

Table of Contents

前言 5

第一章 Python 基础（一） 8

 0.1.1 运行 Python..... 9

 0.1.2 变量和表达式 13

 0.1.3 条件语句..... 16

 0.1.4 文件输入和输出 18

 0.1.5 字符串..... 21

 0.1.6 列表 27

 0.1.7 元组 30

 0.1.8 集合 35

 0.1.9 字典 38

第二章 综合练习(一) 42

 0.2.1 习题 1： 分别定义四种基本类型的变量 43

 0.2.2 习题 2： 定义一个列表， 一个集合， 一个元组和一个字典..... 45

 0.2.3 习题 3： 求一系列数字的平方 46

 0.2.4 习题 4： 求一系列数字的平方和 49

 0.2.5 习题 5： 列出 20 以内的所有偶数..... 51

 0.2.6 习题 6： 计算 30 以内的质数。 53

 0.2.7 习题 7： 计算平面上两点间的距离..... 55

第三章 Python 基础（二） 57

 0.3.1 迭代与循环..... 58

 0.3.2 函数 64

 0.3.3 生成器..... 67

 0.3.4 对象和类..... 71

| | |
|-------------------------------------|-----|
| 0.3.5 异常 | 78 |
| 0.3.6 模块 | 81 |
| 0.3.7 文档 | 84 |
| 第四章 综合练习（二） | 86 |
| 0.4.1 习题 1：使用命令行参数 | 87 |
| 0.4.2 习题 2：在命令行直接执行 Python 语句 | 90 |
| 0.4.3 习题 3：接收命令行输入 | 91 |
| 0.4.4 习题 4：命令行交互 | 93 |
| 0.4.5 习题 5：访问网页 | 95 |
| 第五章 语法规约 | 97 |
| 0.5.1 代码结构和缩进 | 98 |
| 0.5.2 标识符和保留字 | 101 |
| 0.5.3 数字的字面量 | 103 |
| 0.5.4 字符串的字面值 | 105 |
| 0.5.5 容器 | 108 |
| 0.5.6 运算符、分隔符和特殊符号 | 109 |
| 0.5.7 源代码的编码 | 110 |
| 第六章 函数 | 111 |
| 0.6.1 函数 | 112 |
| 0.6.2 参数传递与返回值 | 115 |
| 0.6.3 作用域 | 118 |
| 第七章 面向对象编程 | 121 |
| 0.7.1 创建类 | 124 |
| 0.7.2 创建实例对象 | 126 |
| 0.7.3 访问类成员 | 127 |

| | |
|------------------------------|-----|
| 0.7.4 类的继承..... | 129 |
| 0.7.5 方法重写..... | 132 |
| 0.7.6 类方法和静态方法..... | 134 |
| 0.7.7 基础重载方法..... | 136 |
| 0.7.8 运算符重载..... | 138 |
| 第八章 模块与包..... | 142 |
| 0.8.1 为什么使用模块与包..... | 143 |
| 0.8.2 import 和 from 的使用..... | 147 |
| 0.8.3 模块搜索路径..... | 148 |
| 0.8.4 打包..... | 150 |
| 第九章 GUI 图形界面..... | 153 |
| 0.9.1 Tkinter..... | 155 |
| 0.9.2 PyQt5..... | 160 |
| 0.9.3 PyGObject..... | 170 |
| 第十章 Web 开发..... | 178 |
| 1.0.1 Flask 概要..... | 179 |
| 1.0.1.1 Flask 安装..... | 179 |
| 1.0.1.2 Flask 的一个简单例子..... | 180 |
| 1.0.2 请求-响应循环..... | 184 |
| 1.0.2.1 程序和请求上下文..... | 184 |
| 1.0.2.2 请求调度..... | 186 |
| 1.0.2.3 请求钩子..... | 186 |
| 1.0.3 模板..... | 189 |
| 1.0.3.1 Jinja2 模板引擎..... | 189 |
| 1.0.3.2 模板变量..... | 190 |

| | |
|----------------------------------|-----|
| 1.0.3.3 模板控制语句..... | 192 |
| 1.0.3.4 使用 Flask-Bootstrap | 195 |
| 1.0.3.5 模板链接..... | 198 |
| 第十一章 综合案例..... | 200 |
| 1.1.1 网页爬虫..... | 201 |
| 1.1.2 抓取天气的爬虫示例..... | 202 |
| 1.1.3 下载学堂在线网站视频示例 | 207 |
| 1.1.4 Python 处理 excel 数据..... | 210 |
| 下载安装 Anaconda..... | 212 |
| 背景 | 213 |
| 安装图文简易教程..... | 214 |

前言

这本书的对象是面向 **Python** 语言的初学者。我们认为初学者需要一本有趣并且有用的入门书，能够从中得到与其他书不同的体验。你可以是从来没有编程经验的新人，也可以是学习过其他语言的读者，想了解一下 **Python** 语言与你学过的其他语言有何不同；或者是学习过 **Python** 语言，但还是想要了解一下这本书里有没有一些特别的地方。那么这本书正好就是你需要的。

关于如何学习一门高级编程语言，根据编写组的老师们之前在教学过程中总结的经验来看，我们认为学习者需要的更多是实践的操作练习。学习者可以先看到一段代码的运行效果，然后再回过头来读懂代码，从一些由浅入深的实例中，逐渐理解该门语言的语法、语句含义等概念。对于 **Python** 这样入门友好的语言来说，从现成的代码中反过来读出它的含义，是完全可能的。因为基本上能够读懂英文，就能够读懂 **Python**。但是实践操作这个环节，是无论如何也不能省略的。

我们认为，要学习编程语言，最好的方法是老老实实在地敲代码。

所以在书中有大量的程序代码，是一些教学中使用的实例，这是本书的核心，也是我们认为本书的价值所在。每一个案例都是经过老师们精心编写、并能够正确运行的，每一个知识点的案例都是相互联系、有递进关系的，后面的实例是在前一个实例的基础之上做的某些改进。跟随着实例程序的一步步深入，你会看到你的代码越来越顺畅，功能越来越强大。

请记住：在使用本书的过程中，对于书中的每一个案例的每一行代码，我们都推荐你逐字逐句的敲在你的本机上，看到它运行起来。一定是要一字不差地录入代码，而不是复制粘贴。如果你只是想要看看这段代码运行起来的效果，复制粘贴是可以的，但那也仅仅是看看效果而已。复制粘贴对于你理解代码的含义没有任何帮助，即使你的程序能够运行起来，你也无法理解其中每一行代码甚至每一个字符的具体含义，甚至你也无法感受到成功的喜悦。

在你熟悉并了解了实例代码的含义之后，建议你对代码段做一些调整或者改进，让它实现更多不同的功能。慢慢地，你也就能够写出属于你自己的代码了。

而且我们相信，一个好的程序员一定会特别关注细节方面。如果你总是碰到语法错误，那么，可以仔细检查一下你的输入法。例如，你的括号是中文格式的吗？你的逗号和引号是全角的吗？对于 **Python** 来说，最重要的是缩进格式。那么你用的是几个空格的缩进？你的缩进的距离对不对？这些细节方面的问题最初都会给学习者带来不小的困扰。当你碰到问题并解决了之后，你会对 **Python** 理解得更深，并会养成良好的编程风格。跟着错误提示来改进你的代码，本身就是一个学习的过程，而且你会比单纯的看书要学习到更多。

