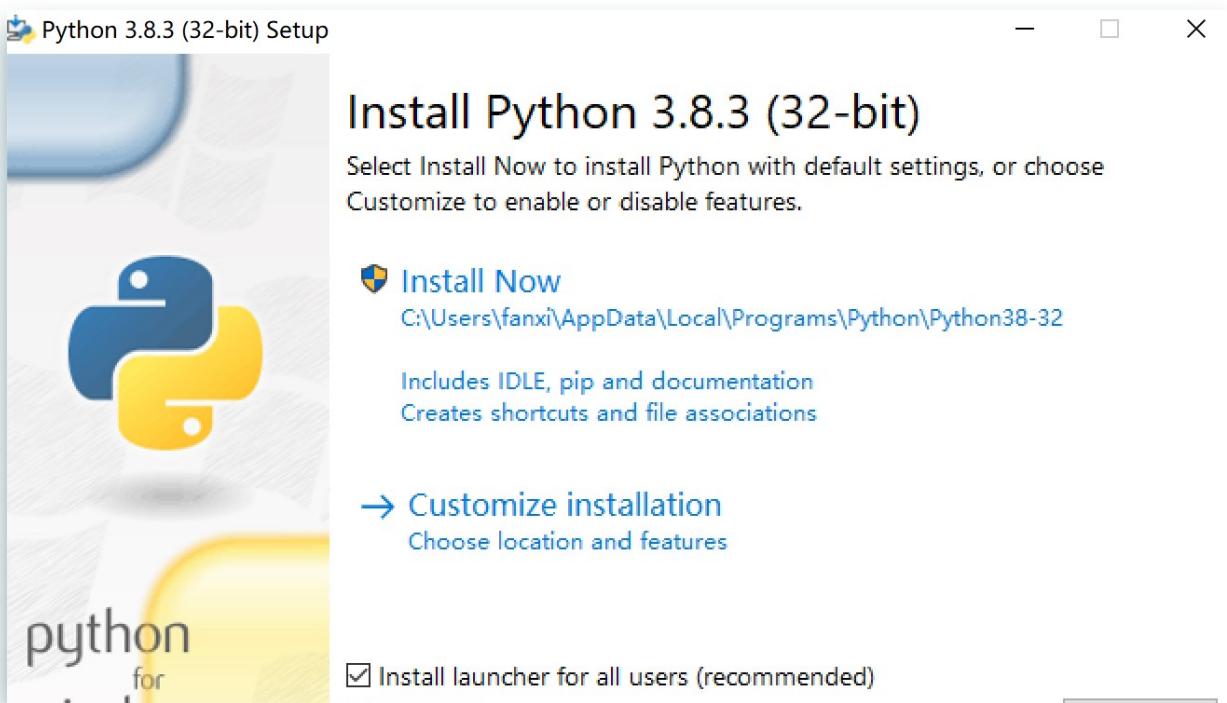




3. 点击 `Download Python 3.8.3`，可以直接下载最新的与你的系统适配的 windows 版本。下载完成之后得到一个 `.exe` 文件，双击进行安装，你会来到这一界面



4. 在进行下一步之前，你需要将 `Add Python 3.8 to PATH` 勾选上，这样可以免去手动设置环境变量的麻烦，就像下图这样



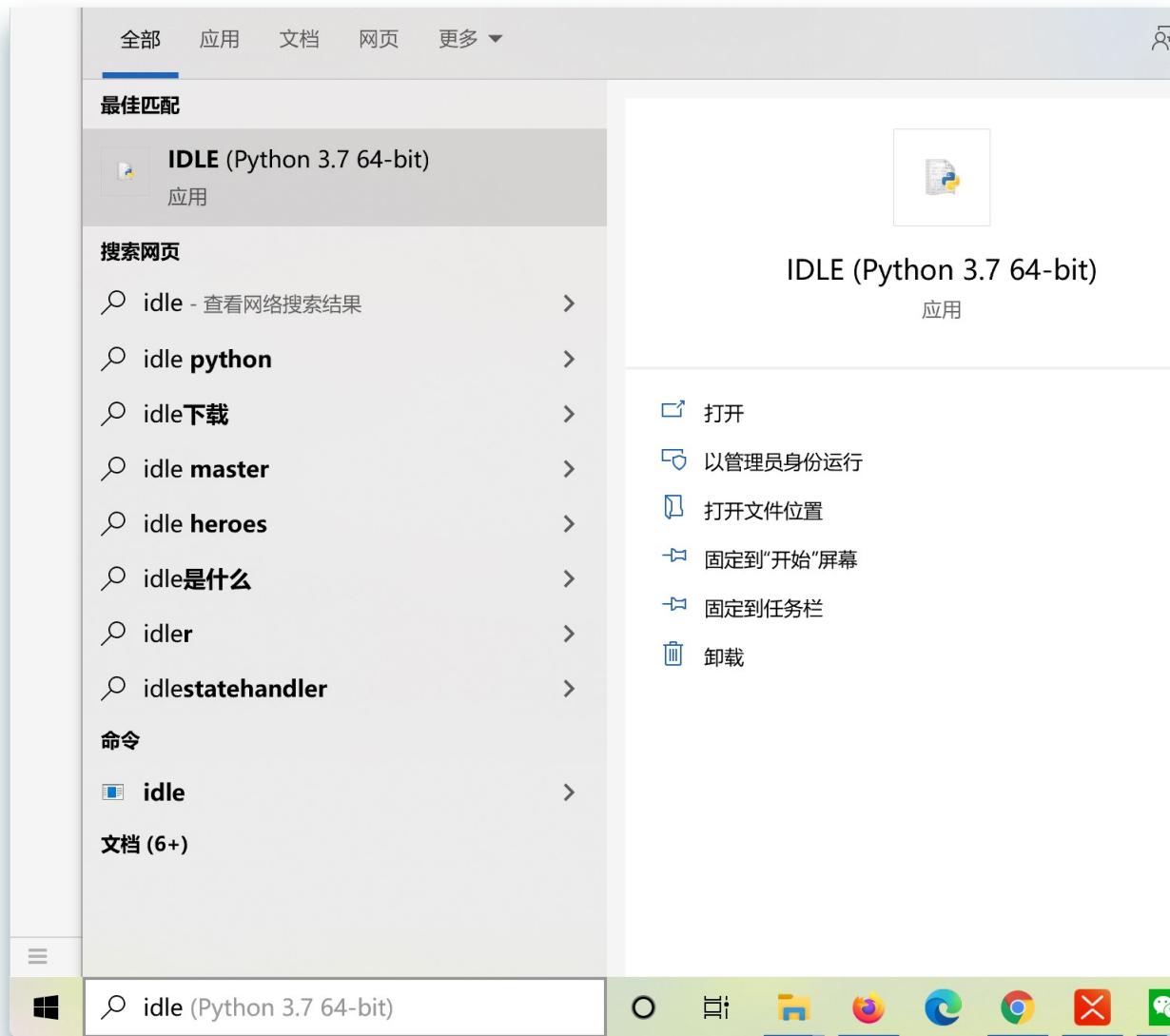
5. 现在，点击 **Install Now**，等待安装完成即可。

至此，你已经配置好了Python环境。

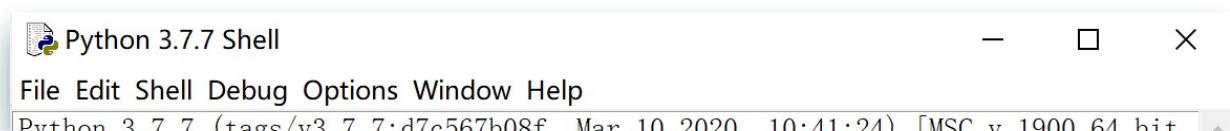
初探Python

在完成环境搭建之后，就可以打开 **IDLE** (你可能不知道它是什么，没关系，将它看成一个普通的软件即可)进行编程啦！

在你完成上一部分的步骤之后，**IDLE** 就已经附带着安装到你的计算机中了，只不过并没有在桌面上创建快捷方式，所以需要手动搜索，具体操作如下图所示：(ps：我是以前安装的3.7版本，你的应该会显示3.8版本)



点击 **打开**，便来到了这一界面：

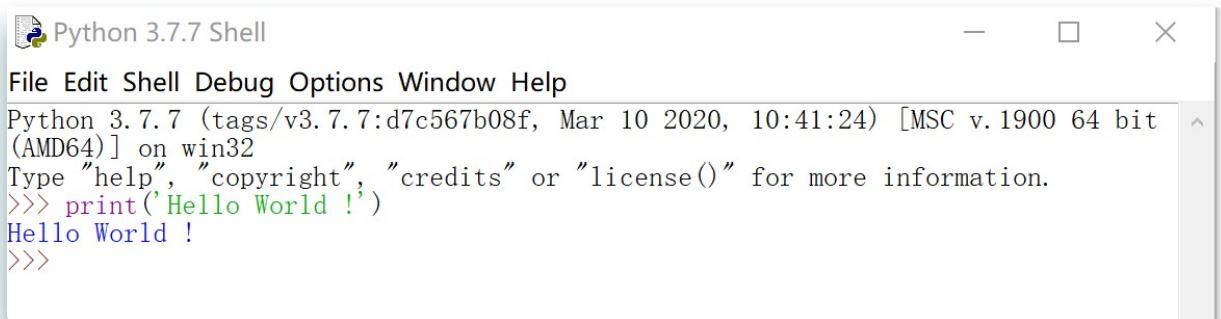


```
Python 3.7.7 (tags/v3.7.7:d7c567b08f, Mar 10 2020, 10:41:24) [MSC v.1900 64 bit  
 (AMD64)] on win32  
Type "help", "copyright", "credits" or "license()" for more information.  
>>> |
```

在 `>>>` 上面的那一段文字是关于版本的介绍以及一些提示信息，你大可不必理会。

在 `>>>` 后面的光标会闪动，代表你可以在 `>>>` 后面编写你的 `Python` 语句。

在编写完成之后，敲一个大大的 `回车`，便可以即时看到运行结果。下面是一个小栗子：



```
Python 3.7.7 Shell  
File Edit Shell Debug Options Window Help  
Python 3.7.7 (tags/v3.7.7:d7c567b08f, Mar 10 2020, 10:41:24) [MSC v.1900 64 bit  
(AMD64)] on win32  
Type "help", "copyright", "credits" or "license()" for more information.  
>>> print('Hello World !')  
Hello World !  
>>>
```

看，我们的 `Python` 语句被执行了！

这里的`**print()**`具有打印功能，也就是将结果输出显示到屏幕上，具体用法将会在后面介绍，你现在只要记住它具有打印输出功能就可以了。

闪动的光标也移到了下一行，代表你再次输入时需要从第二个 `>>>` 之后开始。

之后也是同样的操作，你只要记住，每次输入都要在最下面一行的 `>>>` 之后开始。

以上的编程方式是 `交互式` 的，通俗来讲，就是你写一句代码，只要按下 `回车`，`Python` 就会立即执行你的这一句代码。

这种方式虽然方便，但如果你希望将所写的代码保存起来与朋友共享，这种方式会表现的极其糟糕。下面举一个栗子(如果你是完全零基础，可能代码会看不懂，没关系，在后面会讲到，这里只是拿来举例说明)：

```
>>> x=3  
>>> y=4  
>>> print(x+y)  
7
```

这是使用交互式方式写的代码，功能是实现两个变量之间的加法运算。现在，你希望将这段代码保存起来并分享给另一个同学，你会怎么做？

在介绍怎么做之前，首先要了解的是，在 **IDLE** 中，除了使用交互式的 **命令行** 之外，还有另一种编程方式，那就是创建一个 **脚本文件**（你已经知道，图像文件的后缀名为 **.jpg** 或 **.png** 等，同样，所谓的脚本文件，其实就是一个文本文件，只不过后缀名是 **.py**），在脚本文件中写完全部代码之后，运行该脚本文件即可。

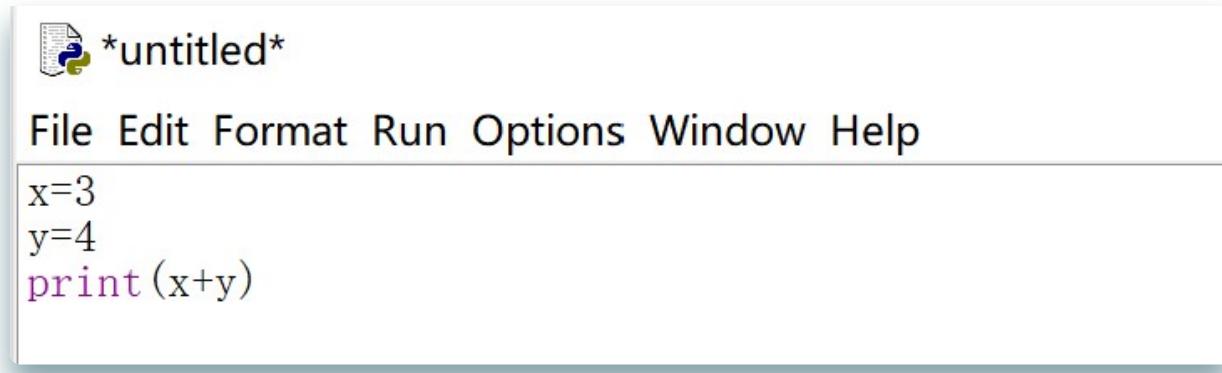
【注1】脚本文件的创建方式有两种，其一是点击 **File->New File**，其二是使用快捷键 **Ctrl+N** **【注2】**脚本文件的运行方式也有两种，其一是点击 **Run->Run Module**，其二是使用快捷键 **F5**

现在可以来讨论下怎么做了。我知道，有的同学可能会创建一个脚本文件，然后直接将命令行中的代码复制到该脚本文件中，就像下面这样：



```
>>> x=3
>>> y=4
>>> print(x+y)
```

但是，你直接将 **>>>** 也一起复制过来了，这是不对的，你应该删去它们，得到下面的结果：



```
x=3
y=4
print(x+y)
```

然后就可以将其保存到计算机硬盘上并分享给你的朋友了。

但是如果你的代码有成百上千行，逐个删除 **>>>** 是一个非常恼人的操作，所以，我们一般都会在脚本文件中进行代码的书写。

当然，在学习一些简单的操作时，交互式的命令行不失为一种简单的方法，因为它提供了即时的输出，可以让人很清楚的明白每一行代码究竟做了什么事。

总结一下，那就是：

进行简单的操作，并且不需要保存所写代码时，首选 **交互式** 方式；

如果希望长期保存代码，那就选择在脚本文件中书写，并保存该脚本文件即可。

Day 2 变量，注释，缩进，细数Python优雅风



今天你将学到以下内容：

- 1. Python中的变量
- 2. 变量的命名规则
- 3. Python中的注释
- 4. Python特有的风格：缩进

”

开篇

在之后的几期文章中，你将会陆续学习到 **Python** 的六个标准数据类型



不过在此之前，有一些先导内容需要掌握，所以这一期就先来介绍一下这些内容。

Python中的变量

所谓 **变量**，顾名思义，就是“会变化的量”。

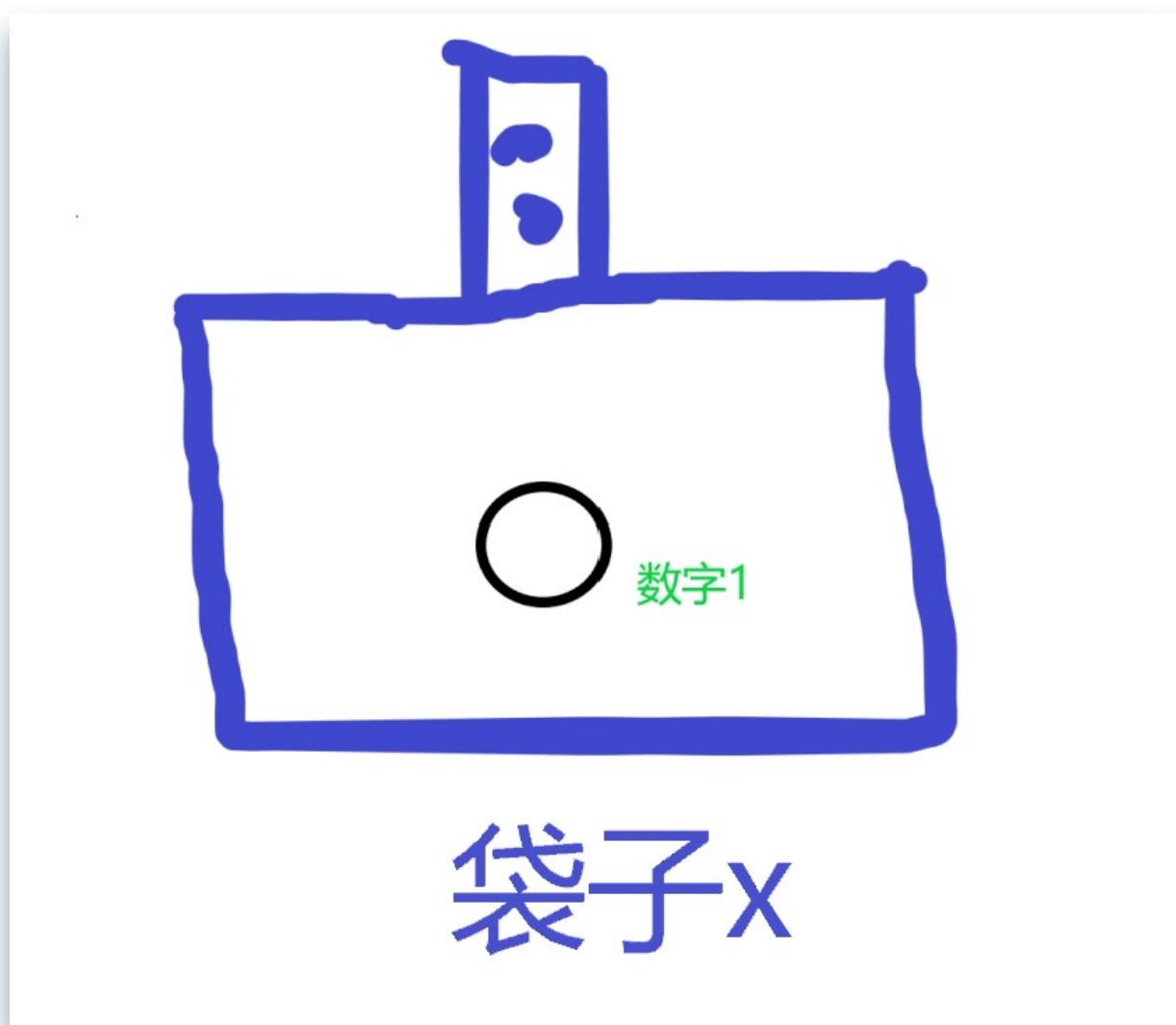
在 **Python** 中，你可以把变量当作一个袋子，袋子里面可以装水果，可以装手机，也可以装书籍。

现在，将 **数字1** 装入一个叫做 **x** 的袋子，可以这样写：

x=1

这一句代码的作用就是定义了一个变量，名字叫做 `x`，同时将 `数字1` 赋值给变量 `x`。用上面的袋子的例子来解释，就是将 `数字1` 装入了袋子 `x`，就像下图这样

(【注】在 `Python` 中，`=` 是赋值号，用于将 `* = *` 右边的值赋值给 `* = *` 左边的变量，并不是数学中的等于号。)



当你不想让这个袋子 `x` 装 `数字1`，而是想装入一个新的 `数字2` 时，可以直接这样写

x=2

此时，袋子 `x` 中所存放的，就只有 `数字2` 了，如下图所示





袋子x

装入新的 `数字2` 的过程，就相当于修改变量内容的过程，这里就体现了变量中的变。

【下面这一段属于扩展知识，目前仅作了解即可，在之后会有详细讲解】你可能会疑惑，之前的 `数字1` 去哪了？其实，`数字1` 被新来的 `数字2` 给覆盖掉了，变量 `x` 是保存的始终是最新的值。这里如果深入探究一下，那就涉及到了可变数据类型和不可变数据类型。数字是不可变数据类型，如果是直接修改变量 `x` 的值（这里的值就是 `数字1`），则会将原先保存在变量内的值（`数字1`）抹去。如果在修改变量 `* x *` 之前，将变量的值赋值给另外一个新的变量 `y`，然后修改变量 `x` 的值为 `数字2`，则变量 `y` 的值仍是 `* 数字1`，变量 `* x *` 的值变为了 `数字2` **，这一过程和我们的直观感觉是一样的。但是，如果变量是一个可变数据类型，比如列表，那情况就不一样了。具体内容将在后面介绍。 *

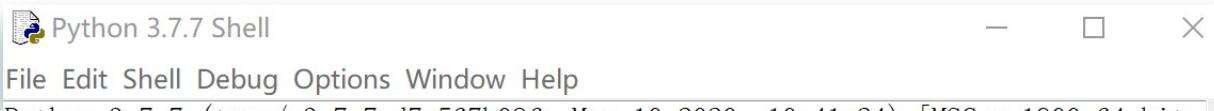
如果之前学习过 * `C` 语言，你应该知道，在 `C` * 语言中，变量在使用之前必须先声明，就像下面这样：

```
int x;x=5
```

对比来看，* `Python` 就简洁很多，一行代码 `x=5` 就能搞定！原因就在于 `Python` * 中的变量不需要先声明再赋值，而是在赋值的同时定义变量。

总之，需要明确的是，在 `Python` 中，变量只有在被赋值时才能被定义。

举个反例：如果直接写一个变量名，比如 `z`，而不给它赋值时，则会报错



```
Python 3.7.7 (tags/v3.7.7:d7c567b08f, Mar 10 2020, 10:41:24) [MSC v.1900 64 bit  
(AMD64)] on win32  
Type "help", "copyright", "credits" or "license()" for more information.  
>>> z  
Traceback (most recent call last):  
  File "<pyshell#0>", line 1, in <module>  
    z  
NameError: name 'z' is not defined  
>>>
```

`Python` 的变量是 **动态类型** 的。所谓 **动态类型**，就是说变量中保存的数据类型是不固定的，可能上一秒保存的是数字，下一秒通过修改就能表示字符串了。

正如之前所讲，在 `Python` 中，变量就是个袋子，袋子里面可以装水果，可以装手机，也可以装书籍。

变量的命名规则

`Python` 中的变量名只能包含 **数字**，**字母** 以及 **下划线**，且**不能以数字开头**。

变量名可以是一个单独的 **字母**，可以是一个单独的 **下划线**，也可以是 **数字**、**字母** 和 **下划线** 的组合，但绝对不能是一个单独的数字。

有点摸不着头脑？没关系，举个栗子就清楚啦：

```
>>> x2=3  
>>> _p=8  
>>>_=8  
>>> _3=99  
>>> 3s=123  
SyntaxError: invalid syntax  
>>> 3=1234  
SyntaxError: can't assign to literal  
>>> |
```

你看，以数字 `3` 开头的 `3s`，单独一个数字 `3` 都报错了，所以它们都不能作为变量名。

还有两点要说明的是：

- `Python` 是区分大小写的。变量 `a` 和变量 `A` 是两个完全不同的变量。
- 不能使用 `Python` 中的 **关键字**（也称 **保留字**）作为变量名，比如 `if`，`for`，`break` 等，这些关键字在后面都会讲到。

Python中的注释

在 `Python` 中，用 `#` 来表明这一行是一条注释语句，还是举个栗子：

```
#print('Hello World !')
```

运行上述代码，不会有任何输出，因为被注释掉的语句会被 `Python` 忽略掉。

```
>>> #print('Hello World !')  
>>> |
```

如果你想要将一整段文字注释起来，画风可能是这样的：

untitled

File Edit Format Run Options Window Help

```
#你好  
#欢迎来到南极  
#让我们一起学Pyhon吧！
```

事实上，`Python` 还提供了另外一种注释方法，那就是使用一对****三引号（单引号 / 双引号均可）可以直接将整段文字注释掉，于是画风由红变绿：

untitled

File Edit Format Run Options Window Help

```
"""  
你好  
欢迎来到南极  
让我们一起学Pyhon吧！  
"""
```

Python特有的风格：缩进

如果你以前接触过 `C 语言`，那么你会惊叹 `Python` 是多么的简洁。

简洁的原因，除了能够使用简单的语法实现强大的功能等之外，还有就是其 `缩进` 的风格。

不同于 `C` 等其他编程语言，`Python` 使用 `缩进` 来表示一个代码块，同时需要用到符号 `:`，这个符号表示缩进的开始，在之后学习的条件语句，循环语句以及函数等知识中都会遇到它。总之，`缩进` 将贯穿编程始终，所以这里就不再举例了。

今天的内容并不多，相信读到这里的你已经吸收了这些知识。

让我们以 *Python* 之禅 结束今天的文章吧（你可以在 `IDLE` 中输入 `import this` 来获取这首 *Python* 之禅）

The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit. Simple is better than complex.
Complex is better than complicated. Flat is better than nested.
Sparse is better than dense. Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never. Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

Day 3 切片，灵活的字符串



今天你将学到以下内容：

1. 什么是字符串
2. 如何选取字符串中的某些元素（重点）
3. 更多字符串操作

”

开篇

之前已经提到过，在 `Python` 中，一共有六大数据类型

Python的标准数据类型

数字 字符串str 列表list 元组tuple 字典dict 集合set

本期就开始学习 `Python` 的六个标准数据类型中的字符串。

什么是字符串

在 `Python` 中，字符串是用**一对引号（单引号/双引号均可）**包裹起来的一串字符，比如之前你见到的 '`Hello World !`' 便是一个字符串。

下面再给出几个字符串的栗子：

```
>>> s='Life is short and U need Python'  
>>> s1='All right'  
>>> s2="I love Coding!"  
>>> |
```

`Python` 提供了 `type()` 函数用于查询某个变量的数据类型，在上图中，数据类型名字后面的英文就是该数据类型所对应的 `type()` 函数返回值，`()` 内填写的是你要查看的变量名。

接着上面的栗子，现在我要查看一下这些变量所保存元素的数据类型(不能你说是字符串就是字符串吧，我要亲自动手验证一下，嘿嘿)

```
>>> s='Life is short and U need Python'  
>>> s1='All right'  
>>> s2="I love Coding!"  
>>> type(s)  
<class 'str'>  
>>> type(s1)  
<class 'str'>  
>>> type(s2)  
<class 'str'>  
>>> |
```

验证完毕，返回值是* `str` *，是字符串无误！

注意，一个单个的字符也是字符串：

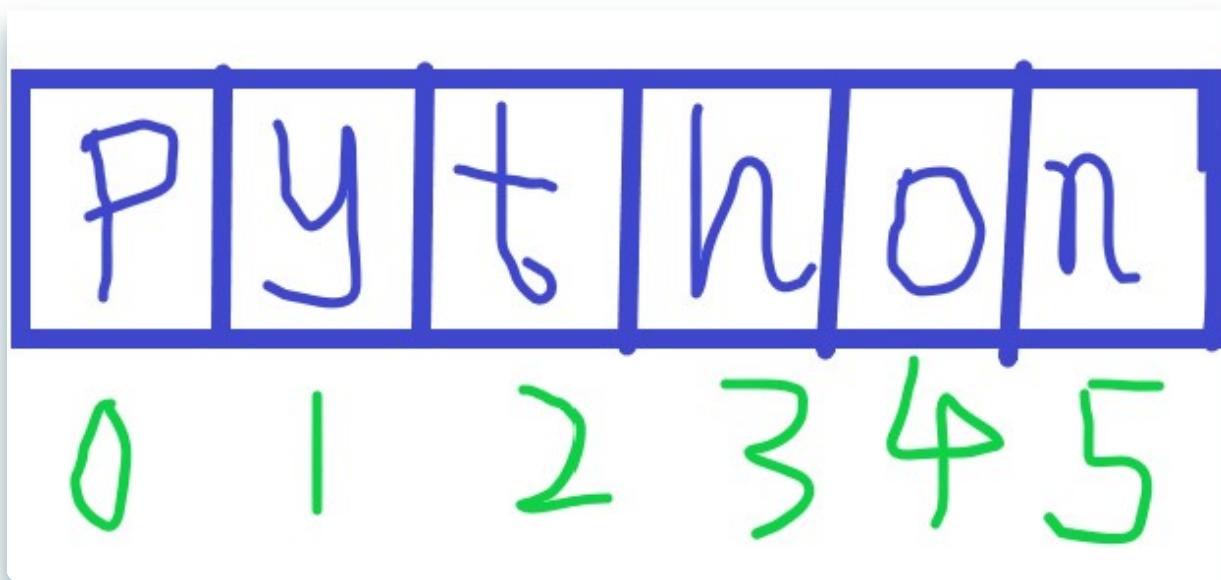
```
>>> ch='u'  
>>> type(ch)  
<class 'str'>
```

你可能会好奇，这里出现的 `函数`，`返回值` 和 `class` 是什么鬼。不要着急，后面会讲到，现在你只要会用 `type()` 来查看一个变量的数据类型就足够了

如何选取字符串中的某些元素（重点）

首先要说明的是，Python 的 `下标`（也称索引）是从 `0` 开始的，这和大部分编程语言是一致的。

下面这张图清晰的展示了 `下标` 与元素位置的关系：字符串 '`Python`' 的长度是 `6`，下标的最大值是 `5`，正好差 `1`。



现在，将字符串 '`Python`' 存入变量 `s`：

```
s='Python'
```

如果想要获取中的某一个字符，直接使用 `s[下标]` 即可：

```
>>> s='Python'  
>>> s[0]  
'P'  
>>> s[1]  
'y'  
>>> s[2]  
't'  
>>> s[3]  
'h'  
>>> s[4]  
'o'  
>>> s[5]
```

```
>>> s[5]  
, n
```

注意，如果你给的下标值超出了字符串的下标最大值，则会报错，就像下面这样：

```
>>> s='Python'  
>>> s[6]  
Traceback (most recent call last):  
  File "<pyshell#37>", line 1, in <module>  
    s[6]  
IndexError: string index out of range  
>>>
```

知道了如何选取字符串中的某一特定下标处的元素之后，你可能又会想：我能不能一次选取好几个元素呢？

当然可以！

Python 的 **切片** 便是用来做这件事情的。

切片 语法如下：

```
s[start:end:step]
```

解释一下：**s** 是一个存储了字符串的变量名，**start** 和 **end** 分别代表了 **切片** 的 **开始位置下标** 和 **结束位置下标+1**，**step** 是 **步长**，如果不写 **step**，那就用默认值 **1**。

(【注】Python 规定，**step** 不能为 0，这一点记住就好，无需深究。)

下面的栗子可以让你更好的理解上面所说：

```
>>> s='Great Python'  
>>> s[0:3] #start=0,end=3,所以区间为[0,2]  
'Gre'
```

这里，从下标为 **0** 的位置元素开始，一直到下标为 **2** 的位置结束，由于我们没有写 **step**，所以使用默认值 **1**，也就是一步一步地向前走（规范化地表述是：从 **start** 开始，以 **1** 为步长，向 **end** 方向扫描）。

最终将下标区间 **[0, 2]** 位置的所有元素选取出来。

一个比较好的记忆方式是：start和end组成的区间是左闭右开的。

如果我们更改了步长，又会怎样？看下面的栗子：

```
>>> s="Great Python"  
>>> s[0:9:2]  
'GetPt'
```

这一次设置了步长为 2，选取的下标范围是 0 到 8。



`s [0:9:2]` 所做的事情如下(结合上图来理解)：

第一次选取下标为`0`处的元素`G`接下来走`2`步，到达下标`2`

第二次选取下标为`2`处的元素`e`接下来走`2`步，到达下标`4`

第三次选取下标为`4`处的元素`t`接下来走`2`步，到达下标`6`

第四次选取下标为`6`处的元素`P`接下来走`2`步，到达下标`8`

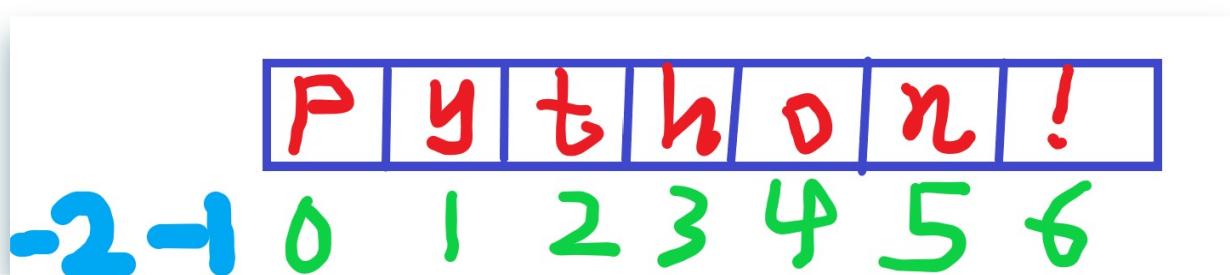
第五次选取下标为`8`处的元素`t`由于已经走到了区间最右侧，所以不再继续走。

至此，选取结束，返回由 G 、 e 、 t 、 P 、 t 组成的字符串 'GetPt' 。

相信通过以上详细的讲解，你已经学会了使用字符串中简单的 [切片](#) 语法。

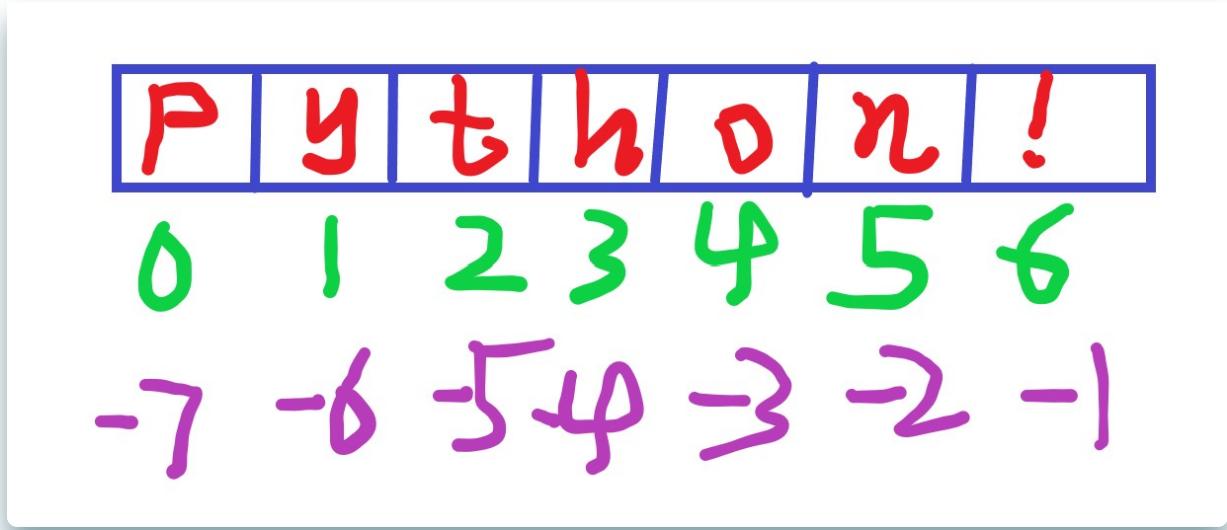
还有一件更神奇的事情：[下标可以为负数！](#)

你可能会问：下标为负数的话不是没有对应元素了嘛？就像下面这样：



事实不是这样的！

负数索引，指的是从最后一个元素开始，往前数（也就是倒着数，比如-1代表倒数第一个）。所以下面这张图才是正确的：



继续之前的栗子（仔细看注释哦）：

```
>>> s='Python'  
>>> s[-1]#选取倒数第一个元素'n'  
>>> s[-2]#选取倒数第二个元素'o'  
>>> s[-123]#不存在倒数第123个元素，所以报错  
Traceback (most recent call last): File "<pyshell#69>", line 1, in <module> s[-123]IndexError: string i
```

对于 `切片` 操作同理，这里给出几个栗子：

```
>>> s="Great Python"  
>>> s[-3:-1]#注意区间左闭右开  
'ho'  
>>> s[-5:-1:2]#注意区间左闭右开  
'yh'
```

如果仔细观察，你会发现，上面所有的栗子中，`start` 都是小于 `end` 的，那能不能出现 `start > end` 呢？

答案是可以！但同时需要将步长 `step` 设置为负，否则选取到的结果一定是空！

```
>>> s='Graet Python'
```

```
>>> s[4:1]#未设置步长，采用默认值1（正），结果为空''  
>>> s[-1:-4]#未设置步长，采用默认值1（正），结果为空  
''  
>>> s[-1:-4:2]#步长为2（正），结果为空  
''  
>>> s[4:1:-1]#设置了步长为-1（负）  
'tea'  
>>> s[-1:-6:-2]#设置了步长为-2（负）  
'nhy'
```

更多字符串操作

定义一个字符串：

```
s='Hello World'
```

**1.* * *求字符串长度：使用 `len()` 函数

```
>>> s='Hello World'  
>>> len(s)  
11
```

**2.* * *选取全部元素

```
#方法1  
>>> s[0:len(s)]  
'Hello World'  
  
#方法2：`切片`特有的操作  
>>> s[:]  
'Hello World'
```

**3.* * *选取倒序的全部元素

```
#`切片`特有的操作  
>>> s[::-1]  
'olleH dlrow'
```

附

本期内容到这里就结束了，关于字符串的用法还没有讲完，剩余内容将在下期介绍。

在结束之前，简单介绍一下 `print()`。

`print()` 用于正如其名，用于输出结果到屏幕上。

将 `'Hello World!'` 输出到屏幕上，只需将这个字符串放进 `()` 内，运行即可：

```
>>> print('Hello World!')
Hello World!
```

`()` 内也可以放入变量：

```
>>> s='Hello World!'
>>> print(s)
Hello World!
```

Day 4 纵经千万次增删改，初心永不变



今天你将学到以下内容：

1. 字符串中常用的查找操作
2. 字符串中的替换操作
3. 字符串的拼接
4. 字符串中常用的分割操作
5. 字符串的格式化操作
6. 删除字符串中的特定内容
7. 字符串中的大小写转换方法

”

开篇

上一期介绍了字符串及其切片用法，相信你已经掌握。

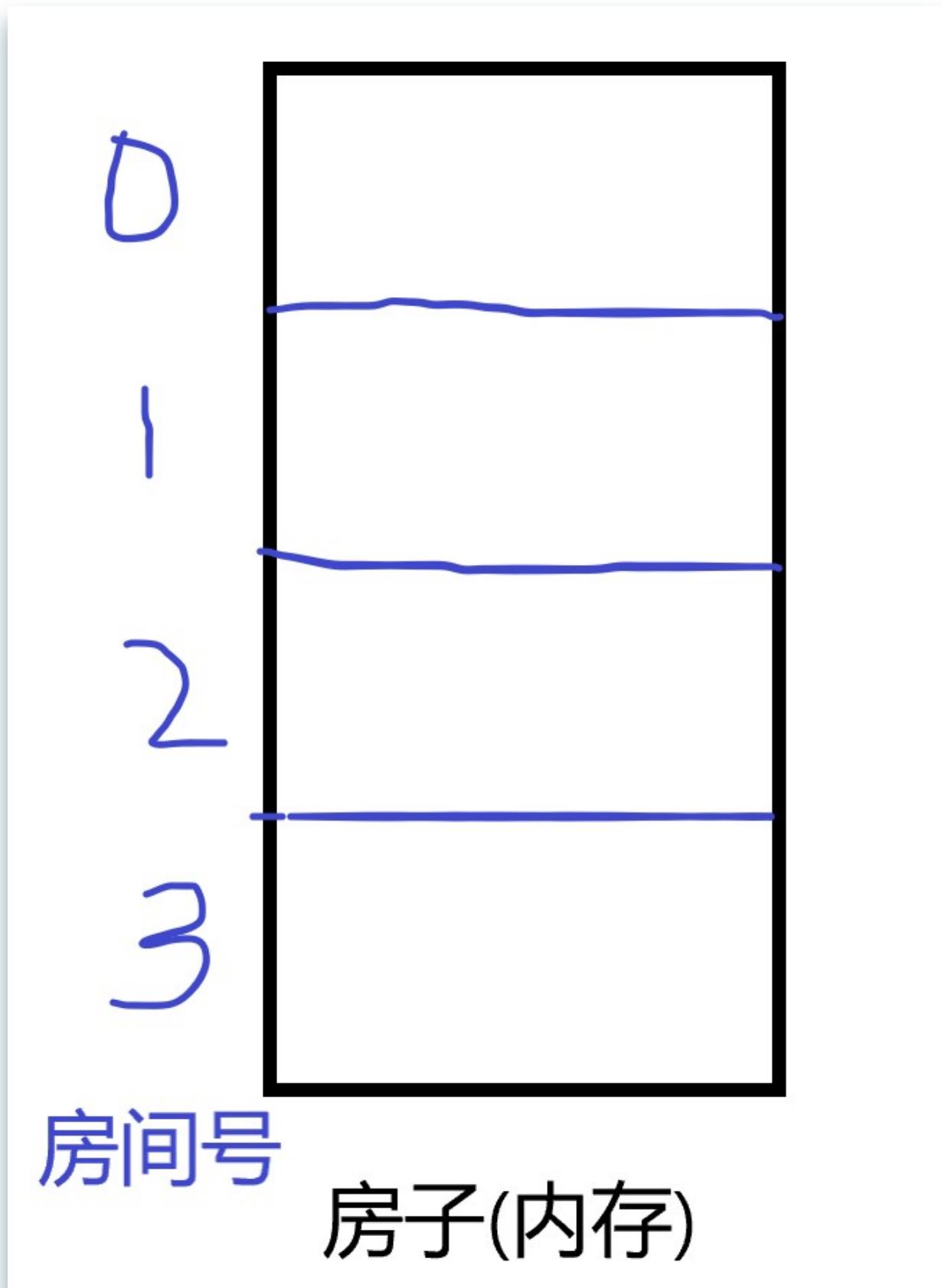
那么这一期，我们就来学习一下字符串的 `增删改查` 等一系列方法，正是因为有了这些方法，你才能像呼吸一样自然地操作字符串。

需要指出的是，字符串是不可变类型，也就是说，只要一个字符串确定了，那么任何操作都不能修改该字符串。

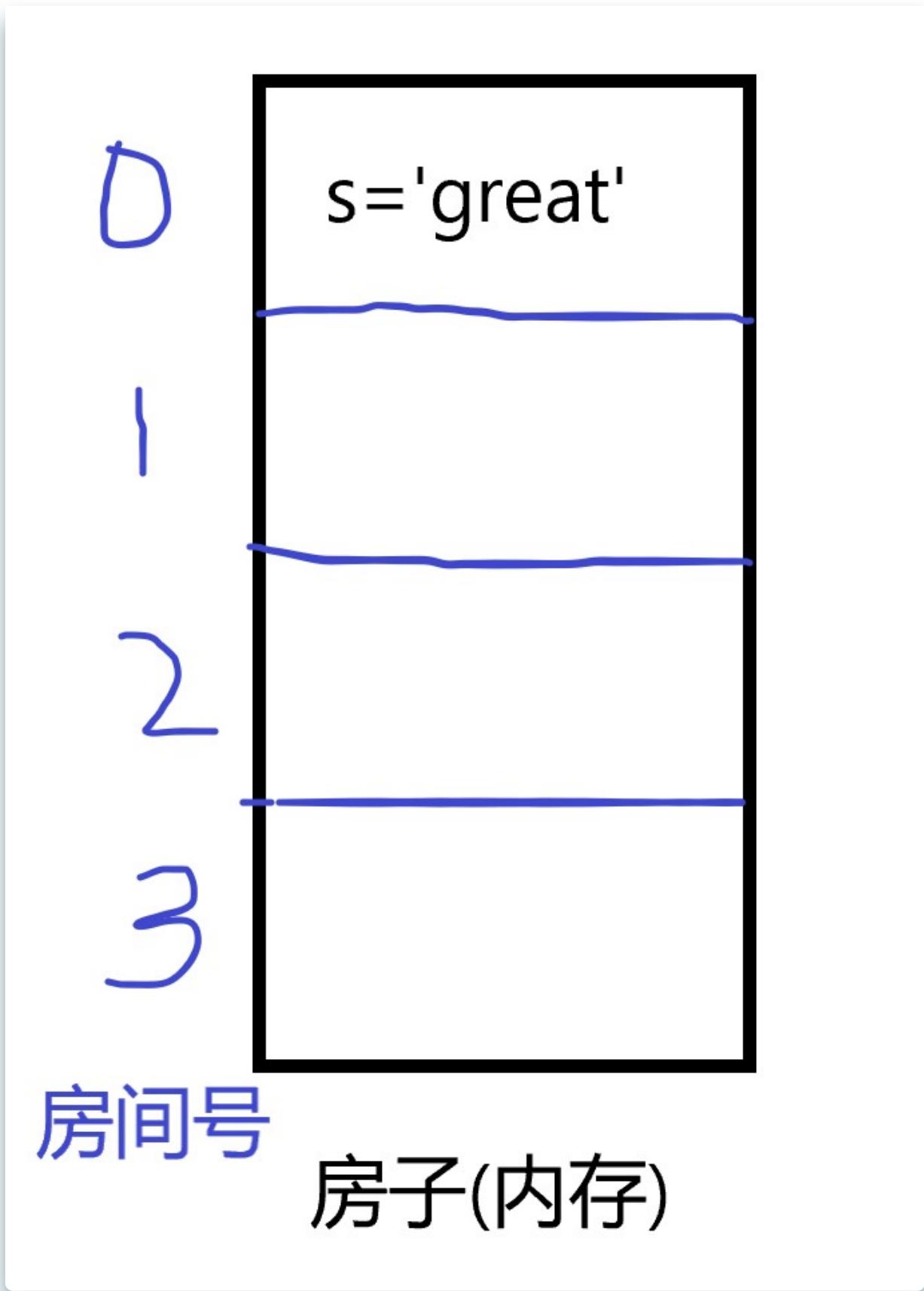
此时的你，可能会很疑惑：既然字符串是不可变类型，那何来增删改这些操作呢？

解答这个问题的过程涉及到了内存，下面我将用画图的方式解释：

将内存看作一个大房子，房子中有许多小房间（地址单元），将它们分别编号为 0，1，2，3。



在运行 `s = 'great'` 这一句代码时，会给变量 `s` 分配一个房间，假设分配的房间号是 `0`：



现在，我想要修改变量 `s` 中的内容，由 `great` 改成 `gre!`。由于字符串是不可变数据类型，所以不能直接在变量 `s` 所在的小房间（地址单元）内进行修改。

正确的做法是新开一个小房间，假如新开的小房间是 1 号，则首先会将 s 的前 3 个字符复制到房间 1 (这里用到了切片，你还记得吗)，然后在后面加一个 !。这样，房间 1 内所保存的就是我们想要的结果了。

以上文字描述过程对应代码如下：

```
>>> s='great'  
>>> new=s[0:3]+'!'# +用于拼接两个字符串，本期马上就讲到  
>>> print(new)  
gre!
```



看，原来的字符串变量 `s` 和修改后的字符串变量 `new` 在两个不同的房间。可以肯定的是，字符串不可不变类型，是不会变的；修改后的结果保存在了新的地址单元内。

`Python` 中提供了 `id()` 函数，用于查看两个内存中的地址单元是否是同一个（这里，你就理解为查看两个小房间的房间号是不是同一个）：

```
>>> id(s)
2121690232688
>>> id(new)
2121689684592
```

结果很明显了，房间号完全不一样！

总结来说：当对字符串执行 增删改 等操作时，最终的结果会保存到一个新的内存地址单元内（这里就是指小房间 1），而原来的内存地址单元（这里指的是小房间 0）所保存的东西是不会发生任何改变的。

此时，可能又会有人问：如果代码改成这个样子呢？

```
>>> s='great'
>>> s=s[0:3]+'!'
>>> print(s)
gre!
```

你看吧，`s` 变了！我只要将修改后的字符串的名字也用 `s` 表示，字符串就变了呀！

答：非也。你把 变量 和 变量所保存的内容 搞混了。还记得 变量 那节的知识吗？ 变量 是可以变化的，所以你这里将新的字符串也用原来的变量名来表示这一操作，实际上 修改的是变量中的内容，原来的字符串中的内容被新的内容“抹去”了。你可以抹去我的存在，但我初心不改---by 字符串。想起来了吧~

好，说了这么多，终于可以开始今天的主要内容了。上面的内容对于零基础的同学来说，如果很难理解也没关系，只要记住字符串是不可变的，继续往下看即可。待到时机成熟，自然就理解了。

接下来的内容，都是一些记忆性的东西，以后用到时再来查也可以。

不过，强烈建议初学者跟着敲一遍，能记多少记多少，这样可以加深印象，培养写代码的手感，也有助于后续内容的学习。

字符串中常用的查找操作

1. `find()`

查找字符串中某个字符串首次出现的下标，不存在则返回 `-1`

```
>>> s='abcdabcd'  
>>> s.find('c')  
2  
>>> s.find('abc')  
0  
>>> s.find('abcd')  
-1
```

上面的查找方式是默认从下标为 `0` 处开始查找的，我们也可以手动修改从哪个下标处开始查找，只需要再传入一个下标即可：

```
>>> s='abcdabcd'  
>>> s.find('a',2)#从下标为2的地方开始查找'a'  
4  
>>> s.find('abcd',1)#从下标为1的地方开始查找'abcd'  
4
```

也可以同时指定查找的开始下标和结束下标：

```
>>> s='abcdabcd'  
>>> s.find('a',1,6)  
4  
>>> s.find('a',-6,-1)  
4  
>>> s.find('a',1,8)#即使越界也不报错  
4  
>>> s.find('a',1,2333)#即使越界也不报错  
4
```

2. `index()`

和 `find()` 功能用法几乎一模一样，只是 `index()` 在查找失败时会报错，而 `find()` 返回 `-1`：

```
>>> s='abcdabcd'  
>>> s.index('a')
```

```
0  
>>> s.index('a',1)#指定查找开始位置  
4  
>>> s.index('a',1,5)#指定查找开始位置和结束位置  
4  
>>> s.index('abce')#找不到就报错  
  
Traceback (most recent call last):  
File "<pyshell#45>", line 1, in <module>  
s.index('abce')ValueError: substring not found
```

3. rfind()

和 `find()` 一样，只是 `rfind()` 是从后面往前面查（从下标大的位置开始，往下标小的位置扫描）

```
>>> s='abcdabcd'  
>>> s.rfind('d')  
7  
>>> s.rfind('d',2)#指定开始位置，这里是2，就说明查找范围是从2一直到最后的len(s)-1，只不过是从后向前查  
7  
>>> s.rfind('d',2,6)#指定开始和结束位置，也就是查找区间  
3  
>>> s.rfind('d',6,2)#指定开始和结束位置，也就是查找区间，但由于start>end，因此区间为空，自然也就找不到'd'了  
-1  
>>> s.rfind('d',-2,-5)#同上  
-1
```

4. rindex()

和 `index()` 一样，只是从后往前查询，并且找不到就报错

```
>>> s='abcdabcd'  
>>> s.rindex('b')  
5  
>>> s.rindex('b',5)  
5  
>>> s.rindex('b',1,7)  
5  
>>> s.rindex('abce')#找不到就报错
```

```
Traceback (most recent call last): File "<pyshell#70>", line 1, in <module> s.rindex('abce')ValueError:
```

字符串中的替换操作

使用 `replace()`

`replace('old str', 'new str')` 用于将字符串中的 `old str` 替换成 `new str`

```
>>> word='fantastic'  
>>> word.replace('a','A')  
'fAntAstic'
```

也可以指定最大替换次数

```
>>> word='fantastic'  
>>> word.replace('a','A',1)  
'fAntastic'
```

这里指定的最大替换次数为 `1`，因此只把第一个出现的 `a` 替换成了 `A`。

字符串的拼接

使用 `+` 即可将若干字符串拼接成一个大的字符串。

```
>>> s1='1! '  
>>> s2='2'  
>>> s3='3'  
>>> s=s1+s2+s3  
>>> print(s)  
123
```

字符串中常用的分割操作

分割操作，指的是把一个字符串分割成几个小的部分。

*【注意】*这里可能会出现列表，元组，我们还没有讲到。不过相信我，它们并不影响本节的阅读。

1. `split()`

() 里面填写分隔符，如果没有填写任何字符的话，则使用默认值(空格)``。

```
>>> s='a-b-c-d'  
>>> s.split()#没有填写的话,默认使用空格分割, 所以分不开  
['a-b-c-d']  
>>> s.split('-')#此时传入了分隔符'-'， 所以分割开了  
['a', 'b', 'c', 'd']
```

我们也可以指定分割的次数，见下栗：

```
>>> s='a-b-c-d'  
>>> s.split('-',2)#分割次数为2  
['a', 'b', 'c-d']
```

2. `rsplit()`

从后往前分割，只有在指定分割次数时才能看得见效果，其余结果和 `split()` 是一样的

```
>>> s='a-b-c-d'  
>>> s.rsplit()#和`split()`结果一样  
['a-b-c-d']  
  
>>> s.rsplit('-',1)#和`split()`结果一样  
['a', 'b', 'c', 'd']  
  
>>> s.rsplit('-',2)#指定了最大分割次数为2, 所以第一个'-'没有被处理  
['a-b', 'c', 'd']
```

3. `partition`

将字符串分割为 3 个部分，可用于分割一个文件的文件名和后缀名，直接上栗子

```
>>> file_name='xxx.py'  
>>> x=file_name.partition('.').#按照'.'来分割  
>>> x[2].#查看后缀名  
'py'  
>>> x[0].#查看文件名  
'xxx'
```

但是，如果我的文件名和后缀名是这个样子的：`xxx.yy.py`，上面的做法就会有问题：

```
>>> new_file_name='xxx.yy.py'  
>>> x=new_file_name.partition('.')
```

看，文件名本来是 `xxx.yy`，而分割而来的文件名是 `xxx`，同样后缀名也出错了。

这是因为，`partition()` 是从左往右扫描的，扫描到第一个 `()` 内指定的分隔符(这里就`'.'`)就开始分割，分割符自己作为一部分，其左右两侧的字符串各作为一部分，总共 `3` 部分。

那也没有解决办法呢？简单，只需要将扫描的顺序改为 `从右向左` 就好了。下面的 `rpartition()` 就做了这件事。

4. `rpartition()`

来直接看栗子：

```
>>> new_file_name='xxx.yy.py'  
>>> x=new_file_name.rpartition('.').#x共3部分  
>>> x[0]  
'xxx.yy'  
>>> x[2]  
'py'  
>>> x[1]  
'.'
```

`x[0]` 取出的便是文件名，`x[2]` 取出的是后缀名。

字符串的格式化操作

1. `ljust()`

让字符串以指定长度显示，如果长度不够，则在 `右边` 填充。默认使用空格填充，可手动修改，看栗子

```
>>> x='Good'  
>>> x.ljust(6).#默认以空格填充  
'Good  '
```

```
>>> x.ljust(6,'*')#以'*'填充  
'Good**'
```

如果指定的长度小于原字符串本身的高度，则返回原字符串

```
>>> x='Good'  
>>> x.ljust(3)#3<4  
'Good'
```

2. rjust

与 `ljust` 功能相同，只不过是在 `左边` 进行填充

```
>>> x='Good'  
>>> x.rjust(6)#在左边默认以空格填充  
' Good'  
>>> x.rjust(6,'*')#在左边以'*'填充  
**Good**  
>>> x.rjust(3) #3<4  
'Good'
```

3. center

功能同上，只是在两侧填充

```
>>> x='Good'  
>>> x.center(6,'*')  
'*Good*'  
>>> x.center(7,'*')  
 '**Good**'  
>>> x.center(8,'*')  
 '**Good**'
```

这里推荐一个记忆的方法：

`ljust` 中的 `l` 代表 `left`，意思是说让原字符串左对齐，所以是在右边填充；

`rjust` 中的 `r` 代表 `right`，意思是说让原字符串右对齐，所以是在左边填充；

`center` 代表 `中间`，意思是说让原字符串居中，所以是在字符串两侧填充；

删除字符串中的特定内容

1. strip()

注意：这个方法只能删除字符串开头或结尾的字符，不能删除中间部分的字符。如果()内什么也不写，默认就是删除字符串中的空格

```
>>> x=' ab c d '
>>> x.strip()#默认删除空格
'ab c d'
>>> s='**abc*d**'#指定删除字符为'*'
>>> s.strip('*')
'abc*d'
```

2. rstrip()

看完了上面的内容，这里已经不必多讲了吧，直接上栗子：

```
>>> x=' ab c d '#删除右边的空格
>>> x.rstrip()
' ab c d'
>>> s='**abc*d**'
>>> s.rstrip('*')#删除右边的指定字符
'**abc*d'
```

3. lstrip()

同样直接上栗子！

```
>>> x=' ab c d '
>>> x.lstrip()
'ab c d '
>>> s='**abc*d**'
>>> s.lstrip('*')
'abc*d**'
```

字符串中的大小写转换方法

1. upper()

将字符串中的所有字符全部变成大写

```
>>> s='abcDEFg'  
>>> s.upper()  
'ABCDEFG'
```

2. lower()

将字符串中的所有字符全部变成小写

```
>>> s='abcDEFg'  
>>> s.lower()  
'abcdefg'
```

3. swapcase()

大写变小写， 小写变大写

```
>>> s='abcDEFg'  
>>> s.swapcase()  
'ABCdefG'
```

4. title

让每个单词的首字母大写

```
>>> seq='hello world I Love Python'  
>>> seq.title()  
'Hello World I Love Python'
```

5. capitalize()

只让字符串的首字母大写， 其余部分均小写

```
>>> seq='hello world I Love Python'  
>>> seq.capitalize()  
'Hello world i love python'
```



今天你将学到以下内容：

1. `input()` 用于接收用户的输入
2. 学会使用 `print()`
3. `print()` 的格式化输出
4. 字符串的补充内容

”

开篇

前面两期详细的介绍了字符串及其相关操作，在今后的编程中，你将经常和字符串打交道。一个比较常见的场景就是 [输入输出](#)，所以本期将结合字符串，讲述最基本同时也是最常用的输入输出语法（`input` 和 `print`），并且会在文章最后补充一些之前没有讲到的字符串的内容。

input() 用于接收用户的输入

`input()` 用于接受用户的输入。

```
>>> input()
```

当你写入以上代码，并按下回车键后，你会发现光标在闪动，这是在提示你要输入一些东西。

当输入完成后，再按一次回车，你所输入的东西便会打印出来显示在屏幕上，就像下面这样：

```
>>> input()  
我将在南极找寻你  
'我将在南极找寻你'  
>>>
```

图中，黑色的字是我输入的，蓝色的字是打印输出在屏幕上的。

那如果不想打印输出在屏幕上，而是想将输入的内容保存起来，又应该怎么做呢？

简单！只要用一个变量接收 `input()` 的输入内容就搞定啦！

```
>>> x=input()  
我将在南极找寻你  
>>>
```

这样，输入的内容就被保存在变量 `x` 中了。

```
>>> x #查看输入的内容  
'我将在南极找寻你'  
>>>
```

细心的你或许已经发现了，蓝色的字被一对 `' '` 包裹着，这不正是之前学习的字符串类型吗？

没错！事实上，你输入 `input()` 中的任何类型的数据，都会变成字符串类型。

你可能不太信，因为上面的栗子中，我输入的内容 `我将在南极找寻你` 本身就是个字符串类型的，所以输出的自然也是字符串咯。为了证明我说的是对的，咱们可以试着输入一个数字，看看出来的是不是还是数字类型。

```
>>> number=input()  
2333  
>>> type(number)  
<class 'str'>  
>>> |
```

看吧，输入的数字 `2333` 也变成了字符串 `str` 类型。

以上就是 `input()` 的基本用法。

那你可能又会想：在提示输入时只有一个光标闪动会不会太不明显了？能不能加一些提示信息呢？

可以！`input()` 中的 `()` 里面可以添加提示信息，看这个栗子：

```
>>> age=input('你今年多大了呀？请输入一个整数：')
你今年多大了呀？请输入一个整数：22
>>> |
```

蓝色的字是打印的提示信息，`22` 是我的输入，它被保存在了变量 `age` 中。

学会使用print()

大家应该对 `print()` 不陌生了，之前的几期内容中也提到过它，今天就对它的常见用法做一个总结。

最基本的，就是将目标内容输出显示到屏幕上：

```
>>> age=22
>>> print(age)
22
```

看起来有些单调，所以可以添加一些描述性的语句，这些语句是字符串类型，所以需要包裹到一对引号内，看下面的栗子：

```
>>> age=22
>>> print('我的年龄是', age)
我的年龄是 22
```

这样看起来就清晰很多，不至于只有一个数字 `22` 而无任何其他信息。

但是对于稍稍有些强迫症的我来说，这还不够完美，因为在输出到屏幕上的内容中，‘我的年龄是’和22之间有一个空格，我想要让他们之间无缝对接，那应该怎么做呢？

其实也不难。`print()`提供了一个`seq`的参数，可以使你自己设置被逗号分隔的两段内容（这里就是‘我的年龄’和`age`）在输出时用什么符号连接，如果不设置，也就是什么都不写，那么默认就是用空格连接起来。