本书的代码是基于 **Python3.7** 版本来编写的，我们采用的运行环境是 **Anaconda**。

在此要感谢编写组的各位老师，没有大家齐心协力的努力，这本书不会面世。从有了最初的想法到最终付梓，其间经历了太多次的

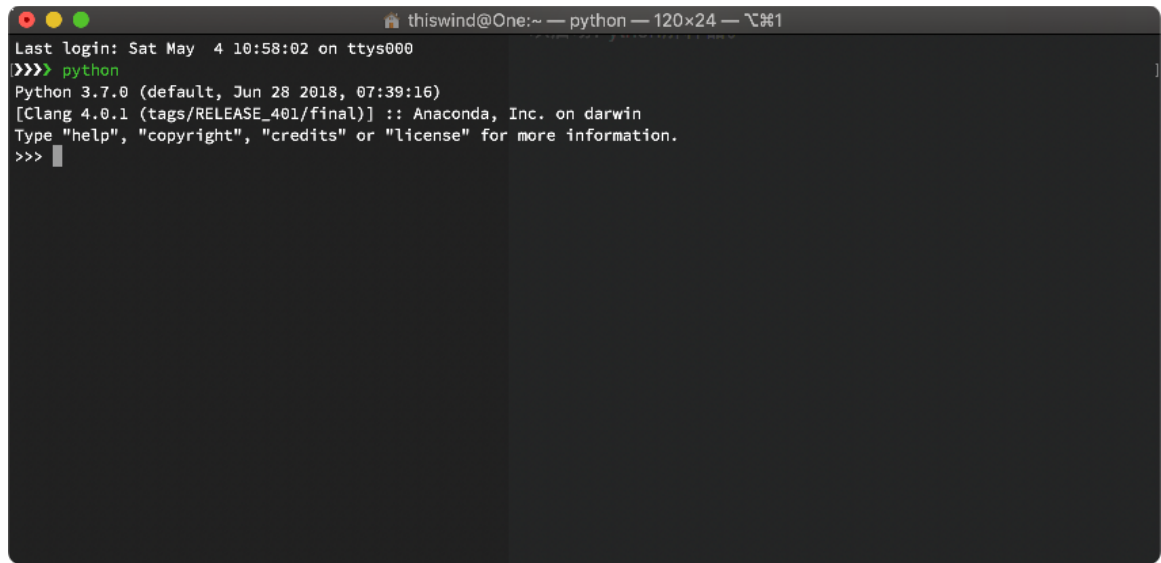
修正和补充完善。也要感谢云南科技出版社的编辑老师们为这本书付出的心血，谢谢。

第一章 Python 基础（一）

这一章我们将快速介绍 Python 语言的常用特性，比如如何运行 python 程序，变量和表达式、条件语句等，让读者对 Python 有一个大概的认识。其中一些内容在后续的章节里还会作为专题进行展开详细介绍。

0.1.1 运行 Python

Python 程序是由解释器执行的，在命令行当中输入 `python` 命令就可以启动 Python 解释器。

A terminal window titled 'thiswind@One:~ — python — 120x24 — ￼1'. The terminal shows the command 'python' being executed, which starts the Python 3.7.0 interpreter. The output includes version information and a prompt 'Type "help", "copyright", "credits" or "license" for more information.' followed by a ' >>>' prompt.

```
thiswind@One:~ — python — 120x24 — ￼1
Last login: Sat May  4 10:58:02 on ttys000
>>> python
Python 3.7.0 (default, Jun 28 2018, 07:39:16)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python 解释器启动之后会出现版本信息以及 `>>>` 提示符，用户可以在提示符后面输入 `python` 语句。

比如：

```
Python 3.7.0 (default, Jun 28 2018, 07:39:16)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello world')
Hello world
>>>
```

注意：本教程的所有例子均使用最新的 Python3.7 版本。如果你使用 python3.6 以下的版本，可能有的内容会报错。

Python 的交互模式是一个非常有用的功能。在交互模式中，可以输入任何合法的语句和语句序列，并且立刻就可以看到执行的结果。很多人甚至直接把 python 的交互环境当作计算器使用。

比如：

```
>>> 1751 + 51^3
1801
>>> _ % 35
16
>>>
```

在交互模式当中的时候，有一个特殊的变量`_`，它保存着最后一次计算的结果。如果要啊在后续继续计算，那么使用这个变量就非常方便。但是要注意，这个变量只在交互模式当中存在。

如果要创建可以重复执行的程序，可以将语句放到一个文件当中：

```
# helloworld.py
print('Hello World')
```

这种保存代码的文件，我们称之为“源文件”，Python 的源文件是一个普通的文本文件，后缀是`.py`。上面例子中的`#`表示这一行是一个注释。注释是用来帮助将来阅读源代码的人理解程序代码的有效工具。

要执行 `helloworld.py` 这个文件，可以通过把文件名作为参数传给 `python` 解释器的方式，用 `python` 解释器进行执行。

例如：

```
$ python helloworld.py
Hello world
$
```

在 Windows 系统中，双击一个.py 文件，或者在 Windows 开始菜单的“运行”命令中输入程序名称，也都可以启动一个命令哈昂，然后用 python 解释器来执行这个程序。但是你要注意，这个程序执行完之后，控制台窗口就会立刻消失，以至于你根本来不及看清执行的结果。所以，如果你要开发比较长的程序，并且还要进行调试，那么建议使用专门的 IDE，比如 Pycharm。

在 UNIX 系统中（比如 Linux、MAC 等），可以在程序的第一行使用#!来指定 python 解释器的位置，例如：

```
#!/usr/bin/env python3
print('Hello World')
```

Python 解释器将不断运行 python 语句，直到达到文件的结尾。如果是以交互模式运行，有两种方法可以退出解释器，一种是直接输入 EOF(end of file，文件结束符)字符，另一种是在 PythonIDE 当中直接终止程序运行。

在 UNIX 系统当中，EOF 是 ctrl+d，在 Windows 系统当中，是 ctrl+z。

此外，也可以直接在程序当中抛出 SystemExit 异常来请求退出程序。

例如，直接在交互模式当中：

```
>>> raise SystemExit  
$
```

0.1.2 变量和表达式

Python 是动态语言，在程序的运行过程当中，变量的名称会根据具体情况，分别绑定到不同的值上面，这些值也可能分别是不同的数据类型。Python 当中的赋值运算符的操作，只是在名称和值之间建立一种关联。

每个值都有自己的类型，比如 str(字符串)、int（整数）等等，但是变量的名称是没有类型的。变量的名称可以引用任何一种数据类型。

与 C 语言不同，在 C 语言当中，变量的名称就已经蕴含了变量类型的信息，包括值的数据类型，占用多少 bit 空间，以及变量在内存当中的相对位置等等。

我们来看一个计算银行利率的例子：

程序 1.2-1 计算银行利率

```
amount = 10000
rate = 0.035

years = 3
year = 1
while year <= years:
    amount = amount * (1 + rate)
    print(f'第 {year} 年，本息合计 {amount} 元')
    year = year + 1
```

此程序的输出如下：

```
第 1 年，本息合计 10350.0 元
第 2 年，本息合计 10712.25 元
第 3 年，本息合计 11087.17875 元
```

一开始我们给 `amount` 赋值为一个整数:

```
amount = 10000 # 存入金额为一万元
```

这条语句把整数 10000 赋值给 `amount` 这个变量。但是在程序当中，它被重新赋值为:

```
amount = amount * (1 + rate)
```

这条语句对表达式进行求值，并把 `amount` 这个名称，重新与计算结果关联起来。`amount` 变量的原始值是一个整数，但是现在它变成了一个浮点数。（因为 `rate` 是一个浮点数，于是 `amount * (1 + rate)` 也被提升为一个浮点数。表面上看是 `amount` 变量的类型从整数（`int`）变成了浮点数（`float`），可是更准确地说，实际上是 `amount` 这个名称所引用的值的类型变了。

在 Python 当中换行表示一个语句的结束。当然，也可以用;（英文的分号）在统一行里分隔多条语句，例如:

```
amount = 10000;rate = 0.035; years = 3
```

接下来我们通过一个循环进行计算:

```
while year <= years:
```

循环语句关键字 **while** 对紧跟着的表达式的值进行检验，如果值为 **True**（真），那么 **while** 的主体就会被执行，之后再一次检验条件，直到条件为 **False**（假）的时候，结束主体的执行。

在 Python 当中用相同的缩进来表示同一个代码块，例如程序 2.1-1 当中：

```
while year <= years:
    amount = amount * (1 + rate)
    print(f'第 {year} 年，本息合计 {amount} 元')
    year = year + 1
```

while 之后的三条语句，他们都缩进了四个空格。在 Python 当中并没有强制规定需要缩进几个空格，只要同一个代码块的缩进保持一致就可以了。一般来说，我们习惯上把缩进设置为 4 个空格。

在每一次循环当中，我们把当前的计算结果通过 **print** 函数输出出来：

```
print(f'第 {year} 年，本息合计 {amount} 元')
```

在字符串前面加一个 **f** 表示这是一个 **f-string**，这是 Python3.6 之后新增加的一种更方便的字符串格式化方式。在这个格式化字符串当中 **{year}** 和 **{amout}** 表示在程序运行的过程当中，直接用 **year**、**amout** 这两个变量的值填入到字符串当中。

0.1.3 条件语句

在 Python 当中，可以用 `if` 和 `else` 语句执行简单的条件检验，例如：

```
if a < b:
    print('a 比较小')
else:
    print('b 比较小')
```

`if` 子句和 `else` 子句的主体代码块是用缩进来表示的。`else` 子句可以不写。

如果要创建一条空的（什么也不执行的）的子句，可以用 `pass` 语句，例如：

```
if a < b:
    pass # 什么也不做，空语句
else:
    print('b 比较小')
```

在判断条件当中，可以使用 `or`、`and` 和 `not` 关键字，组成更加复杂的布尔表达式，例如：