所以，我们只要将`seq`设置为`空`就可以了，就像下面这样：

```
>>> age=22
>>> print('我的年龄是', age, sep=' ')
我的年龄是22
```

当然，你也可以设置为`:`，`*`，`+`以及其他你能想到的字符。

```
>>> age=22
>>> print('我的年龄是', age, sep=':')
我的年龄是:22
>>> print('我的年龄是', age, sep='*')
我的年龄是*22
>>> print('我的年龄是', age, sep='+')
我的年龄是+22
```

现在考虑这么一个问题：

已知两个人的年龄，要求你将两个人的年龄输出到屏幕上(不限格式)。

可以创建一个脚本文件，内容如下：

```
age1=22
age2=23
print('第一个人的年龄是',age1)
print('第二个人的年龄是',age2)
```

你将在屏幕上看到以下输出结果：

第一个人的年龄是 22

第二个人的年龄是 23

>>>

这时候，题目增加了一个要求：两个 `print()` 语句输出的内容必须显示在同一行。

这应该如何实现呢？

其实，`print()` 语句默认在运行结束后打印一个 换行符，我们可以通过 `end` 参数来修改，比如 `end='*'` 表示在运行结束后打印一个 * 而不是 换行符。

所以，问题解决了，在第一个 `print()` 语句中指明 `end` 为空格即可：

```
age1=22
age2=23
print('第一个人的年龄是',age1,end=' ')
print('第二个人的年龄是',age2)
```

输出：

第一个人的年龄是 22 第二个人的年龄是 23

总结一下：

`end` 用于指明在执行完 `print()` **语句之后附带着打印的符号，若不指明，则使用默认的换行符 **

`sep` 用于指名在同一个 `print()` 中，用逗号分隔的每个部分之间连接的符号，若不指明，则使用默认的空格

print()的格式化输出

方法1.类似 C 语言的方法

如果你没有学习过 C 或者早已忘记了 C 也不影响阅读，因为下面要讲的跟 C 本身并无直接关系。

在 Python 的 `print()` 中，可以使用以 % 开头的东西来占位置，所以形象的称之为 占位符。主要有3种，分别是 `%d`，`%f`，`%s`，分别对应了 整数，浮点数（小数）和 字符串。

看下面这个栗子：

```
>>> age1=22  
>>> age2=23  
>>> print('第一个人的年龄是%d,第二个人的年龄是%d'%(age1,age2))
```

第一个人的年龄是22,第二个人的年龄是23

注意，后面的 `% (age1, age2)` 中变量的顺序要与前面的占位符的顺序一一对应

再上个栗子，自己跟着敲一下，基本就掌握了。

```
>>> name='小明'  
>>> age=18  
>>> favorate_number=3.1415926  
>>> print('%s今年%d岁了，他最喜欢的数字是%.2f'%(name,age,favorate_number))#.2f是指保留两位小数
```

小明今年18岁了，他最喜欢的数字是3.14

方法2.使用 `format`，用 `{}` 作为占位符 (ps：我自己经常使用的是这个)

如果占位符 `{}` 内什么也不写，那么每一个 `{}` 就会按出现的顺序与 `format` 中的变量名一一对应，举个栗子就清楚啦：

```
>>> name='小明'  
>>> age=18  
>>> favorate_number=3.1415926  
>>> print('{}今年{}岁了，他最喜欢的数字是{}'.format(name,age,favorate_number))
```

小明今年18岁了，他最喜欢的数字是3.1415926

如果占位符 `{}` 写了下标，那么 `format` 中变量名的位置下标必须与 `format` 中的下标从小到大一一对应，看这个栗子：

```
>>> name='小明'  
>>> age=18  
>>> favorate_number=3.1415926  
>>> print('{1}今年{2}岁了，他最喜欢的数字是{0}'.format(favorate_number,name,age))
```

小明今年18岁了，他最喜欢的数字是3.1415926

结合下图，或许你能更好的理解上面这个栗子

```
print('{1}今年{2}岁了，他最喜欢的数字是{0}'.format(favorite_number, name, age))  
    下标 1     2          0      下标 0    1   2
```

蓝色下标和红色下标必须一一对应

还有一种情况，就是 `{}` 传入的不仅可以是下标，还可以是变量的名字，此时 `format` 中传入的是 `变量名=内容`，看栗子：

```
>>> name='小明'  
>>> age=18  
>>> favorite_number=3.1415926  
>>> print('{name}今年{age}岁了，他最喜欢的数字是{favorite_number}'.format(favorite_number=favorite_number, nam
```

小明今年18岁了，他最喜欢的数字是3.1415926

字符串的补充内容

在上面关于 `print()` 的学习中，我们遇到了 `换行符`，以 `%` 开头的占位符等，这些东西又可以牵扯出字符串的几个知识点，下面来看一下。

1.转义字符

`Python` 中使用 `\` 代表其后面的第一个字符是普通的字符串，而不是 `Python` 中具有某些功能的标识。比如，我想要输出 `I'm ok` 到屏幕上，直接写

```
x='I'm ok'  
print(x)
```

会报错，因为 `Python` 认为 `I` 被一对单引号包裹，单独成一部分，而后面的 `m ok` 是另一部分。这样，由于后面那部分只有右侧的单引号而没有左侧的单引号，所以产生了语法错误。

这时候，转义字符 `\` 就派上用场了！

```
>>> x='I\'m ok'  
>>> print(x)
```

在 \ 后面的 ' 被认为是一个普通的字符，从而可以直接打印输出了。

还有一种特殊情况，就是在上面的格式化输出中，会用到 %s，那万一字符串中正好有一部分叫做 %s 呢？百字不如一栗，上栗子：

在这个栗子中，有个人的名字叫做"王%s"

```
>>> age=18
>>> favorate_color='black'
>>> print('王%s的年龄是%d, 王%s最喜欢的颜色是%s'%(age,favorate_color))
```

王%s的年龄是18, 王%s最喜欢的颜色是black

看，这里的 % 也充当了转义字符的作用，它使得 %s 中的 % 不作为具有某些功能的符号，而只是一个普通字符。

2. 常用转义字符

转义字符	描述
\(在行尾时)	续行符
\\	反斜杠符号
\'	单引号
\"	双引号
\a	响铃
\b	退格(Backspace)
\e	转义
\000	空
\n	换行
\v	纵向制表符
\t	横向制表符
\r	回车
\f	换页

\oyy	八进制数，yy代表的字符，例如：\o12代表换行
\xyy	十六进制数，yy代表的字符，例如：\x0a代表换行
\other	其它的字符以普通格式输出

本图源自网络

** 3.* * 防止转义

看这个栗子：

```
>>> path='C:\Users\username\Desktop\ff.txt'
SyntaxError: (unicode error) 'unicodeescape', codec can't decode bytes in position 2-3: truncated \UXXXXXXXXX escape
```

由于字符串中包含转义字符 \，所以会报错。

所以这里要考虑防止转义。

解决方法也很简单，那就是在转义字符 \ 前面再加一个 \，这样， \\ 代表的意思就是一个普通的反斜杠了：

```
>>> path='C:\\Users\\\\username\\\\Desktop\\\\fj.txt'
```

其实，Python 还提供了另一种更简单的方法，那就是在要防止转义字符生效的字符串前面加一个 r，这个过程正好与上面的转义功能相反

看，在字符串前面加一个 r 就可以防止转义了，这和加两个 \\ 的作用一模一样：

```
>>> path=r'C:\\Users\\\\username\\\\Desktop\\\\fj.txt'
```

Day 6 跳动的数字，熟悉又陌生



今天你将学到以下内容：

1.int & float

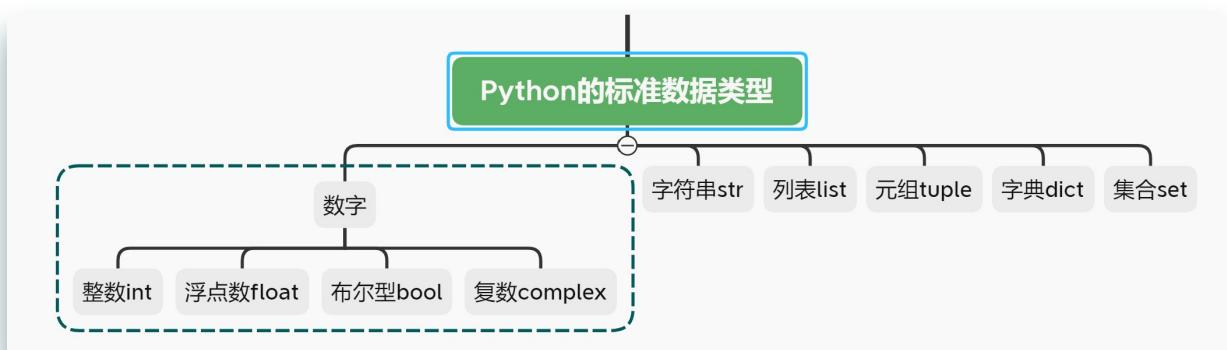
2.bool

3.complex

”

开篇

经过前面与字符串的相爱相杀，相信你已经初步掌握了字符串的基本用法。本期将开始学习一个新的数据类型：**数字**。



在初高中，你已经学习了**整数**，**小数**，**复数**，这些包含在今天要讲的**Python**的数字类型中，正如上图所示（注：在这里你可以先把浮点数理解成小数，这对于之后的编程几乎没有影响，但随着学习的深入，你可能会发现这种理解是片面的）。

下面来逐个击破～

int & float

这两种类型不用多说，你已经与它们打了多年的交道了。

```
>>> a=1
>>> b=2
>>> print(a+b)
3
>>> c=1.5
>>> d=2.5
>>> print(c+d)
```

```
>>> print(a)
4.0
>>> |
```

还记得之前讲过的 `type()` 函数吗？它可以用来看一个变量所属的数据类型。

```
>>> a=1
>>> print(type(a))
<class 'int'>
>>>
>>> b=1.1
>>> type(b)
<class 'float'>
>>>
```

嗯，意料之中。

在一些其他编程语言，如 `C` 语言中，`\` 代表的是整除，比如 `3/2=1` 而不是 `1.5`。

以前的 `Python2` 也这样干过，但是在 `Python3` 中（本系列教程一直在用的 `Python` 版本是 `Python3`，还记得第一期的环境搭建内容吗？），这一规则被更改了，更改后的规则也许更符合大众的逻辑：

`\` 代表我们所认知的普通除法，`//` 代表整除。

上个栗子：

```
>>> x=3
>>> y=2
>>> x/y
1.5
>>> x//y
1
>>>
```

布尔类型(`bool`)只有两种取值，`True`和`False`

你可能产生疑惑，这是哪门子数字类型啊？这不是英文单词吗！

首先用一个栗子反驳你：

```
>>> print(True+False+1)
2
>>> |
```

看，可以做数学运算，说明`True`和`False`是数字。（在`Python`中，字符串是不能用来做加减乘除等运算的）

事实上，`True`和`False`分别代表了数字`1`和数字`0`。

所以，`True+False+1`的结果等价于`1+0+1`的结果。

【扩展】 `bool`继承自`int`类，是`int`的子类。所以`bool`类型唯一的两个取值`True`和`False`自然也可以像`int`那样进行加减乘除等数学运算。

但请注意，`布尔类型`的出现并不是为了做加减乘除运算，若真如此岂不是自找麻烦嘛。

`布尔类型的真正用途在于：判断真假`

判别规则：`空或零为假(False)`，`其余为真(True)`

布尔类型判别真假已经被广泛应用，有许多表达方式可以返回一个‘布尔值’，也就是`True`或`False`

第一种用法是使用`bool()`，看栗子：

```
>>> bool()#括号内什么也不写，那就是空的，所以返回False
False
>>> bool(0)#0和1分别是False和True的值，所以这里返回False
False
>>> bool(1)#
True
>>> bool(2)#非空非零返回True
True
>>> |
```

第二种用法，见下面的栗子。它常用于判断一个 Python 表达式（表达式是 运算符 和 操作数 所构成的序列，比如 `1 + 5` , `3 < 5` ）是否正确，可以类比我们讲的一句话是对的还是错的。

```
>>> a=1
>>> b=1
>>> a==b
True
>>>
>>> c=1
>>> d=2
>>> c==d
False
>>> c!=d
True
>>>
>>> 5>8
False
>>> 0>-1.1
True
```

上面的栗子中只有一个表达式（比如 `1 > -1.1` ），当表达式数目多于 1 时，就牵扯到了 Python 中的 逻辑运算符 。

对于纯小白来说，你可能已经发现，本节内容出现了 运算符 ， 逻辑运算符 ， `==` , `!=` 等等之前没有见过的东西。

所以为了照顾全部的读者，这一块内容等讲到 Python 中的 运算符 时再进行详解。

complex

复数类型主要用在科学计算中，平时编程时很少用，这里就举几个栗子以做简单介绍：

```
>>> x=1+2j#生成复数的方法1
>>> type(x)#复数类型
<class 'complex'>
>>> complex(3, 4)#生成复数的方法2
(3+4j)
>>> type(complex(3, 4))#复数类型
<class 'complex'>
```

```
<class 'complex'>
>>>
>>> x.real#查看实部
1.0
>>> x.imag#查看虚部
2.0
>>> |
```

Day 7 小小运算符，构筑大世界



今天你将学到以下内容：

1. 算数运算符
2. 比较运算符
3. 赋值运算符
4. 逻辑运算符
5. 成员运算符
6. 身份运算符
7. 位运算符

”

开篇

本期将介绍 `Python` 的运算符。



在正式开始之前，应该给 `运算符` 下个教科书式定义：

运算符用于执行程序代码运算，会针对一个以上操作数项目来进行运算(百度百科)

好吧，这定义越看越晕。不管他了，咱们直接开始，在实际操作中理解运算符！

需要注意的是，你如果见到 `x = x + 1` 这种操作，千万不要把这与数学上的操作混为一谈。

前面也提起过，在数学上，`x = x + 1` 中的 `=` 是 等于 的意思，而在编程语言中，`=` 是 赋值 的意思，将 `=` 右边的值赋值给 `=` 左边。

所以，`x = x + 1` 在数学上是不成立的，但是在编程语言中，是将变量 `x` 做了加 `1` 的操作。

算数运算符

1. `+` : 加法运算

2. `-` : 减法运算

3. `*` : 乘法运算

4. `/` : 普通除法运算

5. `//` : 整数除法（整除）运算

6. `**` : 平方运算，`a ** b` 代表求 `a` 的 `b` 次方

7. `%` : 取余运算

上个栗子：

```
>>> 1+12
>>> 1-10
>>> 1*22
>>> 3/21.5
>>> 3//21
>>> 2**3#求2的3次方8
>>> 9%2#取余数, 9//2=4, 余11
>>> 13%5# 13//5=2, 余33
```

比较运算符

1. `==` : 比较两个数是否相等

2. `!=` : 比较两个数是否不等

3. `>` : 比较两个数是否前者大于后者

4. `<` : 比较两个数是否前者小于后者

5. `>=` : 比较两个数是否前者大于等于后者

6. `<=` : 比较两个数是否前者小于等于后者

下面的栗子中的返回值都是上期所讲的布尔类型，代表真或假。

```
>>> a=1
```

```
>>> b=2
```

```
>>> a==b#查看a是否等于b
```

```
False
```

```
>>> a!=b#查看a是否不等于b
```

```
True
```

```
>>> a>b#查看a是否大于b
```

```
False
```

```
>>> a<b#查看a是否小于b
```

```
True
```

赋值运算符

1. `=` : 简单赋值运算符，将赋值运算符右侧的值赋值给左侧

2. `+=` : 加法赋值运算符，`a += b` 等价于 `a = a + b`

3. `-=` : 减法赋值运算符，`a -= b` 等价于 `a = a - b`

4. `*=` : 乘法赋值运算符，`a *= b` 等价于 `a = a * b`

5. `/=` : 除法赋值运算符，`a /= b` 等价于 `a = a / b`

6. `//=` : 整除赋值运算符，`a // b` 等价于 `a = a // b`

7. `**=` : 幂赋值运算符，`a **= b` 等价于 `a = a ** b`

8. `%=` : 取模赋值运算符，`a %= b` 等价于 `a = a % b`

```
>>> x=3#将3赋值给x
```

```
>>> x+=1#等价于x=x+1, 也就是将x+1的值赋值给x, 这样x就被更新了
```

```
>>> print(x)
```

```
4
```

```
>>> x/=2#等价于x=x/2
```

```
>>> print(x)
```

```
2.0
```

```
>> x**=3#等价于x=x**3
```

```
>>> print(x)
```

```
8.0
```

逻辑运算符

1. and : 布尔‘与’

在 `a and b` 中，如果 `a` 与 `b` 的值都是 `True`，那么就返回 `True`，否则，只要有一个为 `False`，那么就返回 `False`。

```
>>> a=1
```

```
>>> b=1
```

```
>>> a and b #a与b都为True, 返回True (1)
```

```
1
```

```
>>> c=0
```

```
>>> d=1
```

```
>>> c and d #有一个为False就返回False (0)
```

```
0
```

```
>>> e=0
```

```
>>> f=0
```

```
>> e and f #都为False, 毫无疑问返回False
```

```
0
```

2. or : 布尔‘非’

在 `a or b` 中，只要有一个为 `True`，就返回 `True`，只有当两者都为 `False` 是，才返回 `False`。

```
>>> a=1
```

```
>>> b=1
```

```
>>> a or b #都为True, 毫无疑问返回True
```

```
1
```

```
>>> c=1
>>> d=0
>>> c or d #有一个为True, 那就返回True
1

>>> e=0
>>> f=0
>>> e or f #全都为False, 那就只能返回False了
0
```

3. not

相当于一个取反的操作

```
>>> a=1
>>> b=2
>>> not (a and b) # a and b 为真, 取反为假
False

>>> c=0
>>> d=3
>> not (c and d) # c and d 为假, 取反为真
True

>>> e=0
>>> f=0
>>> not (e and f) # e and f 为假, 取反为真
True
```

成员运算符

1. in

在指定的序列中查找指定的值，如果能找到则返回 `True`，否则返回 `False`。

这里的序列可以是列表，可以是字符串，也可以是其他数据类型。鉴于目前本系列只讲了字符串，所以这里用字符串举例，其它数据类型其实也一样，以后遇到会说明的。

```
>>> s='hello world'
>>> 'he' in s
True
```

```
>>> 'nanji' in s  
False
```

2. not in

与 `in` 的功能正好相反，直接看栗子：

```
>>> ss='I just love Python'  
>>> 'love' not in ss  
False  
  
>>> 'hello' in ss  
False  
  
>>> 'hello ' not in ss  
True
```

身份运算符

1. is

相信你还记得在[字符串的增删改查](#)那一期中，我们曾介绍过 `id()`，当时是用它来查看两个变量中所存内容是不是来自同一个小房间（内存地址单元）。

这里的 `is` 所做的事情和上面所做事情基本是一样的

`a is b` 就等价于 `id(a)==id(b)`，这两个表达式都会返回一个布尔值，用来表明该表达式的对与错。

如果 `a` 与 `b` 所在的内存地址相同，则返回 `True`，否则返回 `False`。

对于入门来说，了解关于 `is` 的以上使用说明就足够了，本部分接下来的内容对于小白来说可能会感到难以理解，没关系，你大可跳过，以后用的多了自然就理解了。

举个栗子：

【注】以下实例均在 `Python` 自带的 `IDLE(3.7)` 中运行

【栗子1】

```
>>> a='abc'  
>>> b='abc'
```

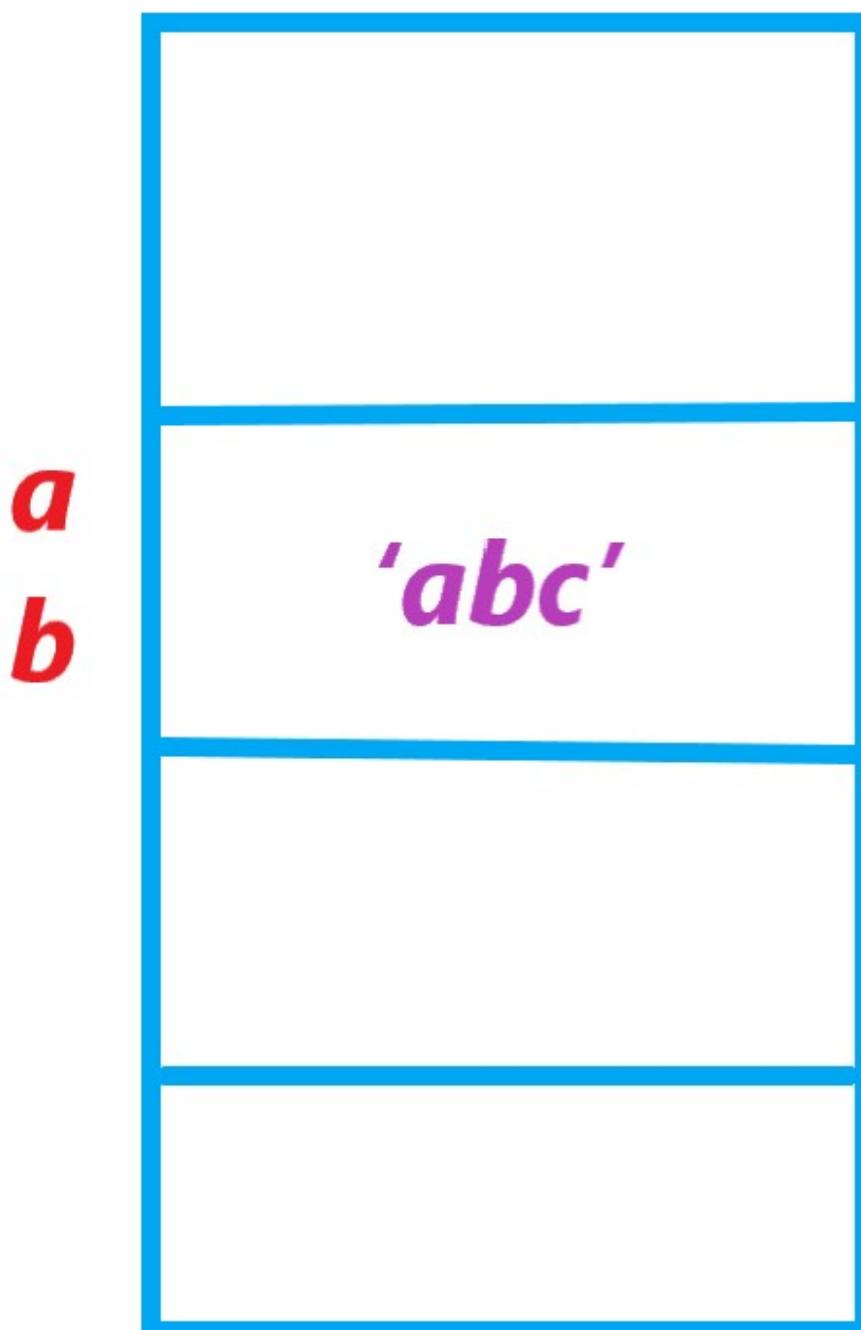
```
>>> a is b
```

```
True
```

```
>>> id(a)==id(b)
```

```
True
```

这个栗子中，在内存（房子）中开辟了一块内存空间（小房间），用来存储字符串 '`a b c`'，然后让变量 `a` 与 `b` 都指向这一块内存空间，就像这样：



内存

再举个数字的栗子：

【栗子 2】

```
>>> a=1  
>>> b=1  
>>> a is b  
True  
  
>>> id(a)==id(b)  
True
```

这和上面的字符串栗子的解释基本一样，只需将上图中的 `a b c` 换成数字 `1` 即可。

但是，有一点要提出来的是，并不是所有的字符串和数字都是这样的，看下面的栗子：

【栗子 3】

```
>>> a="hello world"  
>>> b="hello world"  
>>> a is b  
False  
  
>>> id(a)==id(b)  
False
```

【栗子 4】

```
>>> c=257  
>>> d=257  
>>> c is d  
False  
  
>>> id(c)==id(d)  
False
```

惊不惊喜，意不意外？

究其原因，是因为 `Python` 中设置了一个**“小整数对象池”**，里面存积着许许多多的对象。如果你所创建的对象（这里先按照变量来理解就行）中的内容（假定内容为

`xxx`) 包含在在这个“小整数对象池”里面，那么当有两个不同的变量的内容都是`xxx`时，这两个变量就会同时指向这一块存储了`xxx`的内存空间。

如果`xxx`不包含在“小整数对象池”中，那么就会像【栗子3】和【栗子4】中一样，不同的变量，即使内容相同，也会指向不同的内存空间。

那你肯定想要知道，这个“小整数对象池”中的内容都有哪些？

我也不知道~(\手动狗头)

不过经过一番狂搜，我找到了如下答案：

在python中，为避免小数值整数和短长度字符串频繁申请和销毁内存空间，使用了小整数对象池。Python shell中，对小数值整数的定义是[-5, 256]，小整数字符串是只包含字母、数字、下划线的字符串，这些整数对象是提前建立好的，不会被垃圾回收，新建一个相同的值，永远对应同一个内存地址。

所以，在python shell里面，如果`a, b`数值相同，而且又都在小整数对象池内，那么`a == b, a is b`的答案都是True；

如果`a, b`数值相同，但是不在小整数对象池内，那么`a == b`的答案是True，`a is b`是False：

```
1 >>> a= "111111111111111111111111111111111111111111111" #只包含整数的字符串
2 >>> b= "1111111111111111111111111111111111111111111"
3 >>> a == b
4 True
5 >>> a is b
6 True
7
8
9 >>> a = -5 # [-5,256]的整数
10 >>> b = -5
11 >>> a is b
12 True
13
14
15 >>> a=-88 #[-5,256]以外的整数
16 >>> b = -88
17 >>> a is b
18 False
19
20 >>> a = "aaaaaaaaaaa aaa" # 不是只包含字母、数字、下划线的字符串
21 >>> b = "aaaaaaaaaaa aaa"
22 >>> a is b
23 False
```

(源：https://blog.csdn.net/BGONE/article/details/93311990?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommentFromMachineLearnPai2-1.nonecase&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommentFromMachineLearnPai2-1.nonecase)

(注：图中所讲的`python shell`就是我们一直说的`IDLE`。)

到这里就可以解释本部分的4个栗子了：

【栗子1】中的`abc`只由字母组成，【栗子2】中的`1`属于`[-5, 256]`，所以两个变量可以指向同一块内存空间。而【栗子3】中的`hello world`中包含了`空格`，【栗子

4】中的 257 不属于 [-5, 256]，因此不同的变量即使值相同，也会被存储在两块不同的内存空间中。

还需要说明的是，以上解释仅仅是针对 IDLE，也就是图中所写 python shell，如果你换了另一款 PyCharm，那情况又会不一样。（这算是一种优化吧）。

这是我在 PyCharm 上的运行结果：

```
a = "hello world"
b = "hello world"
print(a is b)
```

Run: test

D:\Anaconda\python.exe C:/Users/fanxi/Desktop/untitled/test.py

True

Process finished with exit code 0

2. is not

相当于是 is 的取反操作， a is not b 等价于 id(a) != id(b)。

位运算符

建议小白同学跳过这一部分，因为涉及到了计算机组成原理中的一些内容。

1. & : 按位与运算符

这里的 与 和前面的逻辑运算符中的'布尔与'中的 与 是一个意思，所以，按位与运算，就是把参与运算的两个值先转换成二进制形式，然后逐位对应做 and 运算，对应位全为 1 则结果为 1，否则只要有一个是位是 0 结果便是 0。

举个栗子就清楚啦：

```
>>> x=12
>>> y=13
>>> bx=bin(x)#bin()用于将10进制转为2进制
>>> by=bin(y)#bin()用于将10进制转为2进制
>>> print(bx,by)
```

0b1100

0b1101

x 和 y 对应的二进制表示分别为 0b1100 、 0b1101 ，注意 0b 是二进制的前缀，用来表明这是一个二进制数，并无数学意义。

```
>>> print(bin(x&y))  
'0b1100'
```

所作运算得具体过程如下图所示：



2. | : 按位或运算符

和 & 一样，只是对应位做的不是 and 运算，而是做 or 运算：

```
>>> print(bin(x|y))  
'0b1101'
```



1 1 0 0

按位或

1 1 0 1

有1则1

有1 则1

全0则0

有1则1

1 1 0 1

3. `^`：按位异或运算符

对应位做异或运算，其余同上。

异或规则：对应位相同得 `0`，不同得 `1`

```
>>> print(bin(x^y))  
'0b1'
```

1 1 0 0

按位异或运算

1 1 0 1

相同得0

相同得0

相同得0

不同得1

0 0 0 1

4. `~`：按位取反运算符

这里涉及到了计算机组成原理中的知识，不必深究，只需了解公式 $\sim x = - (x + 1)$ 即可。

```
>>> x12  
>>> print(~x)  
-13
```

按照公式 $\sim x = - (x + 1)$ 来走一遍：

```
>>> z=x+1  
>>> print(-z)  
-13
```

验证完毕。

5. `<<` 与 `>>`：左移运算符和右移运算符

也建议直接了解下以下公式：

左移 m 位，相当于原来的 10 进制数字乘以 2 的 m 次方，即 $\text{new_x} = x * (2^{**m})$ 。

右移 m 位，相当于原来的 10 进制数字除以 2 的 m 次方并取整，即 $\text{new_x} = x // (2^{**m})$ 。

```
>>> x  
12  
>>> print(x<<2) #x左移2位  
48  
>>> x*(2**2) #按公式计算得到相同结果  
48  
  
>>> x  
12
```

```
>>> print(x>>2) #x右移2位  
3  
>>> x//(2**2) #按公式计算得到相同结果  
3
```

Day 8 分支循环，效率加速器



今天你将学到以下内容：

- 1. 分支
- 2. 循环
- 3. break 和 continue
- 4. while ... else ... 与 for ... else ...