```
if grade == '三年级' and class_name == '1 班' \
    and gender == '女':
    print('请到大操场集合')
```

注意，由于在 Python 当中，换行是表示一个语句的结束，如果代码很长希望分成多行来写，那么可以在行的结尾加一个反斜杠\。在下一行里面将不检查缩进的数量，所以可以根据你的喜好缩进任何长度。

不同于 Java、C 这样的语言，在 Python 当中没有 switch 语句，如果要进行多个判断检测，可以在判断语句中用 elif 子句，例如：

```
if age < 3:
    print('婴儿')
elif age < 6:
    print('幼儿')
elif age < 12:
    print('小学生')
elif age < 15:
    print('初中生')
elif age < 18:
    print('高中生')
else:
    print('成年人')
```

所有的关系运算符，例如>、<等等，都返回 True 或者 False。

关于判断语句的在后面的章节里还会进一步介绍。

0.1.4 文件输入和输出

下面的程序可以打开一个文件，然后逐行读取文件的内容并显示出来：

```
f = open('abc.txt')    # 打开文件名为'abc.txt'的文件
one_line = f.readline() # 读取第一行

while one_line:
    print(one_line, end="") # 输出这一行的内容
    one_line = f.readline() # 重新读取一行

f.close()              # 关闭文件
```

`open()`函数返回一个文件对象，通过这个文件对象，可以对文件进行各种操作。`open()`函数需要接受一个参数作为文件名，比如上面例子中的 `abc.txt`。`readline()`方法读取文件当中的一行，包括行尾的换行符，每次调用的时候，会依次读取下一行。直到一直读取到文件末尾的时候，会返回一个空字符。

由于 `print()`函数默认会给输出的一行添加一个换行符，为了避免换行符重复，所以我们需要通过 `end=""`手动设置行尾的字符，把默认的换行符改为空字符”。

在这个例子当中，程序会读取了 `abc.txt` 文件当中的所有行。对于像这样依次对数据集上进行循环操作，通常称为“迭代”。

迭代是一种很常用的操作，所以 `Python` 当中提供了一个 `for` 语句，专门用来对内容的每一项进行迭代。例如，上面的例子当中的循环，也可以写成下面这种形式：

```
for one_line in open('scratch.py'):
    print(one_line, end='') # 输出这一行的内容
```

如果要将输出写入到文件当中，需要在 `print()` 函数当中把输出重定向到文件当中，例如：

```
f = open('abc.txt', 'w') # 打开文件
print('hello', file=f) # 把输出重定向到文件当中
```

通过 `w` 参数，指定以可写的方式打开名为 `'abc.txt'` 的文件，如果这个文件不存在，那么就会自动创建一个。接下来在 `print()` 函数当中，通过 `file=f` 指定把输出的内容重定向到这个文件当中。

此外，我们也可以用文件对象的 `write()` 方法向文件当中写入数据，一般来说，我们更推荐这种方法，例如：

```
f = open('abc.txt', 'w')
f.write('hello2')
```

上述例子处理的都是文件，但是同样的方法，也可以用来处理 Python 解释器的输出流和输入流。比如，如果我们要从标准输入读取用户的输入信息，可以直接从 `sys.stdin` 文件当中读取。

```
import sys

f = sys.stdin
print(f'用户输入：{f.readline()}')
```

首先需要导入 Python 标准库当中 `sys` 模块。

程序在执行的时候会停下来等待用户输入内容，在用户输入完成按下回车键之后，后面的程序才会继续执行，如下：

```
aaa  
用户输入：aaa
```

如果要把数据输出到屏幕上，可以直接向 `sys.stdout` 文件写入内容。例如：

```
import sys  
  
f = sys.stdout  
f.write('hello\n')
```

程序的输出结果如下：

```
hello
```

0.1.5 字符串

创建字符串，需要将字符串的字面值放到英文的引号当中，可以用单引号'、双引号"、三个单引号'''或三个双引号""""，例如

```
a = 'hello'
b = "hello"
c = '''hello'''
d = """hello"""

print(f'a is {a}')
print(f'b is {b}')
print(f'c is {c}')
print(f'd is {d}')
```

程序的输出为：

```
a is hello
b is hello
c is hello
d is hello
```

这里需要注意的时候，两端的引号必须是匹配的。如果字面内容里包含单引号，那么你可以采用一对双引号来表示字符串，反之亦然。例如：

```
a = "hello 'world'"
print(f'a is {a}')
```

程序输出：

```
a is hello 'world'
```

单引号和双引号表示的字符串都只能是单行的，如果字面内容是多行的，那么可以用三引号来表示字符串，对于三引号当中的字面内容，Python 将原封不动地保存。例如：

```
a = '''
    1 2 3 5 6
  7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30
'''

print(a)
```

输出：

```
    1 2 3 5 6
  7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30
```

字符串是一个序列，这个序列使用整数作为索引，索引从 0 开始。可以根据索引提取字符串其中的一个字符，例如：

```
a = 'hello'
b = a[1]

print(f'a is {a}')
print(f'b is {b}')
```

输出：

```
a is hello  
b is e
```

如果要提取字符串的一个子串，可以用切片运算符，例如 `s[i:j]` 将会提取字符串 `s` 当中索引从 `i` 开始，一直到 `j-1` 位置的所有字符，例如：

```
a = '012345678'  
b = a[1:5]  
  
print(f'a is {a}')  
print('切片范围: [1:5]')  
print(f'b is {b}')
```

输出：

```
a is 012345678  
切片范围: [1:5]  
b is 1234
```

有的时候也可以省略开头和结尾的索引，例如：

```
a = '012345678' # 012345678  
  
b = a[:6]      # 012345  
c = a[5:]      # 5678  
d = a[3:7]     # 3456
```

如果要连接两个字符串，可以使用加号`+`，例如：

```
a = 'hello'
b = 'world'
c = a + b
print(c)    # helloworld
```

如果需要把字符串的字面值用来进行数值计算，那么需要用 `int()`、`float()` 等函数将字符串的字面值转换为数值，例如：

```
a = '33'
b = '95'
c = a + b    # 字符串连接
d = int(a) + int(b)  # 数值计算

print(f'c is {c}')
print(f'd is {d}')
```

输出为：

```
c is 3395
d is 128
```

同样地，也可以使用 `str()`、`repr()` 或 `format()` 函数，将其他非字符串的值转换为字符串，例如：

```
a = 100
b = 'a is ' + str(a)    # b: a is 100
c = 'a is ' + repr(a)   # c: a is 100
d = 'a is ' + format(a) # d: a is 100
```

在这里需要注意的是，`str()` 和 `repr()` 有一个区别，`str()` 所创建的字符串和通过 `print()` 输出的（也就是你看到的）是一致的，而 `repr(x)` 所创建的字符串是 `x` 对象的精确值。例如对于浮点数 `0.1`，在历史上，Py

thon 提示符和内置的 `repr()` 函数会选择具有 17 位有效数字的来显示，即 `0.10000000000000001`。从 Python 3.1 开始，Python（在大多数系统上）现在能够选择这些表示中最短的并简单地显示 `0.1`。所以我们在一般情况下已经很难分辨 `str()` 和 `repr()` 的区别了。例如：

```
>>> x = 0.1
>>> str(x)
'0.1'
>>> repr(x)
'0.1'
```

但这其实只是一个假象，计算机实际处理浮点数的时候，存储的仍然是一个浮点数的近似值，可以通过下面这个例子来验证：

```
x = 0.1
b = x + x + x == 0.3
print(f'x + x + x == 0.3 is {b}')
```

输出：

```
x + x + x == 0.3 is False
```

`format()` 函数可以将数值转换为特定格式的字符串，例如：

```
a = 3.14159265359
b = format(a, '0.5f') # 保留 5 位小数
print(f'b is {b}')
```

输出：

b is 3.14159

0.1.6 列表

列表是由对象组成的一个有顺序的序列，列表当中的元素可以是任何 Python 对象。把值放入到方括号[]当中就可以创建一个列表，例如：

```
colors = ['red', 'green', 'blue']
```

同字符串一样，列表的索引也是从 0 开始，使用索引运算符例如 `s[i]` 就可以访问并且修改列表 `s` 的第 `i` 个元素，例如：

```
a = colors[2]    # 返回列表的第三项"blue"  
colors[0] = 'black' # 把列表的第一项改为"black"
```

如果要在列表的末尾追加元素，可以使用 `append()` 方法：

```
colors.append('yellow')
```

如果要在列表当中插入一个元素，那么可以使用 `insert()` 方法：

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
print(f'before: {a}')  
  
a.insert(3, 100)  
print(f'after: {a}')
```

输出：

```
before: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
after:  [0, 1, 2, 100, 3, 4, 5, 6, 7, 8, 9]
```