”

开篇

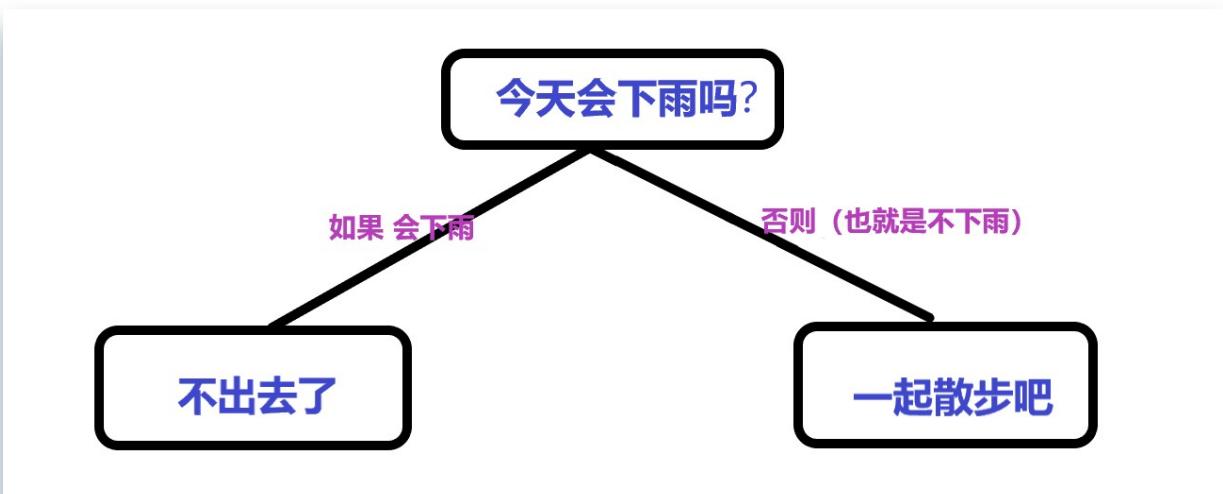
你可能已经发现，前面的七期内容有一个共同的特点，那就是偏向理论基础，似乎学完之后只是了解了一些实用的方法，比如字符串如何进行增删改查，如何交互实现输入输出，Python 运算符的种类等等。

此刻的你，是不是一点也没有感受到程序带来的效率翻倍？

今天就来填补这个大坑，一起来学习 分支 和 循环 吧！

分支

所谓分支，可以形象的理解为一棵树的分叉：



满足哪个条件，就往哪个分叉方向前进。

写成 `Python` 伪代码的格式如下：

```
if 今天会下雨:  
    print('不出去了')  
else:  
    print('一起散步吧')
```

这便是最简单的分支结构：`if...else...`

还记得之前讲的缩进吗？在这里，`:`之后按回车键就会自动缩进了，一个缩进相当于按一次 `Tab` 键

缩进的存在，使得 `Python` 代码的可读性更强。

再举两个栗子来练习一下，建议小白同学手动敲一遍：

【栗子1】数字比较

```
x=1  
y=2  
if x>y:  
    print('x比y大！')  
else:  
    print('x比y小！')
```

输出：

`x比y小！`

【栗子2】年龄验证

```
age=input('请输入你的年龄：')  
age=int(age)#input输入的是字符串，所以使用int()将其转为数字类型，你还记的吗  
if age>=18:  
    print('你好！')
```

```
print('欢迎光临！')
else:
    print('未成年人禁止入内!')
```

输入 18：

请输入你的年龄：18你好！欢迎光临！

输入 12：

请输入你的年龄：12未成年人禁止入内！

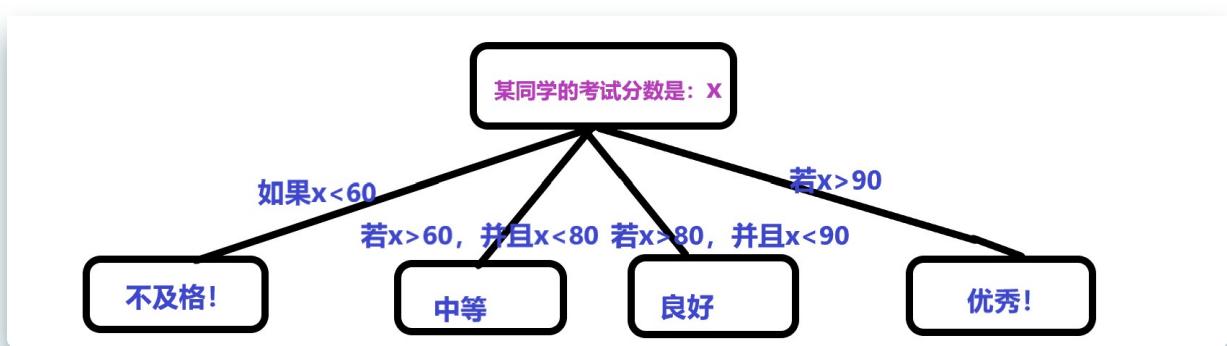
需要注意：

：后面缩进的内容是一个整体，比如【栗子 2】中的 `print('你好！')` 和 `print('欢迎光临！')`

但是，如果某一个问题的答案并不是单纯的为** 是 或者 否 **，而是有多于两种的答案，那么上面所讲的 `if ... else...` 分支语句就无能为力了。

于是 **多分支结构** 出现了！

同样是小树分叉的形式：



图中的栗子写成可以运行的 `Python` 代码如下：

```
x=input('请输入某同学的考试分数: ')
x=int(x)
if x<60:
    print('不及格!')
elif x<80:
    print('中等')
elif x<90:
    print('良好')
else:
    print('优秀!')
```

```
print('良好')
else:
    print('优秀!')
```

运行及结果：

```
请输入某同学的考试分数: 99优秀!
```

这种多分支结构采用了 `elif` 关键字，它的全称是 `else if`，意思是在否定它上面的条件之后，同时又满足它自身的条件时，就执行它所对应的缩进块内的代码。

在上面的栗子中，假设 `x = 99`，我们一起走一遍：

检查`if`语句，发现`if`后面的`x < 60`不满足，因此跳转到`if`下面紧邻的第一条`elif`语句

检查第1条`elif`语句（此时由于已经检查了`if`语句，所以`x >= 60`是默认成立的），发现后面的`x < 80`不满足，因此跳转到第二条`elif`语句

检查第2条`elif`语句（此时由于已经检查了第一条`elif`语句，所以`x >= 80`是默认成立的），发现后面的`x < 90`不满足，因此跳转到最后的`else`语句

检查最后的`else`语句（此时由于已经检查了第2条`elif`语句，所以`x >= 90`是默认成立的），发现`x = 99`满足`x >= 90`这个条件，于是打印'优秀！'

以上便是基本的分支结构包含的知识点。

这里再说一下嵌套的分支结构，也就是 `if` 里面包裹着 `if`，直接上栗子：

```
x=1
y=2
z=3
if x < y:
    if y < z:
        print('x < y < z')
```

输出结果：

```
x < y < z
```

解释：在第一个 `if` 中，满足 `x`，于是继续向下执行第二个 `if` 语句（此时已经在满足 `x` 成立的前提下进行的），条件 `y` 也满足，于是打印输出 `x`。

你可能也发现了，最后一个条件不必须是 `else`，可以没有，当然也可以是 `elif`。这一点在今后的编程中你将逐渐体会。

循环

在 `Python` 中，循环有两种，一种是 `while` 循环，另一种是 `for` 循环。

while循环

`while` 后面跟一个条件表达式，只要条件满足，就一直执行 `while` 里面缩进的代码块。举个例栗子：

```
x=1
sum=0
while x<=10:
    sum+=x
    x=x+1
print('1+2+...+10=',sum)
```

这个栗子用来求解 `1` 到 `10` 之和。

首先，初始化 `x` 为 `1`，用于存放求和结果的变量 `sum` 被初始化为 `0`。

只要条件表达式 `x<=10` 满足，那么就一直执行 `while` 所包裹的缩进的内容，也就是

```
sum+=x
x=x+1
```

`x` 从 `1` 开始，每运行一次，`x` 就加 `1`，最终 `x` 变为 `11`，此时不再满足条件 `x<=10`，于是退出 `while` 循环，打印求和结果，程序运行结束。

for循环

`for` 循环的结构为

`for` 迭代变量 `in` 可迭代对象：

代码块

我们目前所学习到的可迭代对象只有字符串，因此这里便先用字符串举例子了。

```
word='Python'  
for letter in word:  
    print(letter)
```

输出

```
P  
y  
t  
h  
o  
n
```

现在来使用 `for` 循环求 `1` 到 `10` 之和：

```
sum=0  
for number in range(1,11):  
    sum+=number  
print('1+2+3+...+10=',sum)
```

输出

```
1+2+3+...+10= 55
```

这里使用到了 `range()` 函数，它用于生成左闭右开的一个数字序列，具体用法如下：

```
range(start,end,step)
```

其中，

`start`：开始的数字

`end`：结束的数字 +1

`step`：步长

`start` 和 `end` 必须是整数(可正可负)，`step` 无限制

如果不写步长值 `step`，则默认步长为 `1`

如果不写开始值 `start`，默认从 `0` 开始，此时不能指定 `step`，否则逻辑会出错

看几个栗子：

【栗子1】不写 `step`，则默认为 `1`

```
x=range(1,4)
for i in x:
    print(i)
```

输出

```
1
2
3
```

【栗子2】不写 `start`，默认从 `0` 开始

```
x=range(4)
for i in x:
    print(i,end=' ')
```

输出

```
0 1 2 3
```

【栗子3】写 `step`

```
x=range(4,2,-1)
for i in x:
    print(i)
```

输出

```
4
3
```

你甚至可以像操作字符串那样对 `range()` 生成的对象按照下标访问，也可以做切片（只是返回的仍是可迭代对象）：

```
>>> x=range(1,10) #[1,9]
```

```
>>> x[0]
```

```
1
```

```
>>> x[1]
2

>>> x[8]
9

>>> x[9] #下标越界，所以报错
Traceback (most recent call last): File "<pyshell#22>", line 1, in <module>     x[9]IndexError: range object
```

#切片

```
>>> x[1:3]#取下标1到2，注意左闭右开，所以3取不到
range(2, 4)#下标1对应数字2，下标2对应数字3，注意左闭右开，所以数字4取不到
```

说了这么多，你应该已经很清楚，在求 1 到 10 之和的 for 循环程序中，range(1, 10) 得到的是一个从 1 到 10 的可迭代对象。通过 for 循环，依次取出可迭代对象中的每一个值，加到 sum 中，最后打印 sum 即为所求。

break和continue

这两个用于循环中，用来控制循环的走向。

`break`：直接跳出循环，不再运行循环体内的代码块

`continue`：只结束本次循环，不会再执行 `continue` 语句之后的代码，而是转而从头开始运行循环体内的代码块

举两个栗子来对比这两者：

【栗子1】 `break` 的用法

```
sum=0
for i in range(1,7):#i从1到6
    if i>5:
        break      #只有在上面的i>5不成立时才执行下面的代码
        sum+=i
print(sum)
```

输出

解释：当 `i` 变为 `6` 时，满足 `if` 条件，于是执行 `break` 语句，跳出整个循环。

所以，上面的代码的作用是求解 `1+2+3+4+5` 的和，也就是 `15`。

【栗子2】`continue` 的用法

```
sum=0
for i in range(1,7):#i从1到6
    if i==5:
        continue#当i=5时，不再执行下面的代码，而是重新开始一次新的循环
        sum+=i
print(sum)
```

输出

`16`

解释：当 `i=5` 时，满足 `if` 条件，所以会直接进入下一次循环。所以，上面代码的作用就是求解 `1+2+3+4+6` 之和，也就是 `16`。

补充while循环和for循环的另一种用法：`while ...else...`与`for ...else...`

不管是在 `while ...else...` 中，还是在 `for ...else ...` 中，只要循环体内的代码是正常执行完之后跳出的循环，那么都会执行 `else` 中的代码。但是，如果循环是被 `break` 强行终止的，那么抱歉，此时 `else` 中的语句会被忽略掉。

这里就以 `while ...else...` 为例进行演示：

【栗子1】循环体内没有写 `break`，正常结束循环，于是会执行 `else` 中的代码

```
x=1
sum=0
while x<=10:
    print('x=',x)
    sum+=x
    x=x+1
else:
    print('x={}, x<=10不再满足，即将退出while循环'.format(x))
    print('1+2+...+10=',sum)
```

输出

```
x= 1
x= 2
x= 3
x= 4
x= 5
x= 6
x= 7
x= 8
x= 9
x= 10
x=11, x<=10不再满足, 即将退出while循环
1+2+...+10= 55
```

【栗子2】循环体内写了 `break` 语句，但是 `break` 语句并没有被执行，因此也会执行 `else` 中的代码

```
sum=0
for i in range(1,7):
    if sum>100:
        break
    sum+=i
else:
    print('程序正常结束')
print(sum)
```

输出

程序正常结束21

【栗子3】循环体内写了 `break` 语句，并且触发了 `break` 语句，循环被迫终止，所以此时不会执行 `else` 中的代码

```
sum=0
for i in range(1,7):
    if i>5:
        break
    sum+=i
else:
    print('程序正常结束')
```

```
print(sum)
```

输出

15

Day 9 简单的列表，却囊括了世间万物



今天你将学到以下内容：

1. 初识列表
2. 创建列表
3. 操作列表
4. 求列表长度
5. 按照下标取对应位置元素
6. 列表做切片
7. 遍历列表元素
8. 求列表中数字元素的最值
9. 求列表全部元素之和
10. 列表拼接
11. 列表的增删改查
12. 列表推导式

”

开篇

Python的标准数据类型

数字 字符串str **列表list** 元组tuple 字典dict 集合set

我们已经学习了**数字**和**字符串**这两种数据类型，它们都有一个共同的特点，那就是一个变量只能存储一个数字或者字符串，如果有多个数字或者多个字符串，那就必须得用多个变量来存储。

比如，现在有 `'abc'`，`'d'`，`1`，`123`，那么想要存储这些，必须这样做：

```
v1='abc'  
v2='d'  
v3=1  
v4=123
```

当有成百上千甚至更多个待存储的元素时，显然需要的变量个数会非常多，过程会无比繁琐。

别担心！本期要学习的 [列表](#) 就是填补这一缺憾的，学完之后，你会发现，上面的过程只需写成如下一行代码即可：

```
v=['abc','d',1,123]
```

是不是迫不及待了，那就开始吧！

初识列表

列表用 `[]` 来包裹，`[]` 内部可存放元素，这些元素之间用逗号 `,` 隔开。

举个栗子，现在需要统计一下“南极小学”六年级的 `5` 个班中，每个班打算今年暑假去南极旅行的人数，假设各班统计上来的人数分别是 `11`，`111`，`1234`，`2233`，`666`，使用列表可以这样存储：

```
num_go=[11,111,1234,2233,666]
```

通过使用列表，事情一下子变得简洁了！

你已经学习过字符串，并且知道了字符串可以[按照下标取某个字符，做切片](#)等等。

列表也可以这样做！如果字符串的内容学明白了，那么本期内容将不成问题，这两者的某些方法几乎是一模一样的！比如字符串中的 `len()` 用于求字符串中字符的个数，在这里 `len()` 也可以求列表中所含元素的个数。

需要提出的是，列表中的元素可以是不同类型的，就像开篇中 `的` ``['abc','d',1,123]` 一样，这也使得简单的列表变得更加灵活，功能更加强大。

创建列表

方法1：使用 `x=[]` 创建一个空列表：

```
>>> x=[ ]  
>>> print(x)  
[]
```

方法2：使用 `list()` 创建一个空列表

```
>>> x=list()  
>>> print(x)  
[]
```

对比来说，方法1只能创建一个空列表，而方法2不仅可以创建一个空列表，还能将一个字符串，一个元组（后面会讲到）等转为列表数据类型。

举个字符串转列表(`str->list`)的栗子：

```
>>> s='Python!'  
>>> x=list(s)  
>>> print(x)  
['P', 'y', 't', 'h', 'o', 'n', '!']
```

操作列表

1.求列表长度

```
>>> num_go=[11, 111, 1234, 2333, 666]  
>>> len(num_go)#长度为5  
5
```

2.按照下标取对应位置元素

```
>>> num_go=[11,111,1234,2333,666]  
>>> num_go[0]#取下标为0处的元素  
11  
  
>>> num_go[1]#取下标为1处的元素  
111  
  
>>> num_go[5] #最大下标为4，所下标为5会越界报错
```

```
Traceback (most recent call last): File "<pyshell#42>", line 1, in <module> num_go[5]IndexError: list index out of range
>>> num_go[-2]#取倒数第二个元素
2333
```

3. 做切片

```
>>> num_go=[11,111,1234,2333,666]
>>> num_go
[11, 111, 1234, 2333, 666]
```

【栗子1】直接切片

```
>>> num_go[1:3] #取下标1到2的元素（和字符串一样，左闭右开）
[111, 1234]
```

【栗子2】设定步长 step 为 2（若不设定，则使用默认值 1）

```
>>> num_go[0:5:2]#设定步长为2
[11, 1234, 666]
```

```
>>> num_go[-1:-2]#步长为默认值1，也就是每次执行完一次下标就加1，但是start到end是向左走的，加1是向右走的，因此取不到任何元素
[]
```

```
>>> num_go[-1:-2:-1] #设定步长为-1，于是从下标-1开始向左走，所以能取到下标为-1，也就是倒数第一个位置的元素
[666]
```

【栗子3】取全部元素

```
>>> num_go[::]
[11, 111, 1234, 2333, 666]
```

【栗子4】取全部元素的倒序排列

```
>>> num_go[::-1]
[666, 2333, 1234, 111, 11]
```

【栗子5】只指明开始下标而不指明结束下标，会从开始下标一直取到最后

```
>>> num_go[2:]  
[1234, 2333, 666]
```

4.遍历列表元素

可以使用循环操作来遍历列表中的每一个元素，如下：

```
>>> for item in num_go:  
    print(item)  
  
11  
111  
1234  
2333  
666
```

5.求列表中数字元素的最值

```
#最大值  
>>> max(num_go)  
2333  
  
#最小值  
>>> min(num_go)  
11
```

6.求全部元素之和

```
>>> sum(num_go)  
4355
```

7.列表拼接

```
>>> a=[1,2,3]  
>>> b=[4,5,6]  
>>> print(a+b)  
[1, 2, 3, 4, 5, 6]
```

说到增删改查，你是否又想起了在学习字符串的时候，我们也详细的介绍过关于字符串的增删改查。并且经过学习，你已经知道了字符串是不可变数据类型，凡是涉及到字符串的修改的操作，其实都是创建了一个新的字符串，这个新的字符串存储了修改后的字符串，而原字符串依旧丝毫不变。

这里的列表却有所不同，因为 `列表(list)` 是可变数据类型。

在列表中增加，删除，改动等操作都会直接在原列表中修改。

1. 增

在列表中增加元素，常用的有 `append()` 和 `extend()` 这两种方法。

先来个栗子直观感受下，再来讲解两种方法的区别

```
>>> z=['a','b','c']
>>> z.append('d') #添加的元素是一个字符
>>> print(z)
['a', 'b', 'c', 'd']

>>> p=[1,2,3] #添加的元素是一个列表
>>> z.append(p)
>>> print(z)
['a', 'b', 'c', 'd', [1, 2, 3]]

>>> h=list(range(1,6))
>>> print(h)
[1, 2, 3, 4, 5]

>>> u=[6,7,8]
>>> h.extend(u)
>>> print(h)
[1, 2, 3, 4, 5, 6, 7, 8]
```

通过阅读上面两个栗子，可以总结出 `append()` 和 `extend()` 的异同：

两者都是在列表末尾添加元素，不同的是，`append()` 一次只能添加一个元素，而 `extend()` 的 `()` 中必须是一个可迭代对象，因此 `extend()` 可以一次添加多个元素。

- 首先来解释 `append()` 一次只能添加一个元素：

你可能会说，第一个栗子中向列表 `z` 中增加了一个列表，`z` 的内容变为 `['a', 'b', 'c', 'd', [1, 2, 3]]`，这不是增加了 `1`，`2`，`3` 总共3个元素吗？

不是的，这是一个嵌套的列表，也就是列表中的元素的类型也是列表类型，所以里面的 `[1, 2, 3]` 要看成是一个元素。

我们可以通过 [求列表长度](#) 或者 [用下标取元素](#) 的方法来验证以上所说：

```
>>> z['a', 'b', 'c', 'd', [1, 2, 3]]  
>>> len(z)  
5  
  
>>> z[4]  
[1, 2, 3]  
  
>>> z[5]#越界报错  
Traceback (most recent call last): File "<pyshell#77>", line 1, in <module> z[5]IndexError: list index
```

- 再来解释 `extend()` 的 `()` 中必须是一个可迭代对象：

你可以尝试使用 `extend()` 向列表中一个数字：

```
>>> h[1, 2, 3, 4, 5, 6, 7, 8]  
>>> h.extend(2333)  
Traceback (most recent call last): File "<pyshell#88>", line 1, in <module> h.extend(2333)TypeError: 'j
```

看，报错了，报错内容的意思是：整数 (`int`) 不是一个可迭代对象，因此报错。

所以，`extend()` 里面必须是一个可迭代对象。像列表，字符串以及将要学习的元组等都是可迭代对象，因此可以使用 `extend()` 方法添加到列表中。

【添加字符串时，会遍历每一个下标对应的字符，作为列表中的一个元素】

```
>>> lis=[]  
>>> s='I love Python'  
>>> lis.extend(s)  
>>> print(lis)  
['I', ' ', 'l', 'o', 'v', 'e', ' ', 'P', 'y', 't', 'h', 'o', 'n']
```

【添加元组】

#栗子1

```
>>> lis=[]
>>> yz=(1,2,3)#yz是一个元组
>>> lis.extend(yz)
>>> print(lis)
[1, 2, 3]
```

#栗子2

```
>>> lis2=[1,2,3]
>>> a=(2333,) #a是一个元组，并且只包含一个元素，这里看不懂先不要纠结，就快要讲到了
>>> lis2.extend(a)
>>> print(lis2)
[1, 2, 3, 2333]
```

由【栗子2】可以看出，`extend()`方法也可以一次只添加一个元素，只要该可迭代对象中只包含一个元素。

扩展：如何查看是否是可迭代对象？

Python 提供了一种方法：

首先要知道，`isinstance()`用于查看某个对象是否属于某一类型，比如`isinstance(1,int)`就是用于查看数字`1`是否为`int`类型，返回布尔值，这里会返回`True`，因为`1`是`int`类型。

于是推而广之，我们也可以使用`isinstance()`查看某个对象是否是可迭代类型，只需要将该对象与可迭代类型放入`isinstance`中，查看其返回值即可。

可迭代类型`Iterable`需要手动导入：

```
# 这句代码先记住就行，无需深究
from collections.abc import Iterable
```

然后就可以使用了：

```
>>> x=[1,2]
>>> isinstance(x,Iterable)
True#列表是可迭代对象

>>> y='123'
>>> isinstance(y,Iterable)
True#字符串是可迭代对象
```

```
>>> z=2333  
>>> isinstance(z,Iterable)  
False#数字不是可迭代对象
```

以上就是关于 `append()` 和 `extend()` 方法的介绍。那你可能会说，这两种方法都是在列表末尾增加元素，那能不能在列表的最前面或者中间增加元素呢？

`Python` 无所不能！

`insert()` 方法可以实现将某一元素插入到指定下标位置，并将其后元素后移一个位置。

```
>>> x=[1,2,3,'a','b']  
>>> x.insert(0,'2333')#在下标0处添加元素'2333'  
>>> x  
['2333', 1, 2, 3, 'a', 'b']
```

2. 删

- `del`

最粗暴的方法是使用 `del`，这种方法比较强大，可以删除列表中的某一个元素，也可以使用切片方式一次性删除多个元素，甚至可以直接删除整个列表，这样整个列表就不存在了（并不是变为空列表，而是直接抹去）。

看个栗子：

```
>>> x=['2333', 1, 2, 3, 'a', 'b']  
>>> del x[0] #删除下标为0处的元素  
>>> x  
[1, 2, 3, 'a', 'b']  
  
>>> del x[0:3]#删除下标0到2位置的元素  
>>> x  
['a', 'b']  
  
>>> del x #直接删除整个列表，列表就不存在了  
>>> x#所以再次访问列表x的时候会报错  
Traceback (most recent call last): File "<pyshell#51>", line 1, in <module> xNameError: name 'x' is not
```

- `pop()`

`pop()` 会将被删除的元素当做返回值返回。

默认删除列表末尾的元素：

```
>>> x=[1,2,3,4,5]
>>> x.pop()#默认删除末尾的元素5
>>> print(x)
[1, 2, 3, 4]
```

也可以手动指定要删除元素的下标：

```
>>> x
[1, 2, 3, 4]
>>> x.pop(1)#删除下标为1处的元素
2
>>> x
[1, 3, 4]
```

- `remove()`

删除特定的元素，传入的是元素值。看栗子：

```
>>> x=[1,2,3,4,5]
>>> x.remove(3)#删除数字3
>>> print(x)
[1, 2, 4, 5]
```

需要注意的是，若列表中有重复的元素，比如 `lis=[1,1,2,2,3,3,4]`，在使用 `remove()` 方法时每次只能删除下标最小位置处的元素。看栗子：

```
>>> z=[1,2,3,2,1,5]
>>> z.remove(2)
>>> print(z)
[1, 3, 2, 1, 5]
```

- `clear()`

用于清空一个列表，也就是将原列表变为一个空列表：

```
>>> z[1, 3, 2, 1, 5]
>>> z.clear()
```

```
>>> print(z)
[]
```

3. 改

可以一次修改一个元素（通过单个下标实现）：

```
>>> x=['P','y','t','aaa','o','n']
>>> x[3]='h'
>>> print(x)
['P', 'y', 't', 'h', 'o', 'n']
```

也可以一次修改多个元素（通过做切片实现）：

```
>>> z=[1,2,2333,2222,1111,6,7]
>>> z[2:5]=[3,4,5]
>>> print(z)
[1, 2, 3, 4, 5, 6, 7]
```

4. 查

`index()` 用于查看某一元素所在下标：

```
>>> x=['P', 'y', 't', 'h', 'o', 'n']
>>> x.index('y')
1
```

如果有多个相同元素，则取最小下标：

```
>>> z=[1,2,3,2,1]
>>> z.index(2)
1
```

`count()` 方法用于统计列表中所含特定元素的个数：

```
>>> x=[1,2,3,2,1,4,5,4,5,6]
>>> x.count(1)
2
```

```
>>> x.count(5)
```

```
2
```

```
>>> x.count(6)
```

```
1
```

列表推导式

列表推导式可以代替简单的循环，使得你的代码更简洁，更具 `Python` 风，下面以一个栗子来学习一下列表推导式的用法。

问题：求 `1` 到 `10` 内的所有奇数

你会很自然的想到使用循环，就像下面的代码一样：

```
>>> for i in range(1,11):
```

```
    if i%2!=0:
```

```
        print(i)
```

```
1
```

```
3
```

```
5
```

```
7
```

```
9
```

使用列表推导式可以让这一过程写起来更简单：

```
>>> ta=[i for i in range(1,11) if i %2!=0]
```

```
>>> print(ta)
```

```
[1, 3, 5, 7, 9]
```

对于刚开始接触列表推导式的同学，看不太懂也没关系，以后多加练习，有了语感，便能灵活运用了。

这里再给一个栗子，它用于将所有大于 `2` 的元素存到一个新的列表中：

```
>>> x=[]
```

```
>>> for i in [1,2,3,2,5]:
```

```
    if i>2:
```

```
        x.append(i)
```

上面的循环代码等价于下面使用了列表生成式的代码：

```
x=[i for i in [1,2,3,2,5] if i>2]
```

看，列表推导式一行代码就能搞定，真香！

Day 10 字典，寻寻觅觅

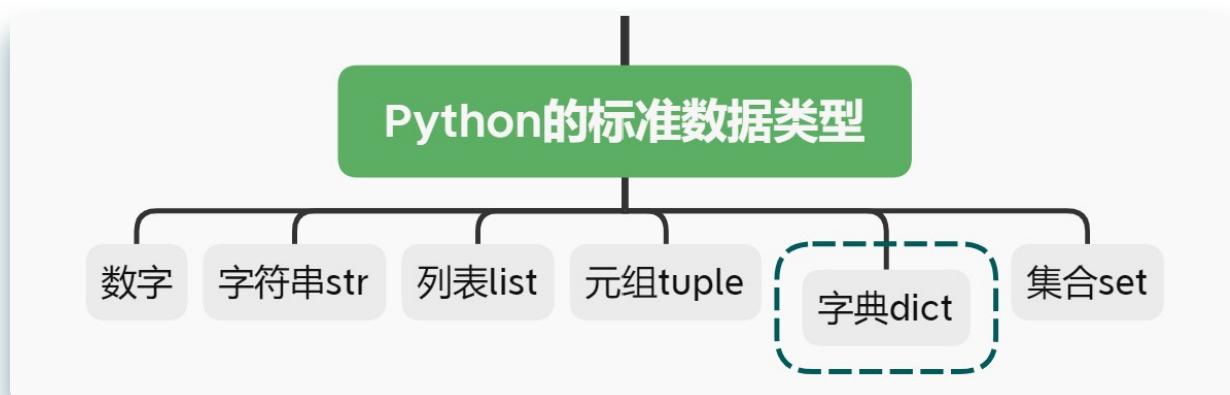


今天你将学到以下内容：

- 1. 初识字典
- 2. 创建字典
- 3. 字典的增删改查
- 4. 字典的遍历方法
- 5. 字典推导式

”

开篇



以前在查阅纸质字典时，有一种音序查字法，可以根据汉字的拼音将查询范围缩小到几页纸中，然后逐页查找目标字。也就是说，一个拼音可以对应多个汉字。

本期所讲的字典数据类型，也是根据某种东西（`key`）去查询另一种东西（`value`），但与上述过程稍微有些不同，至于不同之处嘛，我写在文中咯。

需要说明的是，字典中的许多方法名都类似于列表中的方法名，大家可以对比着学习，同时也不要搞混了哦~

让我们开始吧！

初识字典

字典用 `{}` 包裹，比如 `d={'name': 'Bob', 'age': 13, 'gender': 'male'}` 便是一个字典：

```
d={'name':'Bob', 'age':13, 'gender':'male'}  
      key        value      key        value      key        value
```

观察上面的字典 `d`，可以归纳出字典的基本结构：

`:` 前面的被称为`key`（键），比如`'name'

`:` 后面的被称为`value`（值），比如`'Bob'

键和值合起来被称为一个 `键值对`，比如 `'name': 'Bob'`

各个 `键值对` 之间用逗号 `,` 分隔开。

可以根据 `key` 来查找对应的 `value`：

```
>>> d={'name':'Bob', 'age':13, 'gender':'male'}  
>>> d['name']  
'Bob'
```

创建字典

- **方法1.**使用 `{}` 创建一个字典

【栗子1】创建一个空字典

```
>>> z={}  
>>> print(z)  
{}  
>>> print(type(z)) #type()用于查看变量所存内容的数据类型  
<class 'dict'>
```

【栗子2】创建一个非空字典

```
>>> zz={'height':188, 'age':3, 'city':'Mars'}  
>>> print(zz)  
{'height': 188, 'age': 3, 'city': 'Mars'}
```