列表的切片操作同样可以用于提取子串：

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
b = a[3:5]
print(f'b is {b}')    # b is [3, 4]
```

此外，列表的切片操作还可以用于对子串重新赋值，例如：

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
a[3:5] = 0, 0
print(f'b is {a}')
```

输出：

```
b is [0, 1, 2, 0, 0, 5, 6, 7, 8, 9]
```

使用加号+可以连接两个列表：

```
a = [1, 2, 3] + [7, 8, 9]    # 结果是 [1, 2, 3, 7, 8, 9]
```

列表可以包含任何类型的 Python 对象：

```
a = [1, 2, 3, 'hello', 4, 5]
```

甚至列表还可以包含其他的列表：

```
a = [1, 2, 3, [100, 101]]
a[1]    # 2
```

```
a[3] # [100, 101]  
a[3][0] # 100
```

对于列表，我们还可使用一种叫做推导式（list comprehension）的方式进行构造，例如我们将一个字符串形式表示的数字列表，转化为以数字形式表示的数字列表：

```
a = ['1', '2', '3', '4', '5']  
b = [int(x) for x in a]  
print(f'a is {a}')  
print(f'b is {b}')
```

输出：

```
a is ['1', '2', '3', '4', '5']  
b is [1, 2, 3, 4, 5]
```

上面的代码当中，表达式`[int(x) for x in a]`通过对列表 `a` 当中的每一个字符串进行循环，对每一个元素用 `int()` 函数进行类型转换，从而构造出一个新的列表。

0.1.7 元组

之前介绍的数据都是没有结构的，如果需要创建一些简单的数据结构，可以使用元组将一组值打包到一个对象当中。把一组值放到圆括号(,)当中就可以创建一个元组，例如：

```
point = (30, 40) # x 坐标 30、y 坐标 40 的一个点
address = ('www.ynu.edu.cn', 80) # 域名和端口号
student = (student_name, student_id, phone_number) # 姓名、学号、电话号码
```

一般来说，即使你不写括号，python 也能自动识别出元组，例如：

```
point = (30, 40) # x 坐标 30、y 坐标 40 的一个点
print(f'point is {point}')
```

输出：

```
point is (30, 40)
```

对于没有元素的空元组，和只有一个元素的元组，定义方式比较特殊：

```
a = () # 空元组
b = (10,) # 1 个元素的元组，注意逗号
c = 10, # 1 各元素的元组，省略括号，注意逗号
```

元组同样可以和列表一样，使用数字索引来访问其中的值，例如：

```
a = ('a', 'b', 'c')
```

```
a[0] # 'a'
```

但是，更多的情况下我们可以直接把元组解包成一组变量，例如：

```
a = ('A', 'B', 'C')
```

```
x, y, z = a
```

```
print(f'x = {x}, y = {y}, z = {z}')
```

输出：

```
x = A, y = B, z = C
```

虽然元组的大部分操作和列表相同，例如通过索引访问，切片，连接等。但是元组一旦创建就不能再修改它的内容。也就是无法替换、删除元组中的元素，也不能向元组当中添加元素。

所以一般来说，应该把元组看做是一个有多个组成部分的单一对象。相对的，列表应当看做是多个对象的一个集合。

元组和列表很类似，如果你忽视了元组而只使用列表，虽然列表用起来比元组灵活，但是如果你在计算机里创建了大量的小列表（比如列表包含的数量都小于 20 个），就会造成内存浪费。因为 Python 会给列表分配更多的内存。

我们经常同时使用列表和元组一起表示数据，比如我们编写一段程序来从 CSV 文件当中读取数据。

CSV 文件是一种用逗号把数据的每一列分隔开的文件格式。（在 Excel 以及苹果的 Numbers 当中，你都可以把数据表另存为 CSV 格式），例如下面这个 CSV 文件：

data.csv:

```
王同学,1001,1380000000  
张同学,1002,1830000000  
李同学,1003,1800000000
```

程序 1.7-1 读取 CSV 文件

```
file_name = 'data.csv'  
  
student_list = [] # 创建一个空列表  
for one_line in open(file_name):  
    fields = one_line.split(',') # 用分号进行切分  
    name = fields[0]  
    id = fields[1]  
    phone = fields[2]  
  
    one_student = (name, id, phone)  
  
    student_list.append(one_student)  
  
print(student_list)
```

输出：


```
[('王同学', '1001', '1380000000\n'), ('张同学', '1002', '1830000000\n'), ('李同学', '1003', '1800000000\n')]
```

`split()`方法会用你指定的字符，把一个字符串切分成子串列表。
上面这个程序从 csv 文件当中的学生信息读取进来放到 `student_list` 当中，这个 `student_list` 对象的结构类似一个二维的结构，分别用行索引和列索引来访问其中的对象，例如：

```
student_list[0] # 行索引: 0, 第 0 行, ('王同学', '1001', '1380000000\n')  
student_list[0][0] # 行索引: 0, 列索引: 0, 第 0 行第 0 列, '王同学'
```

此外，在 python 当中还可以利用迭代操作直接将列表当中的数据展开：

```
student_list = [  
    ('王同学', '1001', '1380000000'),  
    ('张同学', '1002', '1830000000'),  
    ('李同学', '1003', '1800000000')  
]  
  
for name, id, phone in student_list:  
    print(f'name is {name}, id is {id}, phone is {phone}')
```

输出：

```
name is 王同学, id is 1001, phone is 1380000000  
name is 张同学, id is 1002, phone is 1830000000  
name is 李同学, id is 1003, phone is 1800000000
```

一般情况，对于较长的列表赋值语句，我们推荐采用多行的方式编写，每一行一个元素。注意不要忘记元素之间的逗号。

0.1.8 集合

集合是一组无序的对象，并且集合中的对象不能重复。要创建集合，可以使用 `set()` 函数并且传入一系列的对象。例如：

```
a = set([1, 2, 3, 4, 5]) # 创建一个数值的集合
b = set('hello')        # 创建一个字符的集合

print(f's is {a}')
print(f't is {b}')
```

输出：

```
a is {1, 2, 3, 4, 5}
b is {'o', 'l', 'h', 'e'}
```

与列表和元组不同，集合是无序的，不能通过索引进行访问。此外，集合当中的元素不能有重复，例如上面的例子当中，字符串 `'hello'` 变成一个字符的集合之后，集合当中只会出现一次 `'l'` 字符。

Python 当中的集合支持一系列集合的标准操作，包括：并集、交集、差集、对称差集等，例如：

```
x = set([3, 4, 5, 6, 7, 8, 9])
y = set([1, 2, 3, 4, 5])

print(f'x is {x}')
print(f'y is {y}')

a = x | y # 并集
b = x & y # 交集
c = x - y # 差集，在 x 当中但是不在 y 当中
```

```
d = x ^ y # 对称差集, 只在x 当中或者只在y 当中 (异或)

print(f'x | y is {a}')
print(f'x & y is {b}')
print(f'x - y is {c}')
print(f'x ^ y is {d}')
```

输出:

```
x is {3, 4, 5, 6, 7, 8, 9}
y is {1, 2, 3, 4, 5}
x | y is {1, 2, 3, 4, 5, 6, 7, 8, 9}
x & y is {3, 4, 5}
x - y is {8, 9, 6, 7}
x ^ y is {1, 2, 6, 7, 8, 9}
```

如果要往集合当中添加元素, 可以使用 `add()` 或者 `update()` 函数:

```
a = set([1, 2, 3])
print(f'a is {a}')

a.add(9) # add() 方法每次只能添加一个元素
print(f'add one: a is {a}')

a.update([100, 101, 102]) # update 方法每次可以添加多个元素
print(f'add multiple: a is {a}')
```

输出:

```
a is {1, 2, 3}
添加一个元素: a is {1, 2, 3, 9}
添加多个元素: a is {1, 2, 3, 100, 101, 102, 9}
```

注意: `add()` 函数每次只能向集合当中添加一个元素, 而 `update()` 函数则可以每次向集合添加多个元素。

如果要从集合当中删除元素，则调用集合的 `remove()` 方法：

```
a = set([1, 2, 3])  
print(f'a is {a}')  
  
a.remove(3)  
print(f'removed: a is {a}')
```

输出：

```
a is {1, 2, 3}  
removed: a is {1, 2}
```

0.1.9 字典

字典是一个由“键-值”对组成的散列表，类似于 Java 当中的 HashMap。通过键进行索引。在打括号{、}中放入“键-值”对就可以创建字典：

```
a = {  
    'name': '张同学',  
    'id': '1001',  
    'phone': '13800000000'  
}  
  
print(f'a is {a}')
```

输出：

```
a is {'name': '张同学', 'id': '1001', 'phone': '13800000000'}
```

如果要访问字典当中的元素，要用键索引运算符：

```
a['name'] # '张同学'  
a['id']   # '1001'
```

同样地，通过键索引运算符，还可修改对象或者插入新的对象，例如：

修改对象：

```
a = {  
    'name': '张同学',  
    'id': '1001',
```

```
'phone': '13800000000'
}

print(f'a is {a}')

a['phone'] = '18300000000'
print(f'changed phone: a is {a}')
```

输出：

```
a is {'name': '张同学', 'id': '1001', 'phone': '13800000000'}
changed phone: a is {'name': '张同学', 'id': '1001', 'phone': '18300000000'}
```

插入对象：

```
a = {
    'name': '张同学',
    'id': '1001',
    'phone': '13800000000'
}

print(f'a is {a}')

a['age'] = 12
print(f'add age: a is {a}')
```

输出：

```
a is {'name': '张同学', 'id': '1001', 'phone': '13800000000'}
add age: a is {'name': '张同学', 'id': '1001', 'phone': '13800000000', 'age': 12}
```

字典是一种很有用的数据结构。同时，字典也可以作为一种可以快速查找的容器，可以把数据以“键-值”对的方式添加到字典当中，当

需要的时候，可以直接访问（而不用像列表那样，需要通过索引依次迭代遍历各个项，才能找到所需要的对象）

例如，一个储存网址的字典：

```
a = {  
    '腾讯网': 'www.qq.com',  
    '百度': 'www.baidu.com',  
    '云南大学': 'www.ynu.edu.cn'  
}
```

有两种方式创建一个空字典：

```
a = {}    # 一个空字典  
b = dict() # 另一个空字典
```

如果要检查某个键在不在字典当中，可以用 `in` 运算符：

```
a = {  
    '腾讯网': 'www.qq.com',  
    '百度': 'www.baidu.com',  
    '云南大学': 'www.ynu.edu.cn'  
}  
  
if '腾讯网' in a:  
    url = a['腾讯网']  
else:  
    url = 'not found'  
  
print(url)
```

输出：


```
www.qq.com
```

上述检查还可以写成更加简练的形式：

```
url = a.get('腾讯网', 'not found')
```

如果我们要遍历整个字典，那么首先必须获得键的列表，这只需要用 `list()` 函数把字典转换为一个列表即可：

```
a = {  
    '腾讯网': 'www.qq.com',  
    '百度': 'www.baidu.com',  
    '云南大学': 'www.ynu.edu.cn'  
}  
  
key_list = list(a)  
print(key_list)
```

输出：

```
['腾讯网', '百度', '云南大学']
```

第二章 综合练习(一)

本章将在上一章介绍的 Python 基础知识的基础上，引入一些编程案例进行强化练习

0.2.1 习题 1：分别定义四种基本类型的变量

解答：

```
a: float = 10.0  
b: int = 10  
c: bool = True  
d: str = 'hello'
```

上述代码当中采用了 Python3.5 之后的版本当中新增加的类型标注（Type Hint），例如在变量后面标注上变量的类型。这些标注并不是可执行的代码，也就是说虽然把一个变量，比如 **b** 标注成 **int** 类型，但是实际执行的时候，仍然是可以通过赋值把它绑定到其它类型的对象上面。这个类型标注的目的，只是在程序运行之前进行静态代码检查的时候，给静态检查器提供一些参考。但是为了便于读者更清晰地阅读代码，本书中的例子程序将尽量把类型标注加上。

Python 当中的所有数据都是对象，对象是 Python 当中对数据的一种抽象。

Python 当中常用的数据类型主要有如下几种：

数字类型 - 整数（**int**） - 浮点数（**float**） - 布尔值（**boolean**）

序列类型 - 字符串（**str**） - 元组（**tuple**） - 列表（**list**）

集合类型 - 集合（**set**）

映射类型 - 字典（**dict**）

更多信息可以查看 Python 官方文档: <https://docs.python.org/zh-cn/3/reference/datamodel.html#the-standard-type-hierarchy>

0.2.2 习题 2：定义一个列表，一个集合，一个元组和一个字典

解答：

```
from typing import List, Set, Tuple, Dict

a: List = [1, 2, 3, 4, 5]
b: Set = set([1, 2, 3])
c: Tuple = ('A', 'B', 'C', 'D')
d: Dict = {
    1: '星期一',
    2: '星期二',
    3: '星期三',
    4: '星期四',
    5: '星期五',
    6: '星期六',
    7: '星期日'
}
```

除了基本类型之外，列表、集合、元组和字典的代码标注，需要使用 `typing` 包里面的类型别名：

```
列表 list() typing.List
集合 set() typing.Set
元组 () typing.Tuple
字典 dict() typing.Dict
```

0.2.3 习题 3：求一系列数字的平方

问题：有一个列表 `a: List = [1, 2, 3, 4, 5]`，请求出这个列表中每一个元素的平方，然后放到一个新的列表当中。

对于这个问题我们可以采取迭代的方法解决：

```
from typing import List
a: List = [1, 2, 3, 4, 5]

b: List = []
for x in a:
    y = x * x
    b.append(y)

print(f'a is {a}')
print(f'b is {b}')
```

输出：

```
a is [1, 2, 3, 4, 5]
b is [1, 4, 9, 16, 25]
```

先定义一个空的列表 `b` 作为储存计算结果的容器。接下来用 `for` 关键字依次访问列表 `a` 当中的元素，并保存到临时变量 `x` 当中，接下来计算 `x*x` 的值，储存到变量 `y` 中。最后把 `y` 追加到列表 `b` 当中。当迭代完成之后，储存结果的新列表 `b` 也就生成好了。

如果使用我们之前提到的推导式，这个问题还有另一种更简洁的解决方式：

```
from typing import List
a: List = [1, 2, 3, 4, 5]
b = [x * x for x in a]

print(f'a is {a}')
print(f'b is {b}')
```

输出：

```
a is [1, 2, 3, 4, 5]
b is [1, 4, 9, 16, 25]
```

除此之外，还可以用函数式编程的方式，即 `map()` 函数和 `lambda` 关键字来解决这个问题：

```
from typing import List
a: List = [1, 2, 3, 4, 5]
b: List = map(lambda x: x * x, a)
b = list(b)

print(f'a is {a}')
print(f'b is {b}')
```

输出：

```
a is [1, 2, 3, 4, 5]
b is [1, 4, 9, 16, 25]
```

使用 `lambda` 运算符可以创建表达式形式的匿名函数：

```
lambda args: expression
```

`args` 是以逗号分隔的一系列参数，`expression` 是用这些参数计算出某个值的表达式，而且这个值就是这个 `lambda` 语句的值。例如：

```
a = lambda x, y: x + y
b = a(4, 5)
print(f'b is {b}')
```

输出：

```
b is 9
```

而 `map()` 是 `python` 其中的一个内置函数：

```
map(function, items, ...)
```

`map()` 函数的功能是对列表当中的每一个元素，分别作为参数输入到 `function` 当中进行计算，并把计算的结果依次放到一个新的列表当中。

关于 `map()` 函数和 `lambda` 关键字的相关内容，将在后面的章节进行介绍。在这里需要注意的是，在 `Python3` 当中 `map()` 函数的返回值是一个迭代器，也就是说要在程序访问其中的值的时候，值才会被计算出来。这是因为在 `Python` 当中的列表使用非常方便，很多程序员会随意地创建很多列表，这就会造成很大的内存浪费，所以到了 `Python3` 当中，为了避免这种不必要的资源浪费，`map()` 函数生成的列表就被做成了迭代器。

0.2.4 习题 4：求一系列数字的平方和

问题：有一个列表 `a: List = [1, 2, 3, 4, 5]`，求这个列表当中所有元素的平方和

同样的，对于这个问题，首先想到的仍然是迭代的方法：

```
from typing import List
a: List = [1, 2, 3, 4, 5]

sum: int = 0
for x in a:
    y = x * x
    sum = sum + y

print(f'sum is {sum}')
```

输出：

```
sum is 55
```

由于最后的结果并不是一个列表，所以这里不能用推导式的方式解决。然而，我们可以使用 Python 其中的一个工具函数来解决这个问题：

```
from typing import List
from functools import reduce

a: List = [1, 2, 3, 4, 5]

print(f'a is {a}')
print(f'b is {b}')
```

输出：

```
a is [1, 2, 3, 4, 5]  
b is 55
```

`reduce()`函数的功能是依次取出列表中的下一个元素，和之前的计算结果一起，传入 `lambda` 函数当中，得到的值形成新的计算结果。例如上例中：

```
lambda x, y: x + y*y
```

之前的计算结果作为参数 `x` 传入，列表中的下一个元素作为参数 `y` 传入，于是平方和就表示为 `x + y*y`

0.2.5 习题 5：列出 20 以内的所有偶数

同样，我们首先用迭代的方式解决这个问题：

```
from typing import List
a: List = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

b: List = []
for x in a:
    if x % 2 == 0:
        b.append(x)

print(f'b is {b}')
```