- **方法2:** 使用 `dict()` 创建一个字典

【栗子1】创建一个空字典

```
>>> x=dict()
>>> print(x)
{}
```

【栗子2】创建一个非空字典

#1. 使用`变量名=值`的方法

```
>>> a=dict(x=1,y=2,z=3)
>>> print(a){'x': 1, 'y': 2, 'z': 3}
```

#2. 使用zip函数做一对一映射

```
>>> zi=zip(['x','y','z'],[1,2,3])
>>> print(list(zi))#查看zip函数作用后的结果
[('x', 1), ('y', 2), ('z', 3)]
>>> b=dict(zi)
>>> print(b)
{'x': 1, 'y': 2, 'z': 3}
```

#3. 使用可迭代对象#列表内元素为可迭代对象: 元组

```
>>> lis=[('x',1),('y',2),('z',3)]
>>> lis[('x', 1), ('y', 2), ('z', 3)]
>>> c=dict(lis)
>>> print(c)
{'x': 1, 'y': 2, 'z': 3}
```

#列表内元素为可迭代对象: 列表

```
>>> lis=[[ 'x',1],[ 'y',2],[ 'z',3]]
>>> lis[['x', 1], ['y', 2], ['z', 3]]
>>> d=dict(lis)
>>> print(d)
{'x': 1, 'y': 2, 'z': 3}
```

- **方法3:** 使用 `dict.fromkeys()` 创建字典, 常用于字典的初始化

```
>>> key=['a','b','c']
>>> v=[1,2,3]
>>> dict.fromkeys(key,v)
{'a': [1, 2, 3], 'b': [1, 2, 3], 'c': [1, 2, 3]}
```

看，每个 `key` 对应的 `value` 都被初始化为 `[1, 2, 3]`。

注意，若想要拿到初始化之后的结果，必须将其赋值给一个新的变量，看这个栗子：

```
>>> d={}
>>> d.fromkeys(['a','b','c'],666)
{'a': 666, 'b': 666, 'c': 666}
>>> print(d)#依旧为空
{}
```

正确的做法是这样的：

```
>>> d={}
>>> dd=d.fromkeys(['a','b','c'],666)
>>> print(dd)
{'a': 666, 'b': 666, 'c': 666}
```

最后需要注意的是，若一个字典中有重复的 `key`，则后面的 `key` 对应的 `value` 会将前面的 `key` 对应的 `value` 覆盖。看个栗子：

```
>>> d={'x':1,'y':2,'z':3,'x':4}
>>> d
{'x': 4, 'y': 2, 'z': 3}
```

字典的增删改查

和列表一样，`字典`也是可变数据类型，因此对字典的修改操作同样也是直接在原字典上进行。

1. 增

可以直接使用 `dict[key]=value` 来添加一个新的键值对，举个栗子：

```
>>> z={'a':1,'b':2}
#添加一个键值对
>>> z['c']=3
>>> print(z)
{'a': 1, 'b': 2, 'c': 3}
```

还有一种 `update()` 方法，它可以将一个字典添加到另外一个字典中：

```
>>> d1={'a':1,'b':2,'c':3}
>>> d2={'d':4,'e':5}
>>> d1.update(d2)
>>> print(d1)
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

2. 删

- `del`

使用 `del` 可以删除某个键值对：

```
>>> z={'a': 1, 'b': 2, 'c': 3}
>>> del z
['a']
>>> print(z)
{'b': 2, 'c': 3}
```

也可以直接删除整个字典（不是变空，而是抹去）：

```
>>> z
{'b': 2, 'c': 3}
>>> del z
>>> z
Traceback (most recent call last): File "<pyshell#110>", line 1, in <module>      zNameError: name 'z' is no
```

- `pop()`

删除某个键值对，并返回被删除键值对的 `value`。

```
>>> d={'x':1,'y':2}
>>> d.pop('x')
1
>>> print(d)
{'y': 2}
```

- `popitem()`

每次运行都只能删除字典最末尾的元素，并且在字典变空时，若继续使用 `popitem()` 则会报错：

```
>>> s={'a':1,'b':2,'c':3}
>>> s.popitem()
('c', 3)
>>> print(s)
{'a': 1, 'b': 2}
>>> s.popitem()
('b', 2)
>>> print(s)
{'a': 1}
>>> s.popitem()
('a', 1)
>>> print(s){}
>>> s.popitem()
Traceback (most recent call last): File "<pyshell#144>", line 1, in <module>     s.popitem()KeyError: 'popi
```

- `clear()`

用于清空整个字典，直接看栗子

```
>>> d=dict(zip(['a','b'],list(range(2))))
>>> print(d)
{'a': 0, 'b': 1}
>>> d.clear()#清空字典d
>>> print(d)
{}
```

3. 改

如果你尝试向字典中增加键值对，而恰巧你所要添加的 `key` 已经存在于字典中了，那么此时你新添加 `key` 对应的 `value` 会覆盖原来的 `value`，也就达到了修改的目的。

```
>>> d
{'a': 0, 'b': 1}
>>> d['a']='hello'#修改
>>> print(d)
{'a': 'hello', 'b': 1}
```

4. 查

可以使用 `key` 来查询对应的 `value`，就像查纸质字典一样，只不过在 `Python` 的字典中，一个 `key` 只能对应一个 `value`，而纸质字典中一个 `key` (拼音)可以对应多个 `value` (具有相同拼音的汉字)。这便是开篇中提到的问题的答案。

```
>>> dd
{'a': 666, 'b': 666}
>>> dd['a']#查询key为'a'对应的value
666
```

但是注意：如果要查询的 `key` 并不存在，那么就会报错：

```
>>> dd
{'a': 666, 'b': 666}
>>> dd['c']
Traceback (most recent call last): File "<pyshell#241>", line 1, in <module> dd['c']KeyError: 'c'
```

那能不能在 `key` 不存在时不要报错呢？

`Python` 的字典提供了 `get()` 方法，可以指定在查找失败时返回的值，而不再报错。

使用了 `get()` 方法后，当 `key` 不存在时，默认返回 `None`，可以手动修改返回值。下面以栗子来说明具体的使用方法：

【栗子1】使用默认返回值 `None`

```
>>> dd
{'a': 666, 'b': 666}#尝试查找key为'c'对应的value
>>> x=dd.get('c')#查找失败，返回默认值None
>>> print(x)
None
```

【栗子2】修改返回值

```
>>> dd
{'a': 666, 'b': 666}#尝试查找key为'c'对应的value
>>> x=dd.get('c',996)##查找失败，返回手动设定的数字996
>>> print(x)
996
```

字典的遍历方法

想要对字典进行遍历，首先想到的方法是：

先拿到 `key`，然后用 `key` 去查询其对应的 `value`。在外面套一层循环用来重复此操作，即可完成遍历。

需要知道的是：

通过 `dict.keys()` 可以拿到所有的 `key`

通过`dict.values()`可拿到所有的`value`

所以我们可以这样遍历：

```
d={'a':1,'b':2,'c':3}  
for key in d.keys():  
    print(key,'=',d[key])
```

输出

```
a = 1  
b = 2  
c = 3
```

事实上，直接写成下面这样也可以得到相同的结果：

```
d={'a':1,'b':2,'c':3}  
for key in d:  
    print(key,'=',d[key])
```

原因是：

在对字典进行遍历时，默认就是对所有的 `key` 进行遍历操作，所以这里的 `for key in d` 就等价于 `for key in d.keys()`。

另外，`Python` 中的字典还提供了 `items()` 方法，用来获取键值对：

```
>>> d  
{'a': 1, 'b': 2, 'c': 3}  
>>> d.items()  
dict_items([('a', 1), ('b', 2), ('c', 3)])
```

所以，我们也可以通过 `items()` 方法先获取键值对，然后对键值对进行遍历，从而也就完成了对整个字典的遍历操作。请看下面的栗子：

```
d={'a':1,'b':2,'c':3}
for i in d.items():
    print(i[0],'=',i[1])
```

输出

```
a = 1
b = 2
c = 3
```

看，和上面的输出是一样的。

你甚至还可以做拆包（在下面会解释什么是拆包），直接同时获取 `key` 和 `value`，遍历之，也可以得到上面的输出结果：

```
d={'a':1,'b':2,'c':3}
for k,v in d.items():
    print(k,'=',v)
```

所谓拆包，就是将一个可迭代对象中的元素拆解成多个变量。

以列表这个可迭代对象为例来说，我们将多个元素用列表存储是为了减少变量的个数，而拆包的目的正好相反，它是将一个列表中的多个元素分别用一个单独的变量来表示，从而变量的个数会增加。

举个栗子就清楚啦：

【栗子1】对字符串进行拆包

```
>>> s='abc'
>>> x,y,z=s #变量个数由1变3(增)
>>> print(x,y,z)
a b c
```

【栗子2】对列表进行拆包

```
>>> lis=[1,2]
```

```
>>> x,y=ls#变量个数由1变2（增）  
>>> print(x,y)  
1 2
```

现在你已经了解了什么是拆包，我们可以来解释一下 `for k,v in d.items()` 所做的事情了：

首先 `d.items()` 的内容为 `dict_items([('a', 1), ('b', 2), ('c', 3)])`，这是一个可迭代对象，我们先来验证这一点：

```
>>> from collections.abc import Iterable  
>>> isinstance(d.items(),Iterable)  
True
```

返回值为 `True`，是可迭代对象，验证完毕（ps:忘记验证方法的同学可以翻看列表那一期的内容）。

所以这段代码

```
d={'a':1,'b':2,'c':3}  
for k,v in d.items():  
    print(k,'=',v)
```

所做事情的具体过程如下：

在做第1次循环时，`k,v=d.items()`等价于`k,v=('a',1)`，于是`k='a',v=1`，打印输出`'a'=1`

在做第2次循环时，`k,v=d.items()`等价于`k,v=('b',2)`，于是`k='b',v=2`，打印输出`'b'=2`

在做第3次循环时，`k,v=d.items()`等价于`k,v=('c',3)`，于是`k='c',v=3`，打印输出`'c'=3`

字典推导式

和列表一样，字典也有推导式，其存在的原因同样是为了使得代码更加简洁。

问题：互换一个字典中的 `key` 和 `value`。比如 `{'a':1,'b':2,'c':3}` 会变成 `{1:'a',2:'b',3:'c'}`。

若使用 `for` 循环，则代码风是这样的：

```
d={'a':1,'b':2,'c':3}
```

```
reversed_dict={}
for k,v in d.items():
    reversed_dict[v]=k
print(reversed_dict)
```

输出

```
{1: 'a', 2: 'b', 3: 'c'}
```

但是如果使用了字典推导式，则代码风是这样子的：

```
d={'a':1,'b':2,'c':3}
reversed_dict={v:k for k,v in d.items()}
print(reversed_dict)
```

代码看起来是不是清爽很多？

当然，这并不是强制性的。如果你习惯了普通的循环操作，完全可以无视这种推导式的写法。不过我相信，当你熟悉推导式的写法之后会爱上它的简洁与优雅。

Day 11 元组与集合

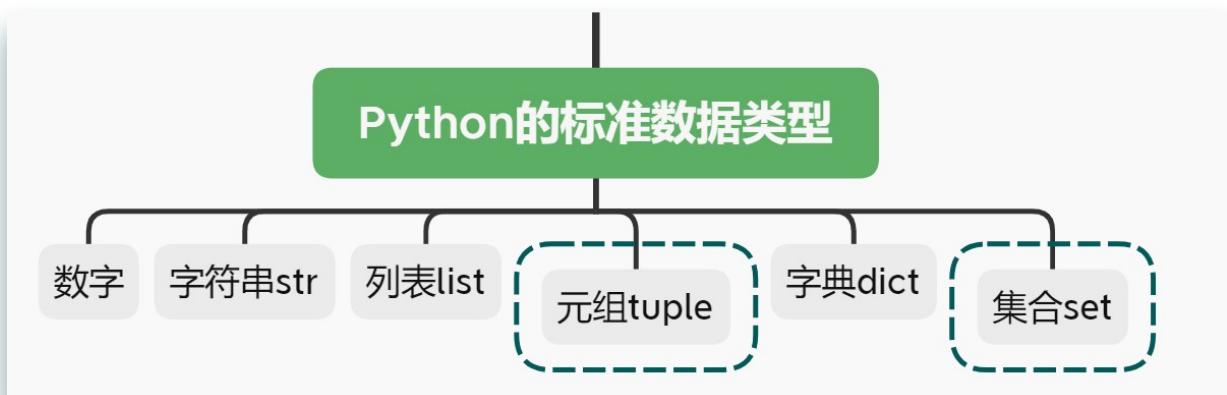


今天你将学到以下内容：

1. 元组及其常用操作
2. 集合及其常用操作

”

开篇



在本期内容中，将介绍 `Python` 六大标准数据类型中最后两种：元组(`tuple`)与集合(`set`)。

元组(tuple)

你已经学习过列表，并且知道列表是可变数据类型。这里要学习的元组和列表非常相似，在掌握列表的使用方法基础之上，再来学习元组会非常容易。

你只需记得元组与列表的不同之处：

1. 元组是不可变数据类型

2. 元组使用`()`包裹元素

元组的不可变特性决定了元组没有增删改等功能，只可查询。

其查询方法归纳如下：

- 1. `index()`

`()`内传入所要查询的元素值，返回该元素值所在的下标，若找不到该元素值，则报错：

```
>>> x=(1,2,3,4,5)
>>> x.index(2)#查找元素2对应的下标1
>>> x.index(2333) #2333不在x中，因此报错
Traceback (most recent call last): File "<pyshell#50>", line 1, in <module>    x.index(2333)
ValueError: t
```

- 2. `count()`

统计某个元素值在元组中出现的个数，若该元素值并不存在于元组中，则返回`0`，表示共找到`0`个：

```
>>> y=[1,2,1,2,3,4,5]
>>> y.count(2) #统计2出现的次数
2
>>> y.count(2333)
0
```

元组中有一些注意事项需要了解：

【注意1】定义一个空元组可以使用 `x=()` 或者 `x=tuple()`，但是在定义只含有一个元素的元组时，需要在元素后面添加一个逗号 `,`，以便和数字区分。看栗子：

```
>>> z=(1)
>>> zz=(1,)
>>> type(z)
<class 'int'>
>>> type(zz)
<class 'tuple'>
```

看，加了 `,` 的变量 `zz` 才是元组类型。

【注意2】虽然元组不可变，其内部元素在定义元组时就固化了，我们不能对其做元素做删除操作。但是，我们可以直接删除整个元组，这并不属于对元组内的元素的删除操作，因此是可行的。

使用 `del()` 可以删除整个元组：

```
>>> x
(1, 2, 3, 4, 5)
>>> del x
>>> x
Traceback (most recent call last): File "<pyshell#71>", line 1, in <module>     xNameError: name 'x' is not
```

和字符串的删除和覆盖操作类似，这里的元组也是被抹去了存在，但初心不改（意思是元组内的元素在整个生命周期从未变动过）。

集合set

集合用 `{}` 来包裹其所含元素，看起来和字典有些许相似，但实际上它们是两个截然不同的东西。

字典中的元素是键值对，而集合中的元素都是单个元素。

举个集合的栗子：

```
>>> z={1,2,'a'}
>>> type(z)
<class 'set'>
```

在高中数学中，你已经学习过集合，那里的集合具有 **无序性**、**不可重复性**，这些特点同样适用于 **Python** 中的集合(**set**)。

正是集合的无序性，导致其不能按照下标来进行查询操作。

但集合是可变数据类型，因此可以对其做一系列的修改操作：

- 1. **add()**

添加元素到一个随机位置

```
>>> z{1, 2, 'a'  
>>> z.add('hello')  
>>> z  
{1, 2, 'hello', 'a'}
```

- 2. **pop()**

随机删除一个元素，并返回被删除的元素值

```
>>> z  
{1, 2, 'hello', 'a'}  
>>> z.pop()  
1  
>>> z  
{2, 'hello', 'a'}
```

- 3. **remove**

删除具有特定值的元素，不存在则报错

```
>>> z  
{2, 'hello', 'a'}  
>>> z.remove('a')  
>>> z  
{2, 'hello'}  
>>> z.remove('sdf')#不存在就报错  
Traceback (most recent call last): File "<pyshell#94>", line 1, in <module> z.remove('sdf')KeyError: 's'
```

- 4. **discard()**

和 **remove** 的功能用法一样，只是待删除元素不存在于集合中时并不会报错，而是什么都不做

```
>>> z  
{2, 'hello'}  
>>> z.discard('sdf')#尝试删除`sdf`  
>>> z  
{2, 'hello'}
```

- 5. `update`

`A.update(B)` 表示将集合 `B` 拼接到集合 `A` 中

```
>>> A={1,2,3}  
>>> B={'a','b','c'}  
>>> A.update(B)  
>>> print(A)  
{1, 2, 3, 'c', 'b', 'a'}  
>>> print(B)  
{'c', 'b', 'a'}
```

操作之后，`A` 是拼接后的集合，而 `B` 不变。

(ps:这里的顺序乱了，这是集合的无序性造成的)

- 6. `union()`

上面的 `update()` 方法用于将一个集合拼接到另一个集合中，而这里的 `union()` 可以将多个集合拼接在一起，形成一个新的集合。

```
>>> A={1,2,3}  
>>> B={'a','b','c'}  
>>> C={11,22,33}  
>>> A.union(B,C)  
{1, 2, 3, 33, 11, 'a', 'c', 22, 'b'}
```

此时，`A`、`B`、`C` 三个集合都没有改变，这一点和 `update()` 不同：

```
>>> A{1, 2, 3}  
>>> B  
{'c', 'b', 'a'}  
>>> C  
{33, 11, 22}
```

(ps: 顺序乱了，还是因为集合的无序性，并不是集合发生了改变)

所以，当使用 `union()` 方法拼接多个集合时，需要使用一个新的变量来存储拼接后的结果，就像这样：

```
>>> new=A.union(B,C)
>>> print(new)
{1, 2, 3, 33, 11, 'a', 'c', 22, 'b'}
```

以上便是集合常用的一些操作。

类似于数学中的集合，这里的集合(`set`)也可以做运算，比如`交`，`并`等，[这些内容用的不多，了解即可](#)。现在以一个综合的栗子来看一下：

```
>>> A={1,2,3}
>>> B={2,4,5}
>>> A-B #差集
{1, 3}
>>> A&B#交集
{2}
>>> A|B#并集
{1, 2, 3, 4, 5}
>>> A^B#反交集
{1, 3, 4, 5}
```

Day 12 函数，化繁为简



今天你将学到以下内容：

1. 初识函数
2. 函数的返回值
3. 多个参数
4. 多个返回值
5. 形参与实参
6. 默认参数
7. 位置参数
8. 关键字参数
9. 可变参数

开篇

考虑这么一个问题：

求解 1 到 4 之和

到现在，你已经学习了分支循环以及六大数据类型，所以这题很简单啦，你应该能很轻松的写出下面的代码：

```
sum=0  
for i in range(1,5):  
    sum+=i  
print('1+2+3+4=' ,sum)
```

bingo！

那如果问题改为求解 1 到 10 之和呢？

也不难对吧，只需对 `range()` 稍作修改：

```
sum=0  
for i in range(1,11):  
    sum+=i  
print('1+2+...+10=' ,sum)
```

那如果问题改为求解 1 到 100， 1 到 1000 之和呢？

同样是修改 `range()` 中的数字即可！

仔细思考一下，你会发现，上面的几个问题其实可以归纳为一个问题：求和

每个问题的解答代码其实都只是更改了 `range()` 中的一个数字

如果对于每一个问题，我们都去写一段上面的代码，那简直是灾难！因为会出现大量重复的代码。

所以，函数应运而生！

初识函数

函数的结构如下：

def func(x):

定义函数用的关键字 函数名

参数

函数体

其中的 `def` 是关键字， `func` 是可自定义的函数名， `x` 是函数的参数，可以有多个（这里的举的栗子只有一个），函数体内书写你的代码，这段代码规定了函数要做什么事。

如果你是初学者，看了上面的定义之后可能还是一头雾水。没关系，看个栗子感受一下：

我们将开篇中提到求和问题写成函数的形式：

```
def get_sum(x):
    sum=0
    for i in range(1, x+1):
        sum+=i
    print('1+2+...+%d=' %x, sum)
```

我们给函数取了名字叫做 `get_sum`，给函数设置了一个参数 `x`

函数的功能是求解 `1` 到 `x` 之和

注意，写完这个函数之后，什么也看不到，因为你只是定义好了函数，而函数只有在被调用时才会执行函数体内的语句

现在来尝试调用函数，我们分别给 `x` 传入 `4` 和 `10`，用于求解 `1` 到 `4` 之和与 `1` 到 `10` 之和：

```
#求解1到4之和
```

```
get_sum(4)
```

```
#求解1到10之和
```

```
get_sum(10)
```

输出

```
1+2+...+4= 101+2+...+10= 55
```

看，使用了函数之后，代码变得简洁起来。

函数的返回值

其实，每个函数都有返回值。在关键字 `return` 后面写需要返回的变量名。如果函数没有用 `return`，那默认返回 `None`。

返回值也是个值，所以可以将这个值赋值给变量，此时称这个变量是用来接收返回值的变量。

我们上面的求和问题中，函数是没有写 `return` 的，因此函数的返回值应该是 `None`，我们来验证这一点：

```
#求解1到4之和
```

```
c=get_sum(4)
```

```
print(c)
```

输出

```
None
```

我们可以给 `get_sum` 函数指定返回值，比如可以指定返回求和结果 `sum`，于是我们可以稍微修改下函数的定义：

```
def get_sum(x):  
    sum=0  
    for i in range(1, x+1):  
        sum+=i  
    return sum
```

此时，再次调用函数：

```
#求解1到4之和
c=get_sum(4)
print(c)
```

输出

10

多个参数

函数的参数可以有多个，比如求解两数之和，我们可以这样定义函数：

```
#求解两数之和，即：x+y
def add(x,y):
    return x+y
```

这里的参数有两个，分别是 `x` 和 `y`。

多个返回值

函数的返回值也可以有多个，比如同时求解两数之和与两数之差：

```
def cal(x,y):
    add_result=x+y
    minus_result=x-y
    return add_result,minus_result
```

这里分的返回值是两数之和与两数之差。

此时我们可以用一个或者两个变量来接收返回值：

【用两个变量接收返回值】这个很符合我们的思维，因为函数的返回值是两个，所以用两个变量来接收返回值

```
add,minus=cal(1,3)
print(add,minus)
```

输出

4 -2

【用一个变量接收返回值】

```
result=cal(1,3)
print(result)
```

输出

(4, -2)

返回值是一个元组，Python 的灵活性在这里又一次被体现。

形参与实参

```
def get_sum(x):
    sum=0
    for i in range(1,x+1):
        sum+=i
    print(sum)
#求解1到z之和
z=4
get_sum(z)
```

在这个栗子中，get_sum 函数中 () 内的 x 是形参，而 z 是实参。

顾名思义，形参就是形式上的参数，在函数定义好之后，形参并没有任何实际值。只有在调用函数时，才将实参传给形参，此时形参将得到的值收下，之后函数内部凡是出现形参名字的地方都具有传入的实参的值。

总结来说：

形参是函数定义时的参数，实参是函数调用时传给函数的参数

你还要了解，形参和实参是可以同名的，因此上例中的 z 即使改名为 x 也可以得到相同的结果。

请再次体会一下上面的栗子，以加深理解。

默认参数

假设要录入"南极小学"六年级一班全体同学的 姓名、年龄 以及 出生地 信息，将每个同学的信息作为一个字典，将所有同学的信息一起存入列表。

为了方便，假设该班共有 3 个同学，则可写出如下代码：

```
lis=[]
def func(name ,age,place):
    person={}
    person[ 'name']=name
    person[ 'age']=age
    person[ 'place']=place
    lis.append(person)
func('Bob',5,'南极洲')
func('Alice',6,'南极洲')
func('Barry',7,'南极洲')
#遍历列表lis，打印所有同学的信息
for item in lis:
    print(item)
```

输出

```
{'name': 'Bob', 'age': 5, 'place': '南极洲'}{'name': 'Alice', 'age': 6, 'place': '南极洲'}{'name': 'Barry',
```

仔细观察，你会发现，每个同学的出生地都是"南极洲"。既然相同，那就可以将形参 place 设置为默认参数，默认值为"南极洲"，调用函数传参时就可以不写 place 了，这样做可以减少无谓的工作量。

于是代码如下：

```
lis=[]
def func(name ,age,place='南极洲'):
    person={}
    person[ 'name']=name
    person[ 'age']=age
    person[ 'place']=place
    lis.append(person)#只需传入name和age，place使用默认值'南极洲'

func('Bob',5)
func('Alice',6)
func('Barry',7)
#打印所有信息，会得到和上面相同的结果
```

```
for item in lis:  
    print(item)
```

当然，如果此时突然新转来了一个同学“大猫”到该班中，并且大猫同学出生在“北冰洋”，那么就不能使用默认的 `place` 了。解决方法很简单，只需像正常传参那样给 `place` 传入“北冰洋”即可，代码如下：

```
lis=[]  
def func(name ,age,place='南极洲'):  
    person={}  
    person[ 'name']=name  
    person[ 'age']=age  
    person[ 'place']=place  
    lis.append(person)  
  
func('Bob',5)  
func('Alice',6)  
func('Barry',7)  
func('大猫',5,'北冰洋')  
  
#打印所有信息  
for item in lis:  
    print(item)
```

输出结果

```
{'name': 'Bob', 'age': 5, 'place': '南极洲'}{'name': 'Alice', 'age': 6, 'place': '南极洲'}{'name': 'Barry',
```

需要注意的是，在定义一个函数时，如果存在位置参数（这里的 `name` 和 `age` 就是位置参数，位置参数的详细讲解就在下一部分），那么必须将默认参数放到最后，就像本栗子中，将默认参数 `place` 放在了最后。

```
def func(name ,age,place='南极洲'):  
    ...
```

位置参数

其实，在上一部分【默认参数】的栗子中，`name` 和 `age` 就是位置参数。

对于位置参数来说，在传参时，各个参数的实际值与其位置下标必须一一对应，否则就可能导致程序输出信息有误或者程序直接报错。

比如我们在给大猫同学录入信息时可以这样写：

```
func('大猫',5,'北冰洋')
```

当然也可以指明 `'place'` :

```
func('大猫',5,place='北冰洋')
```

但绝对不能更换 `name` 和 `age` 的位置，否则年龄就和姓名颠倒了，看下面的反例：

假设我们在传参时互换了 `name` 和 `age` 的位置

```
func(5,'大猫',place='北冰洋')
```

则输出结果中关于大猫同学的信息就会变为：

```
{'name': 5, 'age': '大猫', 'place': '北冰洋'}
```

这表示的是一个叫 `5` 的同学，年龄为‘大猫’，闹笑话了吧~

而如果在其他程序中也犯这样的错误，可能不只是闹笑话，程序可能直接崩溃（因为有些参数类型不同，在函数内部的语句执行时不能对应到相应的数据类型，从而导致程序报错）。

关键字参数

在传参时，如果不单单传入值，还将对应的参数名字也一并传入，那么这就是关键字参数了。

还是拿南极小学的栗子来说明，如果我们在录入大猫同学的信息时这样传参：

```
func(name='大猫',age=5,place='北冰洋')
```

由于使用了关键字参数，因此也可以这样传参：

```
func(age=5,name='大猫',place='北冰洋')
```

看，调换了 `name` 和 `age` 的位置，仍然可以！

甚至你也可以调换默认参数 `place` 的位置（此时是在修改默认参数的前提下进行的，在这种情况下，默认参数其实也算是一个关键字参数）：

```
func(name='大猫',place='北冰洋',age=5)
```

但是，当位置参数和关键字参数同时存在时，位置参数必须在关键字参数前面。

举个反例：

```
func(age=5,'Bob')
```

这个写法是不被 Python 允许的，因为关键字参数 age 放在了位置参数 name 前面。

可变参数

所谓可变参数，就是指在定义函数时，参数的个数并没有被写死，而是可以动态变化的。在上面所介绍的栗子中，其实都是不可变参数，因为参数的个数在定义函数的那一刻就被固化了

正如不可变参数可分为位置参数和关键字参数那样，可变参数也有可变位置参数和可变关键字参数。

可变位置参数前面加一个 *，可变关键字参数前面加两个 **。

```
def f1(*args):           def f2(**kwargs):
```

...

...

可变位置参数

可变关键字参数

注意这里的 args 和 kwargs 是约定俗成的名字，你也可以改成任何合法的名字，但不推荐。

传入可变位置参数的值会被保存在一个元组中，而传入可变关键字参数的值会被保存在一个字典中。

下面还是举两个栗子来分别说明：

** 1. ** 【可变位置参数实例】

```
def add(a,b):  
    print(a+b)
```

这个函数的功能是求解两数之和，如果我们想要求大于两个的数字之和，比如 3 数之和，4 数之和等，由于不确定到底有多少个数，因此可以在定义函数时设置一个可变参数。

那是设置可变位置参数还是可变关键字参数呢？

其实这个要根据实际情况以及方便性等来确定。我们这里只是求解多个数字的和，并不涉及到其他诸如信息存取的操作，因此为了更方便，可以设置可变位置参数。

于是，最终的函数定义如下：

```
def add(a,b,*args):  
    sum=a+b  
    print('args:',args)  
    for i in args:  
        sum+=i  
    print('求和结果为: ',sum)
```

尝试调用函数：

```
add(1,2,3,4,5)
```

得到如下输出：

```
args: (3, 4, 5)  
求和结果为: 15
```

** 2. ** 【可变关键字参数实例】

还是拿南极小学那个栗子来说，如果不单单可以录入 name、age 和 place，而且还可以录入每个同学的爱好 hobby、未来想从事的职业 job，当然新增的这两个是可选的，同学可以只录入 hobby，可以只录入 job，可以同时录入两者，也可以同时不录入这两者。在这个场景下，就可以可变关键字参数了。

```
lis=[]  
def func(name ,age,place='南极洲',**kwargs):
```

```
person={}
person['name']=name
person['age']=age
person['place']=place
print('kwargs:', kwargs)
person.update(kwargs)
lis.append(person)

func('Bob', 5)
func('Alice', 6, hobby='sing')
func('Barry', 7, hobby='run', job='程序猿')

func('大猫', 5, '北冰洋', job='宇航员')
```

```
#打印所有信息
for item in lis:
    print(item)
```

得到的输出结果如下：

```
kwargs: {}
kwargs: {'hobby': 'sing'}
kwargs: {'hobby': 'run', 'job': '程序猿'}
kwargs: {'job': '宇航员'}
{'name': 'Bob', 'age': 5, 'place': '南极洲'}
{'name': 'Alice', 'age': 6, 'place': '南极洲', 'hobby': 'sing'}
{'name': 'Barry', 'age': 7, 'place': '南极洲', 'hobby': 'run', 'job': '程序猿'}
{'name': '大猫', 'age': 5, 'place': '北冰洋', 'job': '宇航员'}
```

在上面的代码中，`Bob` 同学只录入了必选项，而其余三名同学都或多或少的录入了可选项，因此只有 `Bob` 同学的 `kwargs` 为空字典。

Day 13 高阶函数，匿名函数



今天你将学到以下内容：

1.map

2.filter

3.reduce

4.匿名函数

”

前面已经学习了函数以及函数的各种形式的参数。

本期要介绍的高阶函数也是函数的一种，只不过与普通的函数不同的是，高阶函数的参数可以是另外一个函数，高阶函数的返回值也可以是另外一个函数。

本期你将学习到 `Python` 的 3 个常用高阶函数：

`map()`、`filter()` 和 `reduce()`。

你完全可以把高阶函数看做是一个正常的函数（本来就是），只不过其参数和返回值可以是函数。这样学习起来可能会更加轻松。

map

`map` 用于对某一个可迭代对象中的每一个元素施加同一种操作。

`map()` 有两个参数，第一个参数是一个 `函数`，第二个参数是一个 `可迭代对象`。

还是拿栗子来说明：

对于列表 `lis=[1, 2, 3]`，将列表中的每一个元素做平方操作，并将结果存入新的列表

如果采用普通的函数解答，你应该能写出如下的类似代码：

```
lis=[1,2,3]
#求平方的函数
def square(x):
    return x**2
new=[]
#遍历求平方
for x in lis:
    temp=square(x)
    new.append(temp)
print(new)
```

而若使用 `map` 函数，代码是这样子的：

```
lis=[1,2,3]
def square(x):
    return x**2
new=map(square,lis)
print(list(new))
```

看，使用了 `map()` 函数，免去了繁琐的 `for` 循环操作，代码变得非常简洁了。

你可能会说：也没什么大不了的，就多了个循环呗，写就是咯~

其实，当列表中元素个数较多时，使用 `map()` 的优势就体现出来了。

因为 `map()` 函数的返回值是一个 `map object`（至于这是个啥，以后会讲到，这里结合下面的例子来理解即可），并不是存储实际元素的列表。当列表中元素个数较多时，若使用 `for` 循环处理该问题，则会得到一个很大的列表，即使你只是想取结果列表中的一部分元素，也会将全部元素存储起来，此时会占用很多内存，造成内存资源的浪费，而使用 `map` 则不会出现以上问题。

为了加深理解，我们来对比一下分别使用 `for` 循环和 `map` 函数所得结果占用的内存空间大小：

【补充】在 `Python` 中查看变量所占内存大小的方法

```
import sys  
sys.getsizeof(x)
```

其中的 `x` 就是你的变量，`sys.getsizeof(x)` 的返回值就是变量 `x` 所占内存空间大小。

现在，将最开始的问题中的 `lis` 由 `lis=[1, 2, 3]` 改为 `lis=list(range(1, 10000 + 1))`，也就是 `lis=[1, 2, 3, ..., 9999, 10000]`，其余条件不变，重新作答。

[1]. 使用 `for` 循环的方法

```
import sys  
lis=list(range(1,10000+1))  
  
#求平方的函数  
def square(x):  
    return x**2  
  
new=[]  
  
#遍历求平方  
for x in lis:  
    temp=square(x)  
    new.append(temp)  
m=sys.getsizeof(new)  
print("循环方法所得结果占用内存: ",m)
```

输出

循环方法所得结果占用内存： 87624

[2]. 使用 `map()` 函数

```
import sys  
lis=list(range(1,10000+1))  
#求平方的函数  
def square(x):  
    return x**2  
new=map(square,lis)  
m=sys.getsizeof(new)  
print("循环方法所得结果占用内存: ",m)
```

输出

循环方法所得结果占用内存： 56

对比来看，使用 `map()` 函数所占内存更少。

如果想要取到 `map()` 的结果，可以使用 `list()` 将其返回值包裹，也就是将其返回值转为列表类型，这样就能得到和【for 循环方法】中的列表 `new` 一样的结果了。

当然，你可以选择只取前 `n` 个，下面的函数就是用于选取前 `n` 个元素：

```
def get_item(n):  
    for it in new:  
        if n>0:  
            n=n-1  
            print(it,end=' ')
```

比如如果只想获取前 `5` 个元素，就可以这样调用函数：

```
get_item(5)
```

得到结果

1 4 9 16 25

此时，其余未被选取的元素并不会占用内存。

这样就避免了内存资源的浪费。

【温馨提示】如果你尝试打印【for循环方法】中的列表 new，你的 IDLE 可能会卡死，因为数据量有点大。

接下来要讲的 filter 和 reduce 的原理和 map 差不多，大家注意对照着学习。

filter

filter 用于从某一个可迭代对象中筛选满足特定条件的元素。

举个例子，求出 1 到 20 以内所有能被 3 整除的数字，要求使用 filter

我们可以这样写：

```
lis=[i for i in range(1,20+1)]  
def func(x):  
    if x%3==0:  
        return True  
  
#求解  
res=filter(func,lis)  
print(list(res))
```

输出

```
[3, 6, 9, 12, 15, 18]
```

在上面的代码中，我们定义了一个函数 func，用于判断一个数字是不是可以被 3 整除，如果能，就返回 True，如果不能被 3 整除，此时就没有指定返回值，那么返回值就是默认值 None，而 None 的布尔值为 False，也就“相当于”在不能被 3 整除时返回了 False。

接下来使用了 filter 求解，filter 函数的第一个参数是函数 func，第二个参数是可迭代对象 lis。filter(func,lis) 的作用就是对 lis 中的每一个元素都施加 func 函数内的操作，也就是把 lis 中的每一个元素都作为参数传入 func 中。如果返回值为 True，就说明该元素满足筛选条件，于是被选中，否则该元素将不会被选中。最后，filter 函数会返回一个 filter object，你同样可以使用上面介绍的 map 中的操作方法来操作它，这里就不再重复说明了。

reduce

reduce 在使用前需要写一句代码：

```
from functools import reduce
```

它的作用是引入 `reduce`。由于我们还没有讲到模块和包，因此如果你是纯小白第一次接触，只需了解在使用 `reduce` 之前需要写这么一句代码就可以了。

`reduce` 函数同样有两个参数，第一个参数位置需要传入一个函数，第二个参数位置需要传入一个可迭代对象。

`reduce` 的作用是对可迭代对象中的元素做*累计*操作并返回累计值。这个累计，可以是累加、累乘等等。

比如可迭代对象为 `lis=[1,2,3,4,5]`，现在要对 `lis` 做累加求和运算，可以使用 `reduce`：

```
from functools import reduce
lis=[1,2,3,4,5]
def add(a,b):
    return a+b
res=reduce(add,lis)
print(res)
```

在上面的代码中，我们定义了一个函数 `add`，用于求解两数之和。

而 `reduce` 函数内部所做的事情如下：

第一步，将 `lis` 中的1和2传入函数 `add`，得到返回值3

第二步，将第一步中得到的返回值3与 `lis` 中的元素3传入函数 `add`，得到返回值6

第三步，将第二步中得到的返回值6与 `lis` 中的元素4传入函数 `add`，得到返回值10

第四步，将第三步得到的返回值10与 `lis` 中的元素5传入函数 `add`，得到返回值15，执行结束，最终的“累计”结果便是15

匿名函数

匿名函数可以将一些逻辑比较简单的函数写成一句代码，从而使得代码变得更加简洁。

前面我们定义的函数都是使用的关键字 `def`，后接函数名，而匿名函数就是指那些没有名字的函数，既然没有名字，那么就不能使用 `def` 来定义了。

匿名函数使用 `lambda` 来定义，具体格式如下：

lambda 参数：表达式

其中的参数可以有多个。

举个例子，求解一个数的立方

如果使用 `def`，可以这样写：

```
def cube(x):  
    return x**3
```

调用时是这样子的：

```
>>> cube(3)  
27
```

现在，要求你使用匿名函数，则定义函数只需一行代码：

```
lambda x:x**3
```

其中，`:`前面的 `x` 是参数，`:`后面的 `x` 是表达式，相当于 `def` 定义的函数体内的语句。

那如何调用匿名函数呢？

可以这样：

```
>>> (lambda x:x**3)(3)  
27
```

额，可能看起来有点奇怪，来解释下：

`lambda x:x**3` 这一个整体其实就是一个函数，我们可以用 `type()` 来验证这一点：
`
`

```
>>> type(lambda x:x**3)  
<class 'function'>
```

既然是函数，那么就可以和正常函数一样调用，唯一的区别就是匿名函数没有名字而已。正因为没有名字，所以在调用时要用 `()` 将整个函数包裹起来，以表明这是一个整体。我们已经知道了这个整体是个函数，于是在这个整体的后面写一对 `()`，`()` 内部传入参数，就完成了匿名函数的调用过程。

在上面的例子中，我们传入的参数就是数字 `3`，调用该匿名函数就可以求解 `3` 的立方了。

其实，[函数也可以赋值给一个变量](#)。知道这一点的话，我们就可以给匿名函数起名字了。

接着看上面的栗子，我们将 `lambda x:x**3` 这个函数赋值给变量 `f`，即：

```
f=lambda x:x**3
```

这样，我们就完成了给匿名函数起名字的操作，这里，函数的名字就叫做 `f`。

现在，你就可以像调用 `def` 定义的函数那样对函数 `lambda x:x**3` 进行调用了：

```
>>> f(3)#求3的立方
27
>>> f(2)#求2的立方
8
```

说了这么多，你可能会问：这跟今天要学习的高阶函数有什么关系？

其实一开始就说了，匿名函数可以将逻辑比较简单的函数写成一句代码，从而使得代码更加简洁。

我们可以将最开始求平方的那个栗子用匿名函数改写

先回顾一下原代码：

```
lis=[1,2,3]
def square(x):
    return x**2
new=map(square,lis)
print(list(new))
```

接着来使用匿名函数：

```
lis=[1,2,3]
f=lambda x:x**2
new=map(f,lis)
print(list(new))
```

但这不是最正宗的，因为匿名函数嘛，要匿名啊！这里还给匿名函数起了个名字叫 `f`，最后才传入了 `map`。

于是最终的代码如下：

```
lis=[1,2,3]
new=map(lambda x:x**2,lis)
print(list(new))
```

以后，在使用高阶函数时，你将经常看到如上代码所示形式的匿名函数。

Day 14 排序，不止于升降



今天你将学到以下内容：

1. 简单的列表(`list`)排序
2. 简单的集合(`set`)排序
3. 字典(`dict`)排序
4. 更综合的排序问题

”

开篇

本期将介绍排序方法，注意是方法而不是算法，因此更侧重方法的使用，而不对其内部细节的实现原理进行深究。

简单的列表(`list`)排序

`list` 自带有 `sort()` 方法可实现排序

默认是升序排列：

```
>>> lis=[1,3,2,5,4,8,6,9]
>>> lis.sort()
```

```
>>> lis  
[1, 2, 3, 4, 5, 6, 8, 9]
```

可通过传入 `reverse=True` 来实现降序：

```
>>> lis=[1,3,2,5,4,8,6,9]  
>>> lis.sort(reverse=True)  
>>> lis  
[9, 8, 6, 5, 4, 3, 2, 1]
```

你应该已经发现，上述的排序操作是直接在原列表中进行的。

除了上面的 `sort()`，`Python` 语言本身也有一种排序的函数，叫做 `sorted()`

同样默认是升序排列：

```
>>> lis=[1,3,2,5,4,8,6,9]  
>>> sorted_lis=sorted(lis)  
>>> sorted_lis  
[1, 2, 3, 4, 5, 6, 8, 9]
```

同样可传入 `reverse=True` 来实现降序排列：

```
>>> lis=[1,3,2,5,4,8,6,9]  
>>> sorted_lis=sorted(lis,reverse=True)  
>>> sorted_lis  
[9, 8, 6, 5, 4, 3, 2, 1]
```

与 `sort()` 方法不同，`sorted()` 并不是直接将原列表做排序，而是需要用一个新的变量来存储排序后的列表。

简单的集合(set)排序

集合本身是没有排序的方法的，所以只能使用系统自带的 `sorted()` 函数。

```
>>> s={3,2,12345,55,44,345,23456,23}  
>>> ss=sorted(s)  
>>> ss  
[2, 3, 23, 44, 55, 345, 12345, 23456]
```

这样就完成了升序排列，当然，可以传入 `reverse=True` 来实现降序排列：

```
>>> s={3,2,12345,55,44,345,23456,23}
>>> sorted(s,reverse=True)
[23456, 12345, 345, 55, 44, 23, 3, 2]
```

当然，你可能会想到先将集合转为列表，然后使用列表自带的排序方法 `sort()`，最后将有序列表再转回集合。但是，在最后这一步，也就是有序列表转回集合时，由于集合的无序性，可能会导致顺序被打乱。所以不要这样做。

简单的元组(tuple)排序

```
>>> x=(1,3,2,5,4)
>>> z=sorted(x)
>>> z
[1, 2, 3, 4, 5]
>>> type(x)
<class 'tuple'>
>>> type(z)
<class 'list'>
```

看，在使用 `sorted()` 函数时，返回的是一个列表，你只需外面包裹一层 `tuple()`，便可以转回元组了。

本质上还是对列表的排序。

字典(dict)排序

和集合一样，字典本身也没有排序方法，所以还是使用 `sorted()` 进行排序。

但不同的是，之前我们的排序对象都是一整个元素组成的序列，比如列表中的每一个元素组成一个列表，集合中的每一个元素组成了一个集合，而组成字典的每一个元素不是一整个元素，而是由两个小部分组成，一个是键 `(key)`，一个是值 `(value)`。

那怎么办呢？

幸好，不管是内置的 `sorted()` 还是列表独有的 `sort()` 方法，都提供了一个 `key` 参数，它可以让我们将按照每一个元素的哪一个部分进行排序。

所以这里我们将借助 `sorted()` 中的参数 `key` 对字典进行排序。

还记得之前的高阶函数吗？这里的sorted函数其实也是一种高阶函数，因为key后面需要传入一个函数，key用来接收该函数的返回值，该函数规定了排序的规则。

举个例子，假设字典d的组成如下：

```
d={'key1':4,'key2':2,'key3':5,'key4':3}
```

现在想要按照value升序排列

此时排序代码的整体框架为

```
sorted(d.items(),key=...,reverse=True)
```

其中使用d.items()是为了获取所有的键值对（如果写成`d`，那就是默认对键进行排序，而并非对整个字典做排序）

```
>>> d.items()  
dict_items([('key1', 4), ('key2', 2), ('key3', 5), ('key4', 3)])
```

而key（注意不要和字典的键搞混了）是用来接收函数的返回值的，该返回值就是我们指定的排序的依据，比如在本例中，该返回值就是字典中每个键值对的值。

所以，我们要想办法在key后面的...位置需要传入一个函数

那也不难，我们可以定义一个函数：

```
def compare(x):  
    return x[1]
```

然后将函数传给参数key即可：

```
dd=sorted(d.items(),key=compare)  
print(dd)
```

得到输出

```
[('key2', 2), ('key4', 3), ('key1', 4), ('key3', 5)]
```

这是一个列表，我们可以将其转为字典：

```
>>> new=dict(dd)
>>> print(new)
{'key2': 2, 'key4': 3, 'key1': 4, 'key3': 5}
```

至此，我们就完成了对字典的排序。

看完上面的过程，如果你仍有疑惑，我猜应该是在定义函数 `compare` 那里吧，下面来解释一下：

```
sorted(d.items(),key=...,reverse=True)
```

`key` 后面是一个函数，该函数的参数 `x` 是待排序的每一个元素 在本例中，待排序的每一个元素就是 `d.items()` 中的每一个元素，也就是一个个键值对

```
('key1', 4)
('key2', 2)
('key3', 5)
('key4', 3)
```

函数 `compare` 会自动将上面的每一个元组作为参数传到函数内部，然后按照函数内部的语句去做相应操作，最终用 `key` 接收返回值。

由于是按照值 `(value)` 的大小进行排序，因此函数需要返回 `value`，那怎么样拿到 `value` 呢？这就需要我们手写函数内部的代码了，用来规定函数要做的操作具体是什么

很简单，直接取下标为 `1` 处的对应值即可，比如这里的 `('key1', 4)`，下标为 `1` 处的 `4` 便是需要返回的 `value`

所以，在我们定义的函数 `compare` 中，传入了 `x`，而返回了 `x[1]`

解释完毕。

完整代码如下：

```
d={'key1':4,'key2':2,'key3':5,'key4':3}
```

```
#定义函数，传给参数key
def compare(x):
    return x[1]
```

```
#排序  
dd=sorted(d.items(),key=compare)
```

```
#将列表转为字典  
ddd=dict(dd)  
#打印输出  
print(ddd)
```

你应该已经发现，这种函数的定义并不复杂，所以完全可以用一个匿名函数来实现，这样的代码会更加简洁：

```
d={'key1':4,'key2':2,'key3':5,'key4':3}  
dd=sorted(d.items(),key=lambda x:x[1])
```

更综合的排序问题

问题 1：

现在有 3 个同学的年龄和身高

```
lis=[{'age':18,'height':188},{'age':23,'height':200},{'age':22,'height':179}]
```

要求按照年龄降序排列。

【分析】

观察上面的列表，发现其元素是一个个的字典，想要按照年龄排序，那必须获取每个字典中的年龄。

我们的函数可以这样写：

```
#x是lis中的某一个元素，这里也就是某一个字典  
def compare(x):  
    return x['age']
```

只要函数有了，问题就解决了，代码如下：

```
lis=[{'age':18,'height':188},{'age':23,'height':200},{'age':22,'height':179}]
```

```
def compare(x):
```

```
return x['age']

new=sorted(lis,key=compare, reverse=True)

print(new)
```

输出

```
[{'age': 23, 'height': 200}, {'age': 22, 'height': 179}, {'age': 18, 'height': 188}]
```

这样就完成了按照年龄排序。

当然，用匿名函数更简洁：

```
lis=[{'age':18,'height':188},{'age':23,'height':200},{'age':22,'height':179}]

new=sorted(lis,key=lambda x:x[ 'age' ],reverse=True)

print(new)
```

问题 2：

请写出以下代码的运行结果：

```
lis=[3,9,-4,-5,-3,8]
z=sorted(lis,key=lambda x:(x>0,abs(x)))
print(z)
```

这个有两个排序的关键字，一个是 `x > 0`，一个是 `abs(x)`

如果你不清楚，我们可以先将匿名函数转为普通函数：

```
def f(x):
    return (x>0,abs(x))
```

我们不妨先将 `lis` 中的每一个元素都作为参数传给该函数

```
for i in lis:
    print(f(i))
```

于是得到以下结果：

```
(True, 3)
(True, 9)
(False, 4)
(False, 5)
(False, 3)
(True, 8)
```

每个元组的第一个位置（下标 `0`）可以看作是第一个关键字，第二个位置（下标 `1`）可以看作是第二个关键字，它们分别是 `x>0` 和 `abs(x)` 的返回值。

`sorted()` 在进行排序时，会从下标较小的关键字位置（下标 `0`）开始对比，若两者相同，再去比较下一个下标处（下标 `1`）的关键字。

在这个问题中，由于 `False` 小于 `True`，即 `0` 小于 `1`，再加上默认是升序排列，因此会把元组第一个位置为 `False` 的放在前面作为一组，而第一个位置为 `True` 的放在后面作为另一组。

所以，第一组为

```
(False, 4)
(False, 5)
(False, 3)
```

第二组为

```
(True, 3)
(True, 9)
(True, 8)
```

接下来，就说进行组内排序了，首先看排在前面的第一组：

因为此时第一个关键字 `x>0` 均为 `False`，所以尝试比较第二个关键字 `abs(x)`，也是小的在前，大的在后，比较结果为：

```
abs(-3)<abs(-4)<abs(-5)
```

也就是

```
3<4<5
```

所以第一组的排序结果为：

```
[-3, -4, -5]
```

再看后面的第二组，此时第一个关键字都是 `True`，因此需要比较第二个关键字 `abs(x)`，比较结果为：

$3 < 8 < 9$

因此排序结果为

`[3, 8, 9]`

最后组装两组，便可得到最终的排序结果：

`[-3, -4, -5, 3, 8, 9]`

问题 3：

使用集合对列表进行去重，要求不改变列表中元素原来的相对位置，请自行举例说明

【分析】

如果只是单纯去重的话很简单：

```
>>> lis=[2,1,3,2,1,5,7]
>>> x=set(lis)
>>> xx=list(x)
>>> print(xx)
[1, 2, 3, 5, 7]
```

但也很明显，原列表中各个元素的相对位置发生了变化。

那如何才能保持相对位置不变呢？

还是靠我们的老朋友，就是给参数 `key` 传入函数啦！

在上面的代码中，我们已经得到了去重后的列表 `xx`（注意此时的列表不一定有序，这里是巧合），只需要对 `xx` 做个排序就可以了，只不过在这次的排序中，函数无需我们构造，而是可以直接使用 `lis.index` 这个函数，没错，这的确是个函数，我们可以验证：

```
>>> type(lis.index)
<class 'builtin_function_or_method'>
```

所以代码可以接着写：

```
>>> xx.sort(key=lis.index)
>>> print(xx)
```

或者

```
>>> s=sorted(xx,key=lis.index)
>>> print(s)
```

最后都能得到题目所要求的结果：

```
[2, 1, 3, 5, 7]
```

以上就是三道排序的练习题，你掌握了吗？

扯些别的

最后，由问题三出发，想说一些与本期'排序'主题无关的知识：

你如果并不知道有这么个函数 `list.index`，完全可以使用之前讲的循环操作完成列表的去重。代码可以这样写：

```
lis=[2,1,3,2,1,5,7]

xx=[]#去重后的新列表
for i in lis:
    if i not in xx:
        xx.append(i)

print(xx)
```

输出结果

```
[2, 1, 3, 5, 7]
```

通向南极洲的道路不止一条，当无法找寻到最佳的那条道路时，请循着你最熟悉的那一条走下去，这也不失为上策。



今天你将学到以下内容：

1. 为什么需要做异常处理
2. 如何实现异常处理

”

开篇

本期介绍 `Python` 的异常处理方法。

为什么需要做异常处理

看下面的函数：

```
def div(a,b):  
    return a/b
```

这个函数用于求解两数相除的结果

我们可以调用它：

```
#求解1/2  
>>> div(1,2)  
0.5  
  
#求解3/4  
>>> div(3,4)  
0.75
```

程序貌似没问题

但是我们知道，除数是不能为 `0` 的，现在来尝试让除数为 `0`：

```
>>> div(1,0)  
Traceback (most recent call last):  
File "<pyshell#2>", line 1, in <module>  
    div(1,0)
```

```
File "C:/Users/fanxi/Desktop/swe.py", line 2, in div
    return a/b
ZeroDivisionError: division by zero
```

毫无疑问，程序报错了！

报错信息也很明显：`ZeroDivisionError: division by zero`

我们的除数为`0`，所以报错。

你可能会说：那就控制除数不为`0`就好了呀，每次计算时都不要让除数为`0`，这样就不会出错了。

可以，但是，对于大量待计算对的数据来说，人为控制是不现实的，因为数据量太大。

比如将被除数与除数组成一个列表，作为一个整体元素`[被除数, 除数]`，总共`10万`个这样的小列表，将它们一起嵌套在一个大列表中。

就像下面这样：

大列表共10万个元素，也就是10万个小的列表

`[[-20, -16], [-7, 100], [-39, 0], ..., [-23, 93]]`

除数为0

其中的被除数与除数均是如下方式随机生成的：

```
import random
#生成-50到50之间的随机整数
a=random.randint(-50,50)
#生成-100到100之间的随机整数
b=random.randint(-100,100)
```

现在将上面的叙述写成代码：

```
#定义函数
def div(a,b):
    return a/b

import random
```

```
#生成随机整数作为除数与被除数,  
#并分别存入列表a与列表b  
a=[random.randint(-50,50) for i in range(100000)]  
b=[random.randint(-100,100) for i in range(100000)]  
  
#使用`for`循环批量调用`100000`次`div()`函数  
for i in range(100000):  
    result=div(a[i],b[i])  
    print(result)
```

刚开始程序正常执行并打印计算结果：

```
===== RESTART: C:/Users/fanx
0.24731182795698925
0.5952380952380952
0.1566265060240964
0.06349206349206349
0.5714285714285714
0.08888888888888889
0.5517241379310345
-19.0
-0.5897435897435898
23.0
-0.4845360824742268
0.05454545454545454
-0.0
0.1095890410958904
-0.6181818181818182
-0.42105263157894735
-2.7777777777777777
1.0909090909090908
0.5106382978723404
0.8125
-0.9545454545454546
0.13333333333333333
0.02857142857142857
3.076923076923077
-3.4
0.4666666666666667
-1.9090909090909092
-0.9512195121951219
-0.2597402597402597
-1.9444444444444444
0.6172220506172220
```

但是在执行若干步之后，会遇到除数为 0 的情况，程序立马崩溃：

```
0.061224489795918366
-0.15
0.14285714285714285
0.5263157894736842
0.7083333333333334
-0.25
-0.4625
```

```
0.4023  
-0.5588235294117647  
-1.5384615384615385  
0.2708333333333333  
-0.42857142857142855  
-0.6727272727272727  
-0.4387755102040816  
-2.4375  
0.0333333333333333  
0.125  
-1.3448275862068966  
0.3673469387755102  
16.66666666666668  
Traceback (most recent call last):  
  File "C:/Users/fanxi/Desktop/swe.py", line 11, in <module>  
    result=div(a[i], b[i])  
  File "C:/Users/fanxi/Desktop/swe.py", line 2, in div  
    return a/b  
ZeroDivisionError: division by zero
```

之后还有很多没有来的及计算，而我们希望能够完成全部的除法运算，因此，需要引入异常处理。

引入异常处理可以让程序即使出现了除数为0，也不会中断，而是可以继续执行后面的除法运算，最终完成全部的计算。

那怎么实现呢？

别着急，接下来就看一下异常处理的方法~

如何实现异常处理

最简单的是使用 `try..except...`

还是用除法的例子，我们引入异常处理，为了方便观察，我们把随机整数的生成区间改小，代码可以这样写：

```
def div(a,b):  
    try:  
        return a/b  
    except:  
        return '有错误，自动忽略本条计算'  
  
import random  
  
a=[random.randint(-5,5) for i in range(100000)]  
b=[random.randint(-1,1) for i in range(100000)]
```

```
for i in range(100000):
    result=div(a[i],b[i])
    print(result)
```

运行上述代码，当遇到除数为 0，将打印'有错误，自动忽略本条计算'，程序并不会崩溃，而会继续向下执行：

```
5.0
-3.0
-0.0
-2.0
2.0
4.0
有错误，自动忽略本条计算
有错误，自动忽略本条计算
有错误，自动忽略本条计算
2.0
有错误，自动忽略本条计算
-1.0
有错误，自动忽略本条计算
-0.0
1.0
-4.0
有错误，自动忽略本条计算
5.0
-1.0
-4.0
有错误，自动忽略本条计算
有错误，自动忽略本条计算
有错误，自动忽略本条计算
4.0
-4.0
-5.0
-3.0
1.0
3.0
```

你可能会发现，上面的异常处理方式只是告诉我们有错误，但是具体是什么错误呢？

可以使用多个 `except` 语句来设置遇到相应错误之后的提示信息。

那具体怎么操作呢？

先别着急，在学习这个之前，你首先需要了解 `Python` 内置的错误类型：

```
ZeroDivisionError  
FileNotFoundException  
FileExistsError  
ValueError  
KeyError  
SyntaxError  
IndexError
```

我们刚刚遇到的就是第一个错误：`ZeroDivisionError`

现在设置两个错误类型，看一下会不会如我们所想那样报“除数为0”的错误：

```
def div(a,b):  
    try:  
        return a/b  
    except SyntaxError:  
        return '语法错误'  
    except ZeroDivisionError:  
        return '除数为0, 忽略本条计算'  
  
import random  
  
a=[random.randint(-5,5) for i in range(100000)]  
b=[random.randint(-1,1) for i in range(100000)]  
  
for i in range(100000):  
    result=div(a[i],b[i])  
    print(result)
```

输出结果：

```
RESTART: C:/USC
3.0
除数为0, 忽略本条计算
除数为0, 忽略本条计算
除数为0, 忽略本条计算
3.0
-4.0
4.0
2.0
-2.0
-4.0
除数为0, 忽略本条计算
-1.0
除数为0, 忽略本条计算
除数为0, 忽略本条计算
-4.0
-2.0
-2.0
1.0
-0.0
除数为0, 忽略本条计算
-3.0
5.0
除数为0, 忽略本条计算
5.0
除数为0, 忽略本条计算
5.0
2.0
-1.0
-1.0
```

没错，报错信息指向 `ZeroDivisionError`。

当然，`except`之后的报错信息也可以不自己指定，此时如果发生了相应的异常，会打印出`Python`自带的异常提示信息，也就是最开始看到的`division by zero`：

```
def div(a,b):
    try:
        return a/b
    except SyntaxError:
        return '语法错误'
    except ZeroDivisionError as e:
        return e

import random

a=[random.randint(-5,5) for i in range(100000)]
b=[random.randint(-1,1) for i in range(100000)]

for i in range(100000):
    result=div(a[i],b[i])
    print(result)
```

此时的输出结果如下：

```
2.0
-2.0
3.0
division by zero
5.0
-4.0
-2.0
0.0
division by zero
1.0
-1.0
division by zero
```

```
division by zero
5.0
-0.0
-2.0
4.0
5.0
division by zero
3.0
-4.0
-3.0
division by zero
2.0
-2.0
-5.0
```

以上便是 `try...except...` 的基本用法。

你还可以指定在程序不发生异常，也就是正常运行结束（一次也不执行 `except` 中的语句）之后的动作，这会用到 `try...except...else...`。

还是举个除法的栗子：

```
def div(a,b):
    try:
        print(a/b)
    except SyntaxError:
        print('语法错误')
    except ZeroDivisionError as e:
        print(e)
    else:
        print('程序正常结束，这太棒啦！')
```

开始调用：

```
>>> div(1,2)
0.5
程序正常结束，这太棒啦！

>>> div(2,3)
0.6666666666666666
程序正常结束，这太棒啦！
```

此时，除数不为 `0`，程序正常结束运行，最后执行 `else` 中的语句。

而如果除数为 `0`，那 `else` 后的语句将不会被执行：