输出：

```
b is [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

在 Python 当中，用 % 运算符求余数。当一个数字处以 2 的余数为 0 时，这个数就是一个偶数。

在 Python 当中，可以使用内置的 range() 函数生成一个等差数列的列表迭代器：

```
a: List = range(10) # a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

于是我们采用推导式可以有第二种解决方式：

```
from typing import List
a: List = range(20)
b: List = [x for x in a if x % 2 == 0]
print(f'b is {b}')
```

在推导式当中，可以在后面加一个 if 子句对迭代的结果进行过滤

对于要从一个列表当中过滤出一部分元素，在 Python 当中有一个内置函数可以使用：

```
from typing import List
a: List = range(20)
b: List = list(filter(lambda x: x % 2 == 0, a))
print(f'b is {b}')
```

filter 函数的形式是：

```
filter(function, iterable)
```

iterable 是一个迭代器，在我们的例子当中是一个列表，function 是一个判断函数。filter() 函数将依次把列表当中的每一个元素传入 function 进行计算，如果计算的结果是 True 则此元素将保留到过滤的结果当中。

0.2.6 习题 6：计算 30 以内的质数。

问题：质数又叫做素数，是指在大于 1 的自然数当中，除了 1 和它本身之外不能被其它自然数整除的数。请列出 30 以内的质数。

这里我们将综合运用之前学过的知识来解决这个问题。要找出 30 以内的质数，我们只需要把 30 以内的所有自然数列出来作为一个集合，然后从这个集合当中减去不是质数的数字组成的集合，剩下的子集合就是质数。

首先我们构造 30 以内的所有自然数的集合。

```
a: List = range(2, 30)
```

`range()`函数如果传入两个参数，那么第一个参数是开始的数字，第二个数字是结束的数字加 1。由于最小的质数是 2，所以我们只需要从 2 开始即可。

根据定义，由有两个不为 1 的自然数相乘得到的数，就必定不是质数。所以 30 以内的非质数列表可以通过推导式得到：

```
b: List = [x * y for x in a for y in a if x*y < 30]
```

推导式可以对多个变量进行推导，在这里我们分别对 `x` 和 `y` 两个变量进行了推导，最后用一个 `if` 子句过滤出 30 以内的元素。

为了进行集合计算，我们首先需要把两个列表转换为集合：

```
a_set: Set = set(a)
b_set: Set = set(b)
```

最后，质数的集合就是两个集合的差：

```
prime_set: Set = a_set - b_set
```

这样我们就得到了 30 以内的所有质数。完整的代码如下：

```
from typing import List, Set

a: List = range(2, 30)
b: List = [x * y for x in a for y in a if x * y < 30]

a_set: Set = set(a)
b_set: Set = set(b)

prime_set: Set = a_set - b_set

print(f'prime_set is {prime_set}')
```

输出：

```
prime_set is {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

这种算法的效率是很低的，你还可以找到更多更高效的求质数的方法。

0.2.7 习题 7：计算平面上两点间的距离

问题：点 A 的坐标是 (3, 0)，点 B 的坐标是 (0, 4)，计算两点间的距离 R。

二维平面上的点坐标分别有两个分量，所以我们可以考虑用元组来表示点的坐标，于是有：

```
position_a: Tuple = (3, 0)
position_b: Tuple = (0, 4)
```

根据勾股定理，二维平面上两点间的距离公式是：

```
r = sqrt( (x2-x1)**2 + (y2-y1)**2 )
```

sqrt()是求二次根号的函数，这个函数并不是内置函数，需要从 math 包当中导入。在 python 当中，两个乘号**表示乘方，x**2 就表示 x 的平方。于是求两点间的距离可以写作：

```
r = sqrt(
    (position_a[0] - position_b[0]) ** 2
    +
    (position_a[1] - position_b[1]) ** 2
)
```

对于较长的表达式，我们一般写成多行。完整代码如下：

```
from typing import Tuple
from math import sqrt
```

```
position_a: Tuple = (3, 0)
position_b: Tuple = (0, 4)

r = sqrt(
    (position_a[0] - position_b[0]) ** 2
    +
    (position_a[1] - position_b[1]) ** 2
)

print(f'r is {r}')
```

输出：

```
r is 5.0
```


第三章 Python 基础（二）

0.3.1 迭代与循环

在 Python 当中，最常用循环结构是 `for` 语句，它可以用来对容器（列表、元组等）当中的元素进行迭代操作。最常见的迭代形式是通过循环依次访问一个序列当中的元素（例如字符串、列表等，例如：

```
for x in [1, 2, 3, 4, 5]:  
    print(f'x is {x}')
```

输出：

```
x is 1  
x is 2  
x is 3  
x is 4  
x is 5
```

在这个例子中，每次迭代都会把列表中的下一个元素赋值给变量 `x`

像这样在整数范围内进行迭代的情况非常常见，所以 Python 当中有一个内置的 `range()` 函数可以快捷地生成一个整数的列表：

```
range(start, stop, step)
```

`start` 表示开始的整数，`stop` 表示结束的边界，即列表的最后一个整数之后的下一个整数。（列表当中整数的范围是 `start <= x < end`）。`step` 表示步长。

在使用 `range()` 函数的时候，如果起始的 `start` 被省略，会默认为从 0 开始。如果第三个参数 `step` 被省略，则会默认为 1。例如：

```
a = range(5)      # a = [0, 1, 2, 3, 4]
b = range(1, 5)   # a = [1, 2, 3, 4]
c = range(0, 10, 3) # a = [0, 3, 6, 9]
d = range(5, 1, -1) # a = [5, 4, 3, 2]
```

在使用 `range()` 函数的时候需要注意的是，为了防止程序员在不经意间生成很多的 list 进而耗尽内存，在 Python3 当中 `range()` 函数的返回值并不是一个列表，而是一个迭代器：

```
a = range(5)
print(f'a is {a}')
```

输出：

```
a is range(0, 5)
```

如果确实需要一次性把列表当中所有的值都生成出来，需要用 `list()` 函数将迭代器对象转换为列表对象：

```
a = range(5)
print(f'a is {a}')
```



```
a_list = list(a)
print(f'a_list is {a_list}')
```

输出：

```
a is range(0, 5)
a_list is [0, 1, 2, 3, 4]
```

除了处理整数序列之外，for 语句还可以用于迭代字符串、列表、字典、文件等等。

分别访问字符串当中的每一个字符：

```
a = 'hello'
for c in a:
    print(f'c is {c}')
```

输出：

```
c is h
c is e
c is l
c is l
c is o
```

分别访问列表中的每一个元素：

```
a = ['A', 'B', 'C', 'Hello', 100, 200, 300]

for x in a:
    print(f'x is {x}')
```

输出：

```
x is A
x is B
```

```
x is C
x is Hello
x is 100
x is 200
x is 300
```

分别访问一个字典当中的每一个成员：

```
a = {
    'tencent': 'www.qq.com',
    'baidu': 'www.baidu.com',
    'yunnan university': 'www.ynu.edu.cn'
}

for key in a:
    print(f"key is '{key}', a[{key}] is {a[key]}")
```

输出：

```
key is 'tencent', a[tencent] is www.qq.com
key is 'baidu', a[baidu] is www.baidu.com
key is 'yunnan university', a[yunnan university] is www.ynu.edu.cn
```

另外，for 语句还可以迭代访问一个文件的每一行，首先我们创建一个文本文件 abc.txt:

```
aaaaa
bbbbbb
cccccc

ddddd

eeeeee
```

接下来用 for 语句依次访问这个文件的每一行：

```
f = open('abc.txt')
for line in f:
    print(f"line is '{line}'")
```

输出：

```
line is 'aaaaa
'
line is 'bbbbb
'
line is 'ccccc
'
line is '
'
line is 'ddddd
'
line is '
'
line is 'eeeeee'
```

注意输出当中引号的位置可以发现，每一行末尾的换行符也被读取出来了。在有些情况下我们并不希望把换行符去掉。通常可以使用字符串对象的 `strip()` 方法来消除一个字符串两端的空格、TAB、换行等字符：

```
f = open('abc.txt')
for line in f:
    striped_line = line.strip()
    print(f"striped '{striped_line}'")
```

输出：

```
striped 'aaaaa'  
striped 'bbbbb'  
striped 'ccccc'  
striped ''  
striped 'ddddd'  
striped ''  
striped 'eeeee'
```

`for` 循环是 Python 当中最强大的语言特性之一，在后面的章节你会学习如何创建自定义的迭代器对象为 `for` 语句提供用以迭代的序列。

0.3.2 函数

在 Python 当中，使用 `def` 关键字可以创建一个函数：