```
>>> div(12,0)
division by zero
```

你可能还会想：在引入异常处理后，不管程序是正常运行结束，还是产生了异常而执行了 `except` 中的对应语句，我都希望在程序最终结束时执行一些指定的语句，这个应该怎么办呢？

简单！`Python` 提供了 `finally` 关键字，在其后面书写你指定的代码语句即可，这些代码总是在程序结束（无论是否产生异常）后自动运行。接着看除法的栗子：

```
def div(a,b):
    try:
        print(a/b)
    except SyntaxError:
        print('语法错误')
    except ZeroDivisionError as e:
        print(e)
    else:
        print('程序正常结束，这太棒啦！')

    finally:
        print('程序运行结束。')
```

我们设置了 `finally`，此时无论程序如何结束运行，都会在最后打印‘程序运行结束。’

```
>>> div(1,2)
0.5
程序正常结束，这太棒啦！

程序运行结束。
```

```
>>> div(2,0)
```

division by zero

程序运行结束。

以上便是异常处理的基本方式，掌握这些已经能满足一般的需求了。

Day 16 面向对象，站在更高的角度来思考



今天你将学到以下内容：

1. 对比面向过程，初识面向对象
2. 定义一个类
3. 在类中定义方法
4. 从类中实例化对象

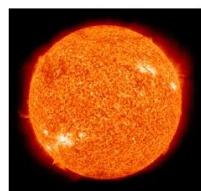
”

开篇

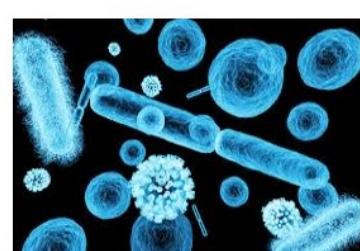
面向过程编程和面向对象编程是两种基本的编程思想。

在这之前，你学习到的都是面向过程编程，即：首先分析出解决问题的每个步骤，然后编写函数实现每个步骤，最后通过合适的顺序依次调用函数来解决问题。

不同于面向过程编程，面向对象编程的思想其实是从自然界的机理中借鉴而来的。正所谓物以类聚，在渺渺的大自然中，有数不胜数的类别，按照不同的星球来分类，可以分为：地球类、太阳类、火星类、月球类、海王星类等等



按照地球上的不同形态的生命，可以分为植物类、动物类、细菌类等等



你还可以按照自己的喜好，划分出自己的分类标准。

使用面向对象编程，可以让你站在一个更高的视角来审视问题，此时，你所解决的不仅仅是一个问题，而是一类问题。

对比面向过程，初识面向对象

在介绍各种概念之前，让我们先做一个小栗子，通过对比面向过程编程的方式去感受面向对象编程的奥妙。

栗子如下：

小明家里养了3只小猫，每到中午饭点，这三只小猫便要吃饭了。吃饭前，小猫们都会先洗手，然后喵喵叫三声。请写一个程序，来分别模拟这三只小猫完整的吃饭过程，部分细节可自行脑部。

分析问题，可以绘制出解决问题的流程图



如果采用面向过程思想进行编程，可以这样写：

首先分别用函数实现以上流程图中的三个步骤

```
import time

#模拟吃饭
def eat(number, food):
    #number: 小猫编号，可取1, 2, 3
    #food: 某编号的小猫所吃食物的名称
    wash(number)#模拟洗手
    miao(number)#模拟喵喵叫三声
    print("{}号小猫正在吃{}".format(number, food))
    time.sleep(5)#吃饭需要5秒钟
    print('{}号小猫进食完毕，睡觉睡觉咯'.format(number))
    print()

#模拟洗手
def wash(number):
    #number: 小猫编号，可取1, 2, 3
    print('{}号小猫正在洗手...'.format(number))
    time.sleep(3)#洗手花费3秒钟
    print('{}号小猫洗手结束'.format(number))
```

```
#模拟喵喵叫三声
```

```
def miao(number):  
    #number:小猫编号，可取1, 2, 3  
    print('第{}号小猫喵喵~'.format(number))
```

有了函数之后，就可以按照合适的序列步骤调用函数来解决问题了：

```
#模拟第一只小猫的吃饭过程
```

```
eat(1, '树叶子')
```

```
#模拟第二只小猫的吃饭过程
```

```
eat(2, '海带丝')
```

```
#模拟第三只小猫的吃饭过程
```

```
eat(3, '石头')
```

得到如下输出：

```
1号小猫正在洗手...
```

```
1号小猫洗手结束
```

```
第1号小猫喵喵~
```

```
1号小猫正在吃树叶子
```

```
1号小猫进食完毕，睡觉睡觉咯
```

```
2号小猫正在洗手...
```

```
2号小猫洗手结束
```

```
第2号小猫喵喵~
```

```
2号小猫正在吃海带丝
```

```
2号小猫进食完毕，睡觉睡觉咯
```

```
3号小猫正在洗手...
```

```
3号小猫洗手结束
```

```
第3号小猫喵喵~
```

```
3号小猫正在吃石头
```

```
3号小猫进食完毕，睡觉睡觉咯
```

而若采用面向对象编程，则可以写出如下代码（稍后会对每一句代码进行解释，这里只是先直观感受下面向对象编程的feel）：

```
import time  
class CatMeal():
```

```

:小猫编号, 可取1, 2, 3
: 某编号的小猫所吃食物的名称

def __init__(self,number,food):
    self.number=number
    self.food=food

#模拟吃饭

def eat(self):
    self.wash()
    self.miao()
    print("{}号小猫正在吃{}".format(self.number,self.food))
    time.sleep(5)#吃饭需要5秒种
    print('{}号小猫进食完毕, 睡觉睡觉咯'.format(self.number))
    print()

#模拟洗手

def wash(self):
    #number:小猫编号, 可取1, 2, 3
    print('{}号小猫正在洗手...'.format(self.number))
    time.sleep(3)#洗手花费3秒种
    print('{}号小猫洗手结束'.format(self.number))

#模拟喵喵叫三声

def miao(self):
    #number:小猫编号, 可取1, 2, 3
    print('第{}号小猫喵喵~'.format(self.number))

#模拟第一只小猫的吃饭过程

miao1=CatMeal(1,'树叶子')
miao1.eat()

##模拟第二只小猫的吃饭过程

miao2=CatMeal(2,'海带丝')
miao2.eat()

#模拟第三只小猫的吃饭过程

miao3=CatMeal(3,'石头')
miao3.eat()

```

得到同样的输出：

```

1号小猫正在洗手...
1号小猫洗手结束
第1号小猫喵喵~

```

[1号小猫正在吃树叶子](#)

[1号小猫进食完毕，睡觉睡觉咯](#)

[2号小猫正在洗手…](#)

[2号小猫洗手结束](#)

[第2号小猫喵喵喵~](#)

[2号小猫正在吃海带丝](#)

[2号小猫进食完毕，睡觉睡觉咯](#)

[3号小猫正在洗手…](#)

[3号小猫洗手结束](#)

[第3号小猫喵喵喵~](#)

[3号小猫正在吃石头](#)

[3号小猫进食完毕，睡觉睡觉咯](#)

在上面的代码中，我们使用关键字 `class` 定义了一个小猫吃饭的类，名字叫做 `CatMeal`，然后实例化出来了 3 只小猫，分别模拟了它们吃饭的完整过程。你可能还是对有些代码，比如 `__init__` 那里搞不清，没关系，先别着急，在下一部分，我们将详解上述代码，在此之前，有一些东西需要讲一下：

如果你阅读过之前的文章，你会发现，那些文章的思路都是先通过介绍某种方法的不足，然后引出正题，比如由于需要用多个变量来存储多个数字，于是引出了列表这一简单高效的数据结构。在这里，我们先介绍了面向过程的方法，之后才介绍了面向对象的方法，那是不是面向对象比面向过程更具有优势呢？

并不是，这两种编程思想各有千秋。

我们可以按照从简到繁的顺序来捋清这一切：

(1)对于一个非常简单的问题，比如计算两数之和，我们甚至可以直接在命令行中敲下 `a+b`，点击回车即可看到问题的答案：

```
>>> 1+2
```

```
3
```

(2)接着将问题变复杂些，这次也是计算两数之和，只不过待计算的数值对不止一个，于是我们可以定义一个函数，专门用于做两数之和：

```
def add(a,b):  
    print('{0}+{1}={2}'.format(a,b,a+b))
```

```
#调用函数计算两数之和
```

```
>>> add(1,2)
```

```
1+2=3
```

(3)然后，将问题更加复杂化，需要一个简易版计算器，可以用来计算加法和减法，并且要求用户只需输入操作数（待运算的数字，比如1, 2, 3），而无需输入运算符（比如+， -）：

如果采用面向过程的思想，由于限制了用户只需输入操作数，因此我们必须同时定义加法和减法两个函数

```
def add(a,b):  
    print('{0}+{1}={2}'.format(a,b,a+b))  
  
def minus(a,b):  
    print('{0}-{1}={2}'.format(a,b,a-b))
```

```
#调用函数
```

```
add(1,2)
```

```
minus(2,1)
```

```
#输出
```

```
1+2=3
```

```
2-1=1
```

而若采用面向对象的思想，我们可以这样写：

```
class Cal():  
    def add(self,a,b):  
        print('{0}+{1}={2}'.format(a,b,a+b))  
    def minus(self,a,b):  
        print('{0}-{1}={2}'.format(a,b,a-b))
```

在上面的代码中，我们定义了一个计算类，名字叫做 `Cal`

接下来从这个类中实例化出来一个对象，名字叫做 `z`

```
z=Cal()  
z.add(1,2)  
z.minus(3,6)
```

```
1+2=3
```

(4)如果现在去除了用户不能输入运算符的限制，那么，使用面向过程编程的代码如下：

```
def add_or_minus(a,b,op):
    if op=='+':
        res=a+b
    elif op=='-':
        res=a-b
    print('{}{}{}={}'.format(a,op,b,res))
```

#调用

```
>>> add_or_minus(1,2,'+')
1+2=3
>>> add_or_minus(1,2,'-')
1-2=-1
```

使用面向对象编程代码如下：

```
class Cal():
    def add_or_minus(self,a,b,op):
        if op=='+':
            res=a+b
        elif op=='-':
            res=a-b
        print('{}{}{}={}'.format(a,op,b,res))
```

#实例化类并计算

```
>>> z=Cal()
>>> z.add_or_minus(1,2,'+')
1+2=3
>>> z.add_or_minus(1,2,'-')
1-2=-1
```

再次对比这两者，可以看出，面向对象编程给人的感觉是代码变得麻烦了些，而面向过程编程则清爽很多。

原因在于，实现加法和减法是个很简单的问题，在这个问题上使用面向对象的编程方式不能很好的体现出面向对象编程的灵活性和规范性，反而简单的面向过程编程更加易

懂、易实现。

在今后，你会看到，当问题规模变大时，面向对象编程将会体现出来显著的优势。

总结来说：面向过程和面向对象这两种编程方式各有千秋，在解决规模较小的问题时使用前者更为方便，而对于大规模问题，后者具有显著优势。所以，对于不同的应用场景，要学会灵活选用不同的编程方式。

面向对象编程的语法

在这一部分，我们将对之前栗子中面向对象代码进行拆分讲解。

定义一个类

在 `class CatMeal():` 中：

`class` 是一个 `Python` 关键字，表明要定义一个类，它的作用就好比定义函数时用到的关键字 `def`

`CatMeal` 是类的名字，一般约定首字母大写的代表类名，而首字母为小写的代表变量命名

在还没有学习后面的内容之前，当你想创建一个新的类时，只需修改类名即可，其余照抄，比如定义一个动物类，可以写 `class Animal():`。当然，在后面的学习中，你会解锁关于这一句代码的更高级用法。

在类中定义方法

我们知道，以关键字 `def` 定义的一个代码块叫做函数，用于完成特定的功能。其实，在类里面，也是使用 `def` 定义一个代码块，同样是用于完成特定的功能，只不过此时的名字不再是“函数”，而是“方法”。在这个栗子中，定义的 `__init__`、`eat`、`wash`、`miao` 都是方法。

`__init__` 方法默认第一个参数是 `self`（约定俗成的叫法），后面的参数可以自行指定，就像给函数传参那样。

在 `__init__` 方法里，以参数形式定义特征，称之为属性。

比如本栗子中的 `number` 和 `food` 就是类的两个属性。

当然，如果像之前的简易计算器栗子一样，不要求在创建类时手动指定属性，那么就可以不显式地写出 `__init__` 方法（可以回看那个栗子来加深理解）。

观察 `eat`、`wash`、`miao` 方法，你会发现，每个方法里面都和 `__init__` 方法一样，第一个参数为 `self`。

其实，无论是 `__init__` 方法也好，其余自定义的方法也罢，它们的第一个参数都必须是 `self`，它代表一个完整的实例化对象。与类相关联的属性和方法都会自动传递给 `self`，通过使用 `self`，可以轻松调用类中的一些属性和方法：

比如在 `eat` 方法中，我们传入了 `self`，于是便可以在该方法中使用 `self.number` 获取在 `__init__` 方法中定义好的 `number` 具体值，同理可以使用 `self.food` 获取 `food` 的具体值。

从类中实例化对象

在编写完成这个 `CatMeal` 类之后，便可以通过类来实例化对象了，比如：

```
miao1=CatMeal(1,'树叶子')
```

运行这一句代码，会从抽象的类中实例化出来一个对象，名字叫做 `miao1`，同时自动调用 `__init__` 方法，并将 `1` 和 `树叶子` 将会分别传给 `__init__` 方法中的 `number` 和 `food`。

其内部具体实现过程如下：先在内存中开辟一块空间，然后调用 `__init__` 方法，并让 `self` 指向那块内存空间，最后让实例化的对象 `miao1` 也指向那块内存空间。

这个实例化的对象 `miao1` 可以通过 `.` 来访问类中的属性和调用类中的方法，比如访问 `food` 属性，可以写 `miao1.food`；调用 `wash` 方法，可以写 `miao1.wash()`。

总结

初学者很容易被一堆 `self` 所困扰，所以这里尽量使用精炼的语言来总结第二部分所讲内容：

在定义类之后，实例化之前，`self` 这个抽象的东西具有访问类中属性和调用类中方法的权限，所以无论是增加属性，还是增加方法，都离不开 `self`。增加属性可以写 `self.new_attribute=new_attribute`，增加方法可以写

```
def new_method(self):  
    pass
```

（可以根据需要自行设定方法中的参数）

比如在定义 `eat`、`wash`、`miao` 方法时，由于还没有实例化对象，因此若要访问类中的属性 `number` 和 `food`，必须借助具有权限的 `self`。因此，每个方法都需要传入 `self` 这个参数，以使用 `self.` 来进行访问，`self` 仿佛就是一把钥匙。

而在实例化（比如 `miao1=CatMeal(1,'树叶子')`）之后，原先 `self` 具有的权限将会复制一份给这个实例化的对象。

之前需要使用 `self.number` 来访问 `number` 属性，现在依然可以，但同时也支持使用 `实例化对象.number` 来进行访问了，比如 `miao1.number`。

Day 17 对象属性与类属性，私有属性与私有方法



今天你将学到以下内容：

1. 属性的增删改查
2. 私有属性和私有方法
3. 类属性和对象属性

”

开篇

上一期介绍了关于面向对象编程的基本语法以及步骤，在此基础上，本期将会介绍更多面向对象编程的概念。

属性的增删改查

通过上一期的学习，你已经知道可以在 `__init__` 方法里定义属性。

这里我们将通过一个栗子，来讲解属性的增删查改操作。

看下面的栗子：

```
class Cat():
    def __init__(self,color,name):
        self.color=color
        self.name=name
```

上面的代码定义了一个类 `Cat`，用来模拟“猫”这个类，`color` 和 `name` 分别代表猫的毛色和名字这两个属性。

接着实例化一只小猫出来，注意在 `__init__` 方法中定义了两个属性，因此在实例化时必须传入这两个属性的值：

```
miao=Cat('white','Mary')
```

这样，一只白色的名字叫做 `Mary` 的小猫就诞生了。

你可以通过 `.` 来访问这只小猫所具有的属性，就像下面这样：

```
print(miao.name,miao.color)
```

输出结果为 `Mary white`

这便是属性的直接查询（访问）方法。

如果此时，我们想给这只小猫换个新名字，可以直接将新的名字赋值给 `name` 属性即可。比如新名字为 `Kite`，则代码如下：

```
>>> new_name='Kite'  
>>> miao.name=new_name  
>>> print(miao.name)  
Kite
```

看，小猫已经换上新名字啦

这便是属性的直接修改方法。

现在我又想给这只小猫定制一套衣服，所以想着增加一个衣服颜色的属性 `clothes_color`。可以直接使用 `.` 为该属性赋值，比如颜色是 `black`，则添加属性的代码如下：

```
miao.clothes_color='black'
```

尝试（查询）访问新增属性：

```
>>> miao.clothes_color  
'black'
```

可以被访问到，说明增加属性成功。

这便是属性的增加方法。

过了一段时间，发现穿着黑色衣服的小猫行动不便，于是决定不再给小猫穿衣服了，因此需要删除之前增加的 `clothes_color` 属性。

这里提供两种删除属性的方法：

其一为 `del`：

```
>>> del miao.clothes_color
>>> miao.clothes_color#该属性已被删除，所以再次访问会报错

Traceback (most recent call last):
File "<pyshell#12>", line 1, in <module>
    miao.clothes_color
AttributeError: 'Cat' object has no attribute 'clothes_color'
```

其二为 `delattr`：

```
>>> miao.clothes_color='black'
>>> delattr(miao,'clothes_color')
>>> print(miao.clothes_color)#该属性已被删除，所以再次访问会报错
Traceback (most recent call last):
File "<pyshell#22>", line 1, in <module>
    print(miao.clothes_color)
AttributeError: 'Cat' object has no attribute 'clothes_color'
```

以上，便是属性的增删改查的相关方法。

其实方法很多，并不局限于以上，比如修改属性可以提供在类中定义方法来实现：

```
class Cat():
    def __init__(self,color,name):
        self.color=color
        self.name=name
    def rename(self,new_name):
        self.name=new_name

miao=Cat('white','Mary')
```

我们定义了 `rename` 方法，专门用于给小猫改名。现在来改一下吧，新名字叫 `Sara`：

```
>>> miao.rename('Sara')
```

```
>>> print(miao.name)
Sara
```

好了，就讲这么多，接下来学习另一组概念。

私有属性和私有方法

在之前的栗子中，你见到的其实都是公有属性和公有方法。你可能会疑惑：在 Python 的属性中，何谓公有与私有呢？

所谓“私有”，指的是在类中定义的属性或方法不允许被实例化出来的对象调用，只能在类的内部被调用，而“公有”则没有这个限制。

在 Python 的类中，在属性或方法前面添加双下划线 `__`，就得到了私有属性或私有方法。

我们仍搬出上面的栗子，只不过现在不希望实例化出来的小猫可以直接访问到自己的名字，所以将 `name` 属性和 `rename` 方法都设为私有类型：

```
class Cat():
    def __init__(self,color,name):
        self.color=color
        self.__name=name
    def __rename(self,new_name):
        self.__name=new_name

miao=Cat('white','Mary')
```

此时，所有的小猫的初始化的名字都是 `Mary`

你可以尝试访问 `miao` 的名字，看看会不会成功：

round 1:

```
>>> miao.name
Traceback (most recent call last):
File "<pyshell#36>", line 1, in <module>
    miao.name
AttributeError: 'Cat' object has no attribute 'name'

>>> miao.__name
Traceback (most recent call last):
File "<pyshell#37>", line 1, in <module>
    miao.__name
AttributeError: 'Cat' object has no attribute '__name'
```

round 2 :

```
>>> miao.rename('Kite')
Traceback (most recent call last):
  File "<pyshell#39>", line 1, in <module>
    miao.rename('Kite')
AttributeError: 'Cat' object has no attribute 'rename'

>>> miao.__rename('Kite')
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    miao.__rename('Kite')
AttributeError: 'Cat' object has no attribute '__rename'
```

两个回合下来，接连失败。

“私有”的作用在上面的栗子中已经体现出来了。

那为什么搞这么一个东西出来呢？

私有类型的设定，使得类更加稳定，更加安全。我们可以将一些不允许访问或更改的属性和方法设为私有类型，从而避免了由于使用该类时的失误而导致类被更改（正常情况下，我们使用别人定义好的类时，都是去调用一些公有的属性或方法，它们的变量名都是没有双下划线前缀的。而如果类中不希望别人更改的属性或方法没有加双下划线前缀，可能会出现属性会被重新赋值，方法被改写等等问题）。

当然，事无绝对，万一有人真的恶意去改写类，即使你的类中不希望被改动的属性或方法已经设为私有类型了，还是有方法被访问到。所以说，[这种“私有”并不是真正意义上的私有，只是约定俗成的共识](#)。

什么，你很好奇实例化的对象怎么样去访问甚至修改这些私有属性以及调用私有方法？

其实很简单，继续看咯

还是用上面的栗子，代码如下：

```
class Cat():
    def __init__(self,color,name):
        self.color=color
        self.__name=name
    def __rename(self,new_name):
        self.__name=new_name

miao=Cat('white','Mary')
```

无论是访问私有属性还是调用私有方法，都是一个模板：

实例名._类名__属性名或方法名

1. 访问并修改私有属性 `__name`

#访问私有属性

```
>>> miao._Cat__name  
'Mary'
```

#修改私有属性

```
>>> miao._Cat__name='Kite'  
>>> miao._Cat__name  
'Kite'
```

2. 调用私有方法 `__rename`

```
>>> miao._Cat__rename('Laora')  
>>> miao._Cat__name  
'Laora'
```

3. 修改私有方法 `__rename`

```
def trick(some_str):  
    print('我已经成功修改了这个方法，你的调用已经失效，哈哈哈哈')  
>>> miao._Cat__rename=trick  
>>> miao._Cat__rename('Baby')  
我已经成功修改了这个方法，你的调用已经失效，哈哈哈哈
```

以上便是关于私有属性和方法的介绍。

再接下来又会出现另外一组概念，让我们开始吧

类属性和对象属性

在 `__init__` 方法中定义的属性，以及实例化对象自己添加的属性，都是对象属性。

而定义在类之内，方法之外的属性叫做类属性。

每个实例化对象都具有类属性，所以可以把每个通过这个类实例化出来的对象都有的属性定义为类属性。

举个傻瓜似的栗子，比如，每只猫都有2只尖尖的耳朵，即每只猫耳朵的个数都是2，所以可以将耳朵个数这一属性定义为类属性：

```
class Cat():
    ear_num=2#类属性

    def __init__(self,color,name):
        self.color=color
        self.name=name

miao=Cat('white','Mary')
```

实例对象可以访问类属性：

```
>>> miao.ear_num
2
```

但不能修改类属性的值：

```
>>> miao.ear_num=3
>>> miao.ear_num
3
```

可是看完上面的代码，你发现类属性好像可以被修改哎orz~

不是的！

其实，上面的代码修改的是实例化对象的 `ear_num` 属性，类属性 `ear_num` 并没有变，我们可以看下：

```
>>> Cat.ear_num
2
```

看吧，仍然是 2。

如果还是想进一步验证，可以使用 `id()` 来查看内存地址。

```
class Cat():
    ear_num=2

    def __init__(self,color,name):
        self.color=color
        self.name=name
```

```
miao=Cat('white','Mary')
```

```
print('Before:')  
print(id(Cat.ear_num))  
print(id(miao.ear_num))
```

```
miao.ear_num=3  
print('After:')  
print(id(Cat.ear_num))  
print(id(miao.ear_num))
```

输出如下：

Before:

140703953747040

140703953747040

After:

140703953747040

140703953747072

看，在执行 `miao.ear_num=3` 之前，`miao` 这个实例化对象本身并没有 `ear_num` 属性，于是第一次引用该属性时，会去 `Cat` 类的类属性中查询该属性的值，所以该实例化对象的 `ear_num` 属性就是引用的类属性的值，因此两者指向了同一块内存地址 `140703953747040`。

而执行 `miao.ear_num=3` 之后，`miao` 这个实例化对象已经拥有了 `ear_num` 属性，就不会再去类中寻找相应的类属性。此时，类属性仍旧指向原来内存地址 `140703953747040`，而 `miao` 的 `ear_num` 指向了新的内存地址 `140703953747072`，也就是说，`miao` 的 `ear_num` 属性的值被存储在了 `140703953747072` 这块内存空间中。

解释这么多，简单来讲就是一句话：类对象和实例对象都可以访问类属性，但只有类对象才可以修改类属性。

Day 18 封装、继承与多态，面向对象的三大特性



今天你将学到以下内容：

1. 封装
2. 继承
3. 多态

”

开篇

封装、继承和多态是面向对象编程的三大特性，本期将逐一介绍这三大特性。

封装

什么是封装？

所谓封装，就是将类内部的一些属性和方法对外隐藏起来，只提供一些入口（方法）供外部调用。



看上图，整个类被封装在了黑匣子中，只暴露出来4个入口。

那为什么要封装呢？又如何封装呢？

请往下看：

第一，封装之后，可以将内部实现细节隐藏起来

在现实生活中，许多地方都用到了这一思想。比如你所使用的手机，并不要求你理解其内部的实现原理，而只需动动手指点/划几下屏幕就可以操控手机了。这里，手机将其内部的实现封装了起来，手机屏幕就可以理解为手机提供给外部（即用户）调用的入口。

我们来尝试模拟一个简单的手机类 `Mobile`

```
class Mobile():
    def __init__(self, screen_size, kernal_num, system, current_battery):
        self.screen_size = screen_size
        self.kernal_num = kernal_num
        self.system = system
        self.current_battery = current_battery

    # 更新当前电量，每调用1次，电量就减去1%
    def update_current_battery(self):
```

```
if self.current_battery>=1:  
    self.current_battery=self.current_battery-1  
else:  
    print('电量不足, 请充电')
```

在上面的类中，我们定义了屏幕尺寸、核心数，系统以及当前电量这4个手机的属性；定义了更新当前电量的方法（用于模拟手机电量的消耗过程）

这样，我们就把 `Mobile` 这个类封装好了，并且只对外提供了 `update_current_battery` 方法。

这是最简单的一种封装。

第二，将一些私有属性或私有方法封装之后，可以有效防止用户不小心改动了类的内部实现

根据约定俗成的共识，这些私有类型只允许在类的内部被调用，而实例化的对象无法访问，这样可以保证私有属性或方法不会被修改；类只向外部提供几个入口，以便调用类中允许被外部调用的方法，这样就提升了类的稳定性。

我们在上面的栗子基础上稍加修改：

```
class Mobile():  
    def __init__(self,screen_size,kernal_num,system,current_battery):  
        self.__screen_size=screen_size  
        self.__kernal_num=kernal_num  
        self.__system=system  
        self.__current_battery=current_battery  
  
    #更新当前电量，每调用1次，电量就减去1%  
    def update_current_battery(self):  
        if self.__current_battery>=1:  
            self.__current_battery=self.__current_battery-1  
        else:  
            print('电量不足, 请充电')
```

我们只是将该类的4个属性设置为了私有属性

这样，使用 `实例名.属性名` 就不能再访问到对应属性了（约定俗成，忘记的话可以回看上一期哦），从而保证了屏幕尺寸，核心数等硬指标不会被随意访问甚至篡改。

那如何才能访问到这些属性呢？

我们可以在 `Mobile` 类中定义某种方法，用于访问这些私有属性。

在类中添加如下方法：

```
def get_screen_size(self):  
    return self.__screen_size
```

这样，外部就可以访问到 `screen_size` 这一属性了，但不能随意修改：

```
>>> oppo=Mobile(5.2,4,'Android',99)  
>>> oppo.get_screen_size()  
5.2
```

以上，介绍了封装的作用以及如何封装。

继承

先看一个栗子：

```
class Creature():  
    def __init__(self,kind):  
        self.kind=kind  
  
    def grow(self):  
        print('又长高了一点点')
```

上面定义了生物 `Creature` 这么一个类，属性 `kind` 代表生物的种类，方法 `grow` 指的是生物的生长这一动作。

现在，我想定义 `Human` 这么一个类，该类也是具有一个 `kind` 属性，一个 `grow` 方法。可以这样写：

```
class Human():  
    def __init__(self,kind):  
        self.kind=kind  
  
    def grow(self):  
        print('又长高了一点点')
```

仔细观察，发现 `Human` 和 `Creature` 这两个类除了类名不一样外，其余都是一样的。

实际上，人类也属于生物类，所以，能不能通过某种手段来将这两个类联系起来，达到减少冗余代码的效果呢？

于是，[继承](#)出现了！

所谓继承，一般指子类可以继承父类的属性和方法。

在这里，`Human`是子类，`Creature`是父类，原因在于人类是属于生物类的。

所以，当有了`Creature`类之后，我们再去构造`Human`类时，可以直接从`Creature`那里继承：

```
class Human(Creature):
    def __init__(self, kind):
        super().__init__(kind)
```

在子类之后的括号中写入父类的名字，表示子类是继承自父类的

使用`super().__init__(属性名)`来继承父类的属性

此时，子类`Human`便拥有了父类的`kind`属性和`grow`方法：

```
>>> z=Human('huamn')
>>> z.kind
'huamn'
>>> z.grow()
又长高了一点点
```

【注】在使用`super().__init__(属性名)`时，若父类的属性不止一个，需要在`__init__`函数中同时将父类的全部属性名传进来。比如，假设`Creature`还有一个属性叫做`attr1`，那么需要将代码写成下面的形式：

```
class Human(Creature):
    def __init__(self, kind, attr1):
        super().__init__(kind, attr1)
```

上面的栗子只是其中一种情况，还有其他情况，现总结如下：

****情况1：****如果子类仅仅继承父类，但没有重写`__init__`方法，则会自动继承父类的全部属性：

```
class Creature():
    def __init__(self,kind):
        self.kind=kind
    def grow(self):
        print('又长高了一点点')
class Human(Creature):
    pass
```

实例化：

```
>>> z=Human('huamn')
>>> >>> z.kind
'huamn'
>>> z.grow()
又长高了一点点
```

情况2：如果子类继承了父类，并重写了 `__init__` 方法，但是没有使用 `super()`，那么子类将不会继承父类的属性：

```
class Creature():
    def __init__(self,kind):
        self.kind=kind
    def grow(self):
        print('又长高了一点点')
class Human(Creature):
    def __init__(self,name,age):
        self.name=name
        self.age=age
```

实例化：

```
>>> z=Human('Bob',13)
>>> z.name
'Bob'
>>> z.age
13
>>> z.grow()
又长高了一点点
>>> z.kind#没有去继承父类的kind属性，所以报错
>>> Traceback (most recent call last):
File "<pyshell#57>", line 1, in <module>
```

```
z.kind
```

```
AttributeError: 'Human' object has no attribute 'kind'
```

情况3：如果子类继承了父类，重写了 `__init__` 方法，并使用了 `super()`，则子类会继承父类的全部属性

本部分一开始的栗子就属于这种情况，为了总结的完整性，我再把它搬过来吧：

```
class Human():
    def __init__(self,kind):
        self.kind=kind
    def grow(self):
        print('又长高了一点点')

class Human(Creature):
    def __init__(self,kind):
        super().__init__(kind)
```

实例化：

```
>>> z=Human('human')
>>> z.kind
'human'
>>> z.grow()
又长高了一点点
```

总结完毕。

子类在继承父类之后，便拥有了父类的属性（情况3）和方法。但请注意，**子类不能继承父类的私有属性和私有方法**，我们可以来验证一下：

```
class Creature():
    def __init__(self,kind,attr1):
        self.kind=kind
        self.__attr1=attr1
    def grow(self):
        print('又长高了一点点')
    def __private_method(self):
        print('我是私有方法')

class Human(Creature):
    pass
```

实例化：

```
>>> z=Human('human','sth')
>>> z.kind
'human'
>>> z.grow()
又长高了一点点

>>> z.attr1#不能访问私有属性
Traceback (most recent call last):
  File "<pyshell#91>", line 1, in <module>
    z.attr1
AttributeError: 'Human' object has no attribute 'attr1'

>>> z.private_method()#不能调用私有方法
Traceback (most recent call last):
  File "<pyshell#92>", line 1, in <module>
    z.private_method
AttributeError: 'Human' object has no attribute 'private_method'
```

当然，和上一期中所讲的私有类型那部分一样，这只是约定俗成的共识，如果子类的实例化对象非要访问父类的 `attr1` 属性、调用父类的 `private_method` 方法，也不是不可以。根据上一期的知识，我们可以这样做（但是强烈不建议）：

```
>>> z._Creature__attr1
'sth'
>>> z._Creature__private_method()
我是私有方法
```

以上便是关于继承的基本内容，最后要提一点无关紧要的东西：

在一开始定义一个类时，使用的是 `class A()`，而后面定义 `A` 的子类 `B` 时，要写成 `class B(A)`

发现了没？`A` 的括号里面什么都没写

事实上，在 `Python3` 中，每当定义一个类时，都会默认继承自 `object` 类

所以，`class A()` 等价于 `class A(object)`。

而如果在括号中写入自己指定的类，那么就继承自这个指定的类了，就像 `class B(A)` 一样，`B` 将不再继承自默认的 `object` 类，而是继承自 `A`。

多态是基于继承的。通过子类重新写父类的方法，可以得到不同的结果，以提高代码的灵活性，这便是多态。

多态可以使得代码变得容易维护，减少冗余代码。

让我们用一个栗子来体会一下多态：

南极小学六年级一共有三位老师，分别是语文老师、数学老师以及英语老师，请用类简单模拟一下这三位老师上课。

如果不使用多态，我们可以这样写：

```
class Chinese():
    def having_chinese(self):
        print('语文老师正在上课')

class Mathmetics():
    def having_mathmetics(self):
        print('数学老师正在上课')

class English():
    def having_english(self):
        print('英语老师正在上课')

class Course():
    #kind是老师所教课程的种类：[语,数,外]
    def __init__(self,kind):
        self.kind=kind
    def chinese_class(self):
        self.kind.having_chinese()
    def mathmetics_class(self):
        self.kind.having_mathmetics()
    def english_class(self):
        self.kind.having_english()
```

实例化：

```
>>> c=Chinese()#所教课程的种类
>>> t=Course(c)
>>> t.chinese_class()#模拟上课
语文老师正在上课
```

如果现在又来了一个科学老师，那么需要在定义科学这门课的类 `Science` 的同时，在 `Course` 类中添加 `science_class` 方法，代码修改如下：

```

class Chinese():
    def having_chinese(self):
        print('语文老师正在上课')

class Mathmetics():
    def having_mathmetics(self):
        print('数学老师正在上课')

class English():
    def having_english(self):
        print('英语老师正在上课')

#new
class Science():
    def having_science(self):
        print('科学老师正在上课')


class Course():
    def __init__(self,kind):
        self.kind=kind
    def chinese_class(self):
        self.kind.having_chinese()
    def mathmetics_class(self):
        self.kind.having_mathmetics()
    def english_class(self):
        self.kind.having_english()

#new
def science_class(self):
    self.kind.having_science()

```

以后，每来一个教新的课程的老师，就要重复上述动作，你的 `Course` 类中的代码会越来越长

仔细观察 `Course` 类，发现除了 `__init__` 之外，其余的方法都调用了 `kind` 属性，如果我们可以将这些方法合并成一个通用的方法，它可以根据 `kind` 自动确定是在上哪一门课，就可以保证 `Course` 类的代码不变了。

我们的目标代码是这样的：

```

class Course():
    def __init__(self,kind):
        self.kind=kind
    def have_class(self):
        self.kind.having_class()

```

为了实现这一目的，我们可以先定义一个 `Teacher` 类，之后新来的老师都会继承自 `Teacher` 类：

```
class Teacher():
    def having_class(self):
        print('老师正在上课')
```

然后利用多态的特性，让每个老师都根据自己所教科目重写父类的 `having_class` 方法：

```
class Chinese(Teacher):
    def having_class(self):
        print('语文老师正在上课')
class Mathmetics(Teacher):
    def having_class(self):
        print('数学老师正在上课')
class English(Teacher):
    def having_class(self):
        print('英语老师正在上课')
```

这样便通过多态的特性优化了之前的程序，优化后的完整代码如下：

```
class Teacher():
    def having_class(self):
        print('老师正在上课')

class Chinese(Teacher):
    def having_class(self):
        print('语文老师正在上课')
class Mathmetics(Teacher):
    def having_class(self):
        print('数学老师正在上课')
class English(Teacher):
    def having_class(self):
        print('英语老师正在上课')

class Course():
    def __init__(self, kind):
        self.kind = kind
    def have_class(self):
        self.kind.having_class()
```

实例化：

```
>>> c=Chinese()
>>> t=Course(c)
>>> t.have_class()
语文老师正在上课
```

此时，若新来了一位科学（`Science`）老师，只需构建一个继承自 `Teacher` 的类即可，其余代码无需变动：

```
class Teacher():
    def having_class(self):
        print('老师正在上课')

class Chinese(Teacher):
    def having_class(self):
        print('语文老师正在上课')

class Mathmetics(Teacher):
    def having_class(self):
        print('数学老师正在上课')

class English(Teacher):
    def having_class(self):
        print('英语老师正在上课')

class Science(Teacher):
    def having_class(self):
        print('科学老师正在上课')

class Course():
    def __init__(self,kind):
        self.kind=kind
    def have_class(self):
        self.kind.having_class()
```

实例化：

```
>>> c=Science()
>>> t=Course(c)
>>> t.have_class()
科学老师正在上课
```

以上便是关于多态的基本使用方法。



今天你将学到以下内容：

1. 模块

2. 包

”

开篇

开门上代码：

```
import random  
import pandas as pd  
from pandas import *  
import matplotlib.pyplot as plt  
from matplotlib import pyplot as plt  
...  
  
emm...
```

你也许已经知道，这些语句为了导入别人写好的代码文件，以供自己调用

你可能也听说过“包”和“模块”这些概念

没有也不要紧～

今天，我们就来简要介绍下 `Python` 中的包和模块，以及各种导入和调用 [包和模块](#) 的方法。

(以下操作均使用 `IDLE`)

模块

你所写的每一个 `.py` 文件，其实都是一个单独的模块

现在，将以下代码复制到你的 `.py` 文件中，起名为 `demo`，你就拥有了一个叫做 `demo.py` 的模块

```
m='南极冷不冷了'
```

```
def add(a,b):
    print(a+b)

class BlackDog():
    color='black'

    def __init__(self,name):
        self.name=name

    def run(self):
        print('正在跑...')
```

接下来我们将使用该模块演示各种 `import` 的使用大法：

现在，请你新建一个 `test.py` 文件，为了便于演示，请和上面的 `demo.py` 文件放到同一路经下

- `import xxx`

在 `test.py` 中输入

```
import demo
```

运行之后，不会报错，说明已经成功导入

我们可以在 `IDLE` 的命令行输入 `demo` 来查看：

```
>>> demo
<module 'demo' from 'C:\\\\Users\\\\fanxi\\\\Desktop\\\\demo.py'>
```

这样，我们就导入了整个 `demo` 模块

你可以使用 `.` 操作来选取你所需要的对象（变量，函数，类）：

```
#调用demo模块中的变量
>>> demo.m
'南极冷不冷了'

#查看demo模块中的函数
>>> demo.add
<function add at 0x00000299E512F318>

#调用demo模块中的函数
```

```
>>> demo.add(1,2)
```

```
3
```

#调用demo模块中的类，生成实例化对象mydog

```
>>> mydog=demo.BlackDog('旺财')
```

#实例化出来的对象

#可以正常调用其属性和方法

```
>>> mydog.color
```

```
'black'
```

```
>>> mydog.name
```

```
'旺财'
```

```
>>> mydog.run()
```

```
正在跑...
```

这里，我们的模块名是 `demo`，只有四个字母。而有时候一个模块的名字是很长的，本着能偷懒就偷懒的原则，我们不愿意每次都写那么长的模块名，于是有了下面的做法：

```
import demo as d
```

这句代码的作用是导入了 `demo` 模块，并给整个模块重新起了个名字，叫做 `d`

之后，你想调用 `demo` 模块中的某个对象，比如 `add` 函数时，就可以写 `d.add()`。

```
>>> import demo as d
```

```
>>> d.add(1,2)
```

```
3
```

- `from xxx import yyy`

上面的这种方法，是通过 `模块名.xxx` 来调用模块中的函数、变量等，如果你不想使用 `.`，而是直接使用特定的对象(函数,变量,类等)，那么你可以使用下面的两种方法：

现在我只需要用到 `demo` 模块中的 `add` 函数，那么完全没有必要导入整个模块，可以这样写：

```
from demo import add
```

这样，就可以直接使用 `add` 函数了：

```
>>> add(1,2)
```

此时，若想使用 `demo` 模块中的 `m` 变量，由于还没有导入 `m`，因此会报错：

```
>>> m
Traceback (most recent call last):
File "<pyshell#9>", line 1, in <module>
m
NameError: name 'm' is not defined
```

以上是从模块中导入特定对象的方法

如果想要一次导入模块中的全部对象，可以这样写：

```
from demo import *
```

此时，你可以随意使用 `demo` 模块中的全部对象：

```
>>> m
'南极冷不冷了'
>>> add(1,3)
4
>>> mydog2=BlackDog('汪汪')
>>> mydog2.run()
正在跑...
```

当然，你也可以给调用的对象重新起名：

```
>>> from demo import add as a
>>> a(1,2)
3
```

包

你已经了解了关于模块的调用方法，再来学习包的概念也就不难了

在 `Python` 中，包是包含了若干模块的集合，也就是说，若干模块可以组成一个包。

举个例子，假设你已经写好了许多个模块文件，并且将它们放在了一个叫做 `my package` 的文件夹里面，这就差不多是一个包了。

之所以说差不多，是因为在 `Python` 的包中，还包含了一个特殊的模块文件，叫做 `__init__.py`，后面将介绍它的用途。

关于如何导入包以及包里的模块等，和模块中的方法基本上是一样的，你只需要在前面再加上 `包名.` 就好了。

这就像剥洋葱一样，从最外层(包名)开始，一层一层深入(模块名，对象名)

来举个例子吧：

我将 `demo` 和 `zz` 模块存在 `mypackage` 文件夹下



然后在与 `mypackage` 同一路径下新建 `.py` 文件，输入以下代码进行测试：

```
#导入整个包
import mypackage

#导入整个包并起个别名叫做pck
import mypackage as pck

#从包中只导入demo模块并起个别名叫做d
from mypackage import demo as d

#导入demo模块中的全部对象
from mypackage.demo import *
```

但是当你运行

```
from mypackage import *
```

之后，会惊讶的发现，并没有将 `demo` 和 `zz` 模块导入：

```
>>> demo
Traceback (most recent call last):
File "<pyshell#0>", line 1, in <module>
    demo
```

```
NameError: name 'demo' is not defined
>>> zz
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    zz
NameError: name 'zz' is not defined
```

这是因为你的包中没有 `__init__.py` 模块

之前说过，在 `Python` 的包中，包含了一个特殊的模块文件，叫做 `__init__.py`。

它的用途在于：

规定在执行 `from 包名 import *` 时，查找可以被导入的模块名。也就是说，只有你的模块名被写入 `__init__.py` 中，才能被 `from 包名 import *` 这句代码导入，否则需改换其他导入语句。

所以，以上的 `mypackage` 并不是真正意义上的包。

这一点了解即可，大多数时候，我们都是在调用别人写好的包，因此掌握以上关于包和模块的导入以及调用方法就达到目的了。

Day 20 动手打造属于自己的流水账记录器，实践是检验真理的唯一标准



今天你将学到以下内容：

1. 打造流水账记录器

”

开篇

本期将动手制作属于自己的流水账管理工具，先看一下最终成品：



操作提示：输入数字1进行添加，输入数字2进行查询，输入字母q退出程序

只需双击写好的 `.py` 文件，即可进入以上界面，是不是有一点点酷呢~

好了，现在开始吧~

前置知识

- eval

栗子1

```
>>> x='1+1'  
>>> type(x)  
<class 'str'>  
>>> eval(x)  
2
```

栗子2

```
>>> d={"name": "zz"}  
>>> type(d)  
<class 'str'>  
>>> dd=eval(d)  
>>> type(dd)  
<class 'dict'>
```

不难看出，`eval()` 可以将字符串转为 Python 的数据类型，只要该字符串满足 Python 数据类型的基本格式。

- 文件操作

使用 `open()` 函数打开文件；

使用 `read()` 或者 `readlines()` 来一次性读取全部文件(`read`)或者逐行读取文件得到一个列表，列表中的每个元素都是文件的一行内容(`readlines`)；

使用 `write()` 向文件中写入内容。

具体用法请在接下来的内容中体会。

思路分析

我们主要实现添加和显示清单的功能，以及如何将数据存储到本地。

主要需要完成的工作如下：

(1) 分析所需文件

(2) 获取当前时间

(3) 定义一个流水账类

(3) 实现"添加并保存至本地"的功能

(4) 实现"展示全部记录"的功能

(5) 实现"用户调用的接口"

分析所需文件

本程序中，将总收入、总支出以及总记录数存入 `info.txt` 进行保存，将全部的记录存入 `all_item.txt` 进行保存。

为了避免错误，请在与你的 `.py` 文件同级目录下手动创建以上两个文件。

获取当前时间

每一条新的记录都需要显示被创建的时间，因此需要想办法获取当前时间。

```
import datetime

def get_current_time():
    time_stamp = datetime.datetime.now()
    return time_stamp.strftime('%Y.%m.%d-%H:%M:%S')
```

借助 `datetime` 模块，我们可以得到当前时间，运行上述函数，就可以得到当前时间了，比如我写这篇文章的时间是 `2020.09.08-20:11:45`。

定义一个流水账类RunningAccount

```
class RunningAccount():

    def __init__(self):
        xx=open('info.txt', 'r')
        content=xx.read()
        if content=='':
```

```
print('数据仓库为空! ')
self.all_income,self.all_outcome,self.cnt=0,0,0
else:
    self.all_income,self.all_outcome,self.cnt=eval(content)
xx.close()
```

上面定义了 RunningAccount 类，并初始化了 `__init__` 方法：

首先，打开 `info.txt`，并读取其中内容，存入 `content`；

然后分为两种情况：

(1) 如果 `info.txt` 为空，则说明这是第一次运行程序，于是会初始化总收入、总支出以及总记录数都为 0；

(2) 否则 (`info.txt` 不为空)，读取当前的总收入、总支出以及总记录数。

最后别忘记关闭文件。

添加并保存至本地

#num是金额，plus是备注（比如该金额的用途等）

```
def add_and_save(self,num,plus):
    #使用追加模式
    with open('all_item.txt','a+') as file:
        dic={}
        dic[get_current_time()]=[num,plus]
        file.write(str(dic)+'\n')
        print("添加成功!")
    #更新数据条目
    self.cnt=self.cnt+1
    #更新总收入和总支出
    if num<0:
        self.all_outcome+=num
    else:
        self.all_income+=num
    with open('info.txt','w') as f:
        f.write(str([self.all_income,self.all_outcome,self.cnt]))
    print('全局信息更新成功!')
```

首先打开 `all_item.txt` 文件，使用追加模式(`a+`)来在文件末尾添加一条新的记录，以免产生覆盖；

然后定义了一个字典 `dic`，用于暂时保存一条新的记录，键代表时间，值代表金额和备注；

接着将 `dic` 写入 `all_item.txt`，总的记录数加1，并更新总收入和总支出；

最后以写的方式(`w`)打开 `info.txt`，更新总记录数、总收入与总支出这三个数字。

展示全部记录

```
def show(self):
    f=open('info.txt','r')
    all_income,all_outcome,cnt=eval(f.read())
    f.close()

    print('#####')
    print('总收入: ',self.all_income)
    print('总支出: ',self.all_outcome)
    print('净余额: ',self.all_income+self.all_outcome)
    print('#####')
    print('          共检索到%d条记录'%cnt)
    print('#####\n')

    with open(r'all_item.txt','r') as f:
        print('日期\t收入/支出金额\t用途\n')
        for line in f.readlines():
            line=eval(line)
            date,info=list(line.keys())[0],list(line.values())[0]
            number,use=info
            print('{0}\t{1}\t{2}'.format(date,number,use))

    print('#####\n')
```

首先打开 `info.txt` 文件，读取总收入、总支出以及总记录数并打印出来；

然后打开 `all_item.txt`，逐行读取其中内容，并按照 `日期，金额，用途` 的格式打印。

用户调用的接口

以上已经实现了一个流水账记录器所需全部组件，现在将它们组合在一起，就可以完成我们的流水账记录器啦：

```
while True:
    x=RunningAccount()
```

```

ch=input('操作提示：输入数字1进行添加，输入数字2列出清单，输入字母q退出程序\n')

if ch=='q':
    break;
number=int(ch)
if number==1:
    num,use=input('请输入金额以及用途，以空格分隔，比如"-100 购物"\n').split()
    try:
        num=float(num)
        x.add_and_save(num,use)
    except:
        print('数字输入有误，操作取消...')
elif number==2:
    x.show()

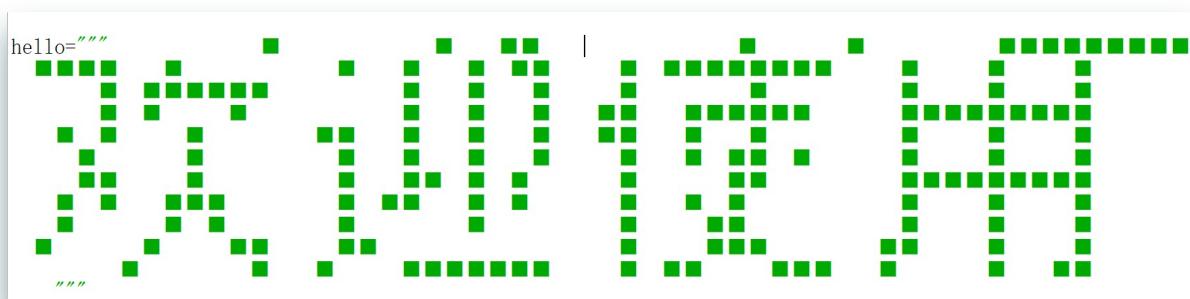
```

这样，只需双击本文件，就可以在弹出的窗口中进行操作啦



等等，是不是少了些什么？

哦，是每次进入程序时大大的 `欢迎`。这个很简单，直接找一个字符文字生成器生成，然后拷贝到上面的 `while` 死循环最开始位置就好了，就像这样：



到这里，全部代码已经编写完成，最终的结构是这样的：

名称	修改日期	类型
> 流水账Python脚本		

	all_item.txt	2020/9/8 20:51
	info.txt	2020/9/8 20:51
	run.py	2020/9/8 20:11

最后列出完整的代码：

```
##### 获取当前时间#####
import datetime

def get_current_time():
    time_stamp = datetime.datetime.now()
    return time_stamp.strftime('%Y.%m.%d-%H:%M:%S')

##### 功能实现部分#####

class RunningAccount():

    def __init__(self):
        xx=open('info.txt','r')
        content=xx.read()
        if content=='':
            print('数据仓库为空！')
            self.all_income,self.all_outcome,self.cnt=0,0,0
        else:
            self.all_income,self.all_outcome,self.cnt=eval(content)
        xx.close()
        #print(self.all_income,self.all_outcome,self.cnt)

    #num是金额， plus是备注（比如该金额的用途等）

    def add_and_save(self,num,plus):
        #使用追加模式
        with open('all_item.txt','a+') as file:
            dic={}
            dic[get_current_time()]=[num,plus]
            file.write(str(dic)+'\n')
            print("添加成功！")

        #更新数据条目
        self.cnt=self.cnt+1
        #更新总收入和总支出

        if num<0:
            self.all_outcome+=num
        else:
            self.all_income+=num
        with open('info.txt','w') as f:
            f.write(str([self.all_income,self.all_outcome,self.cnt]))
        print('全局信息更新成功！')
```

```
def show(self):  
    f=open('info.txt','r')  
    all_income,all_outcome,cnt=eval(f.read())  
    f.close()  
  
    print('#####')  
    print('总收入: ',self.all_income)  
    print('总支出: ',self.all_outcome)  
    print('净余额: ',self.all_income+self.all_outcome)  
    print('#####')  
    print('          共检索到%d条记录'%cnt)  
    print('#####\n')  
    with open(r'all_item.txt','r') as f:  
        print('日期\t收入/支出金额\t用途\n')  
        for line in f.readlines():  
            line=eval(line)  
            date,info=list(line.keys())[0],list(line.values())[0]  
            number,use=info  
            #print(date,' ',number,' ',use)  
            print('{0}\t{1}\t{2}'.format(date,number,use))  
  
    print('#####')
```

#####用户调用接口#####

```
hello=""  
while True:  
    x=RunningAccount()  
    ch=input('操作提示: 输入数字1进行添加, 输入数字2列出清单, 输入字母q退出程序\n')  
    if ch=='q':  
        break;  
    number=int(ch)  
    if number==1:  
        num,use=input('请输入金额以及用途, 以空格分隔, 比如"-100 购物\n').split()
```

```
try:  
    num=float(num)  
    x.add_and_save(num,use)  
except:  
    print('数字输入有误，操作取消...')  
elif number==2:  
    x.show()
```

Day 21 一切都要从搭建环境说起（新的开始！）



今天你将学到以下内容：

- 1. Anaconda介绍
- 2. PyCharm介绍

”

开篇

通过前面20期的学习，你已经掌握了 Python 的基础用法，对于非程序员而言，掌握这些已经足够了。接下来需要做的是，选几个感兴趣的应用方向，比如数据分析、机器学习、深度学习（这些是我自己的目标方向，哈哈）等，去学习相关包的使用方法。

当然，工欲善其事，必先利其器，在之前的教程中，我们总是在使用 IDLE，这对初学者来说够用了，但并不是最佳选择。当你已经基本掌握了 Python 基础知识，并打算应用这些知识为自己服务时，以下的推荐软件不容错过。

Anaconda 与 PyCharm

python 翻译过来是蟒蛇，而 anaconda 翻译过来是巨蟒，显然，后者定有其强大之处。

没错，Anaconda 几乎是数据科学的超强利器，只要安装了 Anaconda，基本上常用的数据分析包(如 numpy, pandas)、机器学习包(如 scikit-learn)以及绘图工具(如 seaborn, matplotlib)都被自动安装上了，省去了手动安装带来的不必要的麻烦。

并不仅于此，Anaconda 还自动安装了 jupyter notebook，这款工具非常适合使用 Python 处理数据的场景，它提供了模块化的代码运行方式，你每次的输入都可以得到一个即时的输出，所写即所得。

只需打开你的电脑终端，输入 `jupyter notebook`，即可自动打开你的默认浏览器，并进入到功能区页面

The screenshot shows a Jupyter Notebook interface. At the top, there's a menu bar with File, Edit, View, Insert, Cell, Kernel, Widgets, Help, and a toolbar with various icons. On the right, it says "可信的" and "Python 3". Below the menu, there are two code cells:

```
In [132]:  
1 #预测未来  
2 def predict_future(model,time):  
3     start=dataset[-3:,:].reshape(dataset[-3:,:].shape[1],dataset[-3:,:].shape[0],1)  
4     print(start)  
5     predictions=[]  
6     for i in range(time):  
7         z=model.predict(start)  
8         pred=list(z)[0][0]  
9         predictions.append(pred)  
10        #pred=scaler.inverse_transform(pred)  
11        print(pred)  
12        a=float(list(start)[0][1][0])  
13        b=float(list(start)[0][2][0])  
14        c=pred  
15        start=[a,b,c]  
16        start=numpy.array(start)  
17        start=start.reshape(start.shape[0],1)  
18        start=start.reshape(start.shape[1],start.shape[0],1)  
19        print(i,start)  
20    return predictions
```

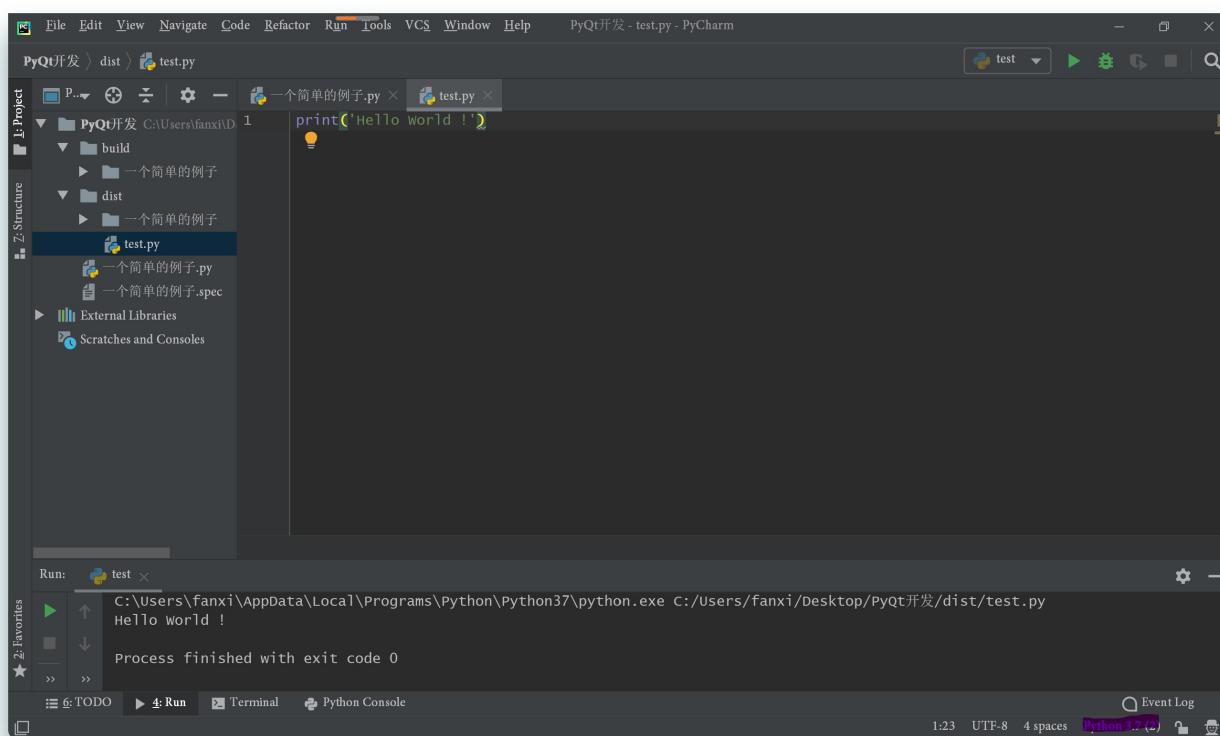
```
In [133]:  
1 pred=predict_future(model,25)  
[[[0.9340415]  
[0.9666894]  
[1.  
 ]]]  
0.95883524  
0 [[[0.96668938]  
[1.  
 ]  
[0.95883524]]]  
0.9642074
```

当然，以上仅是其功能的一小部分，它还可以创建指定 `Python` 版本的虚拟环境，用于隔离不同项目。

这些都是常用的功能，更多功能可以在用到时再进行探索。

相比于 `Anoconda`，`PyCharm` 更适合一些项目的开发。它支持每次创建一个新的项目 (`Project`)

下图展示了一个简单的栗子



(点击右下角的紫色部分可以选择 `Python` 解释器)

对于这两个工具，我个人平时用的更多的是 `Anoconda` 的 `jupyter notebook`，所以这里不对 `PyCharm` 做过多讨论，以免再误导大家（孩怕~）。

总之，看到这里的你已经完全可以走出 `IDLE`，去拥抱以上两款软件啦！

最后，好像还没讲怎么安装它们...

其实都是一路无脑化 `next`，大家下载下来按照提示安装即可。

对了，`PyCharm` 分为专业版和社区版，前者功能更强大，但是是付费的。后者也可满足日常需要，而且是免费的，因此我毫不犹豫地选择了社区版（~）。

写在最后

到这里，整个系列就完结了。当我打下这些字时，我正在前往学校的火车上，这也预示着长达近9个月的假期"完结"了。

回顾这九个月，

从一月份开始，断断续续的学习计算机组成原理，计算机网络以及数据库，同时在 `OJ` 网站刷题(以 `C` 语言为主力语言)

这种生活一直持续到5月份的复试结束，虽然只是学得了一点皮毛，但最后结果总算还理想。

之后开始学习深度学习的基础知识，并接触了深度学习框架 `Tensorflow`。说来惭愧，之前还想着趁着这段假期把 `Tensorflow` 学个差不多，结果到现在还是停留在入门的状态，真的是没有考试就没有驱动力了哈哈。

就这样啦，下个系列见~

至此，本书就结束了，但这并不代表Python学习之路到此为止，相反，真正的学习才刚刚开始：Python只是一个工具，接下来，你需要学习如何将其运用到你所在的行业，无论是数据科学、机器学习，还是游戏开发，web爬虫等。

知识是永无止尽的。在鱼和熊掌不可兼得的情况下，只求全力以赴，并一以贯之。

不当之处，欢迎指正！

若本书对您有所帮助，不妨分享出去，这便是对我最大的支持！