```
def 函数名(参数列表):  
    函数体
```

定义函数的时候，函数体需要缩进，例如：

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

`return` 语句定义了这个函数的返回值。并不是所有的函数都有返回值，`return` 语句并不是必须的。

这样就创建了一个 `add()` 函数，这个函数接收两个参数 `a` 和 `b`。这个函数的功能是计算 `a`、`b` 两个数的和。要调用函数，只需要在函数名的后面，加上用括号 `()` 括起来的参数。例如：

```
x = add(3, 4) # x = 7
```

如果要想函数返回多个值，可以使用元组作为储存返回值的容器，例如求一个数除以另一个数的商和余数：

```
def divide(a, b):  
    q = a // b # // 是 Floor 除法，计算结果只保留整数部分  
    r = a % b  # % 是取模运算，计算结果只保留余数部分  
    return (q, r)
```



```
x, y = divide(10, 3) # 元组自动解包
print(f'x is {x}, y is {y}')
```

输出：

```
x is 3, y is 1
```

如果要给函数的参数设置一个默认值，可以直接使用下面这种赋值的方式：

```
def max(a, b = 0):
    if a > b:
        return a
    else:
        return b

c = max(3, 5) # c = 5
d = max(10)  # d = 10
e = max(-1)  # e = 0
```

这里 **b** 的默认值为 0，如果没有传入参数 **b**，那么函数就会取默认值。

在函数当中创建变量或者给变量赋值的时候，这个变量是局部变量，也就是变量的作用域只局限在函数的函数体当中。当函数执行结束，函数中的局部变量就会被销毁。

```
a = 100

def foo(x):
    a = x
    print(f'in foo(): a is {a}')
```

```
print(f'a is {a}')  
foo(5)  
print(f'after called foo(5): a is {a}')
```

输出：

```
a is 100  
in foo(): a is 5  
after called foo(5): a is 100
```

如果要在函数里修改某个全局变量，需要使用 `global` 语句：

```
def foo(x):  
    global a  # 声明此处的 a 即时全局变量当中同名的 a  
    a = x  
    print(f'in foo(): a is {a}')  
  
print(f'a is {a}')  
foo(5)  
print(f'after called foo(5): a is {a}')
```

输出：

```
a is 100  
in foo(): a is 5  
after called foo(5): a is 5
```

在使用 `global` 语句声明全局变量之后，函数里的变量名 `a` 就不再是局部变量，而是全局变量当中的同名变量 `a` 了。

0.3.3 生成器

在之前的章节中，我们如果需要一个序列，一般会采用推导式的方式生成一个列表。但是有时候这样做会消耗大量的内存，例如，如果我们要依次访问十亿个自然数，那么是不是我们必须先生成一个长度为十亿的列表，然后用 `for` 循环依次迭代呢？

在 `Python` 当中提供了一种叫做生成器的功能，可以在具体执行的时候才临时把值计算出来，对于上面的情况，我们就只需要每次计算出下一个自然数，而不用一开始就把十亿个自然数都放到内存当中去。例如：

```
def natural_number(limit):  
    n = 1  
    while n < limit:  
        yield n      # 把 n 的值传给生成器对象  
        n = n + 1
```

通过 `yield` 关键字，我们把 `n` 的值传送给生成器。此时 `while` 循环会暂时停下来，等生成器取走当前的值之后，循环才会继续进行下一轮。

接下来，定义十亿以内的自然数对象，通过 `next()` 函数依次得到下一个值：

```
natu = natural_number(1000000000)  
  
a = next(natu)    # a = 1  
b = next(natu)    # b = 2  
c = next(natu)    # c = 3
```

一般情况下我们会把生成器和 for 循环配合使用，for 循环会自动遍历生成器产生的序列：

```
def natural_number(limit):  
    n = 1  
    while n < limit:  
        yield n      # 把 n 的值传给生成器对象  
        n = n + 1  
  
natu = natural_number(5)  
  
for i in natu:  
    print(f'i is {i}')
```

输出：

```
i is 1  
i is 2  
i is 3  
i is 4
```

如果我们需要一次性把生成器产生的序列都取出来，可以用 list() 函数把生成器转换为一个列表：

```
def natural_number(limit):  
    n = 1  
    while n < limit:  
        yield n      # 把 n 的值传给生成器对象  
        n = n + 1  
  
natu = natural_number(5)  
print(f'natu is {natu}')  
  
natu_list = list(natu)  
print(f'natu_list is {natu_list}')
```

输出：

```
natu is <generator object natural_number at 0x107235a98>
natu_list is [1, 2, 3, 4]
```

对于生成器，还有一种更为简便的定义方法，就是在列表推导式里，把中括号[]换成圆括号()即可：

```
a = (2**x for x in range(1000000000))
print(f'a is {a}')

print(f'next(a) is {next(a)}')
print(f'next(a) is {next(a)}')
print(f'next(a) is {next(a)}')
print(f'next(a) is {next(a)}')
print(f'next(a) is {next(a)}')
```

输出：

```
next(a) is 1
next(a) is 2
next(a) is 4
next(a) is 8
next(a) is 16
```

上面例子当中，如果把推导式两端的圆括号改为中括号，即直接生成一个列表，那么程序会占用巨大的内存，并且需要消耗很长的时间才能创建完成。请不要在你的电脑上尝试下面的代码，因为创建这个列表会消耗巨大的内存，并且会执行很长时间：

```
a = [2**x for x in range(1000000000)]  
print(f'a is {a}')
```

0.3.4 对象和类

在 Python 当中所有的值都是对象。对象是由内部数据和对数据操作的方法组成的。例如之前处理字符串和列表这些内置类的时候，就已经用到了对象的方法了。例如：

```
numbers = [1, 2, 3, 4, 5] # 创建一个列表对象

a = numbers.pop() # 调用对象的方法
print(f'numbers after pop is {numbers}')
print(f'a is {a}')
```

输出：

```
numbers after pop is [1, 2, 3, 4]
a is 5
```

对象的方法和之前提到的函数非常类似，二者一个最大的区别是函数是可以独立存在的，而方法必须要附着在某一个对象上面，是对象的一部分。（其实函数本身也是一个对象）

有一个内置的 `dir()` 函数，可以把一个对象都有哪些方法全部列出来。在交互模式下面，这个方法是一个很有效的实验工具。例如：

```
>>> numbers = [1, 2, 3, 4, 5]
>>> dir(numbers)
['_add_', '_class_', '_contains_', '_delattr_', '_delitem_', '_dir_', '_doc_', '_eq_', '_format_', '_ge_', '_getattribute_', '_getitem_', '_gt_', '_hash_', '_iadd_', '_imul_', '_init_', '_init_subclass_', '_iter_', '_le_', '_len_', '_lt_', '_mul_', '_ne_', '_new_', '_reduce_', '_reduce_ex_', '_repr_', '_reversed_', '_rmul_', '_setattr_', '_setitem_', '_sizeof_', '_str_', '_subclasshook_', 'append', 'clear', 'copy', 'count',
```

```
'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']  
>>>
```

在上面列出的这些方法当中，除了我们熟悉的 `append()`、`insert()` 方法以外，还有很多以两个英文下划线_开头的特俗方法。这些是对象的内置方法，用来实现对象的各种功能。例如，如果两个对象相加，即用+进行计算，那么具体操作就是在`__add__()`方法中定义的。所以也可以直接调用`__add__()`函数

```
a = [1, 2]  
b = [4, 5]  
  
c = a + b  
print(f'c is {c}')  
  
d = a.__add__(b)  
print(f'd is {d}')
```

输出：

```
c is [1, 2, 4, 5]  
d is [1, 2, 4, 5]
```

但是一般来说，不建议直接使用类的内置函数。

`class` 关键字可以用来定义一个新的对象类型，实现面向对象编程。如果你只能使用系统现有的对象类型而不能创建新的类型，这样的编程称为基于对象的编程，比如 `JavaScript` 和早起的 `ActionScript1.0` 就是这样的。如果你不仅可以使系统内置的对象类型，还可以创建

新的对象类型，这样的编程就称为面向对象编程。目前的主流编程语言例如 Java、C++、C#、Go 和 Python 都支持面向对象的编程方式。

例如，我们不满足于系统内置的 list 类型，我们想创建一种新的类型来实现堆栈（Stack）的功能，并且支持 push()、pop()和 length()操作，那么我们可以用 class 关键字来创建这个类型，例如：

```
class Stack(object):  
  
    def __init__(self): # 初始化一个堆栈对象  
        self.data = []  
  
    def push(self, item):  
        self.data.append(item)  
  
    def pop(self):  
        return self.data.pop() # list 对象已经默认实现了 pop()  
  
    def length(self):  
        return len(self.data)
```

在类型的定义当中，第一行 class Stack(object)声明了一个叫做 Stack 的新类型，圆括号当中的 object 表示这个类型继承自 Python 内置的 object 类型，object 类型是 Python 当中所有类型的根类型。在 Python 的类型定义当中，在类型名称后面用圆括号(、)表示继承（Extend，也被称为“派生”）关系。

类的定义当中，用 def 语句定义类的方法。定义类的方法和定义函数非常类似，一个重要的区别是方法的第一个参数必须是指向对象本身变量，这个特殊变量的名字必须是 self。在方法当中，涉及到对象本身的所有操作，都必须显式地引用 self 变量。

如果需要知道一个列表当中有多少元素，可以使用内置的 `len()` 函数：

```
def length(self):  
    return len(self.data) # 返回列表对象 data 的元素个数
```

在类的定义当中，以两个英文下划线 `_` 开始和结束的方法，都是类当中的特殊方法。例如：

```
def __init__(self):  
    # 进行对象初始化工作  
    self.data = []
```

`init()` 方法就是用于在创建此类型的对象时初始化对象的状态的。比如在这里例子里创建了一个列表对象用于储存数据。

类型是一个抽象的概念，我们不能说使用一个类，在计算机程序当中可以直接使用的都是对象，所以应当首先用这个类型创建出一些对象，然后通过调用对象的方法来使用所需的功能。例如：

```
# 定义类型  
class Stack(object):  
  
    def __init__(self): # 初始化一个堆栈对象  
        self.data = []  
  
    def push(self, item):  
        self.data.append(item)  
  
    def pop(self):  
        return self.data.pop() # list 对象已经默认实现了 pop()
```

```
def length(self):  
    return len(self.data)  
  
# 用类型创建对象  
s = Stack()  
  
# 使用对象的功能  
  
s.push('A')  
s.push('B')  
s.push([1, 2, 3])  
  
size = s.length() # size = 3  
  
a = s.pop() # a = [1, 2, 3]  
b = s.pop() # b = 'B'
```

创建一个对象是用类型的名称后面加一对圆括号(,)。有时候创建对象的时候可以向圆括号当中填入一些参数用于对象的初始化。初始化参数需要在`_init_()`方法当中定义。这里需要注意的是，`init()`方法当中的 `self` 参数在类型定义之外是不可见的，也就是说如果`_init_()`方法只有一个 `self` 参数，那么在创建对象的时候就不需要填入参数。

最后，如果需要删除一个对象，可以使用 `del` 语句：

```
s = Stack() # 创建对象  
  
del s # 删除对象
```

要注意的是，`del` 语句所进行的操作只是把对象的名字和对象本身进行解绑，如果还有其它的名字绑定到这个对象上的话，对象仍然会继续存在。直到所有的名字都已经被解绑之后，对象才会被销毁。在

对象使用结束之后，如果程序还要继续运行，那么手动调用 `del` 语句删除对象是一个良好的习惯，这有助于 `Python` 进行内存回收，避免程序占用过多地占用不必要的内存。

上面的例子创建了一个全新的对象来实现堆栈的功能。但是事实上这里堆栈的功能和 `Python` 内置的列表对象几乎完全一致，所以可以采用另一种方式来创建 `Stack` 这个类型：

```
class Stack(list): # 继承（派生）list 类型

    def push(self, item): # 只需要补充定义增加的方法
        self.append(item)
```

在定义类型的时候，如果要继承（派生）现有的类型，只需要在 `class` 语句当中，在类型名字的圆括号里填入要继承的类型名字就可以了。继承了一个类型之后，就可以使用这个类型的所有方法，并且在新的类型定义当中补充上需要增加的额外方法即可。

一般来说，类型定义当中的方法，都只能被具体的对象使用。但是也可以定义一种特殊的方法，这种方法不需要创建对象，可以直接通过类型名称来使用。这种方法被称为“静态方法”，例如：

```
f = open('not_exist.txt')
```

输出：

```
a is 27
```

`@staticmethod` 这样的写法，叫做“装饰器”，装饰器是一种特殊的函数。在这个例子当中使用了一个 Python 内置的装饰器 `@staticmethod`，这个装饰器把类定义其中的一个方法声明为静态方法。静态方法由于不关联到任何一个具体对象，所以不需要传递 `self` 参数作。

0.3.5 异常

如果在程序运行过程当中，遇到意料之外的情况，就会引发异常。比如程序要打开一个文件，但是这个文件并不存在，程序就会因为无法继续运行而停止，并且打印出回溯信息以提示异常发生的原因：

```
f = open('not_exists.txt')
```

将输出下面这样的会输信息：

```
Traceback (most recent call last):  
  File "scratch.py", line 1, in <module>  
    f = open('not_exists.txt')  
FileNotFoundError: [Errno 2] No such file or directory: 'not_exists.txt'
```

这个回溯信息指出异常的类型 `FileNotFoundError`，异常的相关信息 `No such file or directory: 'not_exists.txt'`，同时还指出了发生异常的源文件名和发生异常的行号 `File "scratch.py", line 1`。

一般情况下，发生异常之后，程序将因为无法继续执行而退出。但是我们也可以在程序当中捕捉异常，并对异常进行处理，使得程序可以继续执行下去。捕捉异常使用 `try ... except` 语句，例如：

```
try:  
    f = open('not_exists.txt')  
except FileNotFoundError as ex:  
    print(f'捕捉到异常 : {ex}')  
  
print('程序正常退出')
```

通过 `except` 语句定向捕捉了 `FileNotFoundError` 这种异常情况，代码中定义了当这种异常发生的时候，简单地把异常信息打印出来，然后让程序继续正常执行。

`except` 语句需要明确指明需要捕捉的异常，如果程序遇到的异常情况不在 `except` 语句捕捉的名单当中，那么程序仍然会因为不知道如何处理，无法继续运行而终止。

除了程序在执行过程当中自发地遇到异常情况之外，也可以在代码当中人工抛出异常。手工抛出异常使用 `raise` 语句，例如：

```
raise RuntimeError('这是一个异常')
```

输出：

```
Traceback (most recent call last):
  File "scratch.py", line 1, in <module>
    raise RuntimeError('这是一个异常')
RuntimeError: 这是一个异常
```

下面是一个需要在代码当中人为抛出异常的例子：

```
age = int(input('请输入您的年龄: '))

if age < 0:
    raise RuntimeError('年龄小于 0')
```

输出：

```
请输入您的年龄: -2
Traceback (most recent call last):
  File "scratch.py", line 4, in <module>
    raise RuntimeError('年龄小于 0')
RuntimeError: 年龄小于 0

Process finished with exit code 1
```

在这种情况下，必须人为抛出异常让程序终止，否则如果继续执行下去，后续计算得到的结果只能是错上加错。

0.3.6 模块

随着程序越来越大，代码越来越多，为了方便开发和维护，就需要把代码分别放到多个不同的文件当中。在 Python 当中，就可以把一个文件中事先写好的代码作为模块导入另外一个文件的代码当中。导入模块使用 `import` 语句，例如：

```
import math
```

导入了标准库当中的 `math` 模块。除了使用标准库当中提供的模块之外，你也可以创建自己的模块。

要创建模块，可以将相关的定义和语句放入到一个文件当中，这个文件的文件名必须以 `.py` 作为后缀，文件的名称，即是模块名称。例如：

```
# 文件名: div.py

def divide(a, b):
    q = a // b
    r = a % b
    return q, r
```

上面这段代码被放到一个叫做 `div.py` 的文本文件当中，模块名就是 `div`。如果要在另一个文件当中使用这个模块，只需要用 `import` 语句加载这个模块：

```
import div
```

```
a, b = div.divide(10, 3) # a = 3, b = 1
```

`import` 语句创建了一个新的命名空间，并在这个命名空间当中执行与导入的.py 文件相关的所有语句。如果要访问命名空间当中的内容，只需要在你要访问的名称前面加上模块名称作为前缀，例如上面例子中的 `div.divide()`

有时候模块的名称比较长，每次引用都要加上名称前缀会造成代码过长，而这与 Python 追求代码简练的目标是不符合的，所以在 Python 当中，也可以用 `import ... as` 语句为引入的包设置一个别名，例如：

```
import matplotlib.pyplot as plt
```

就把 `matplotlib.pyplot` 这个包取了一个叫做 `plt` 的短别名。后面代码当中只需要用 `plt.show()` 就可以调用到这个包的名字空间里的名字。

如果要直接把模块里面的名字导入到当前的名字空间中，可以用 `from ... import` 语句：

```
from div import divide
```

```
a, b = div.divide(10, 3) # a = 3, b = 1
```

这样把 `divide` 这个名字导入到当前的名字空间当中，可以像使用在当前文件当中定义的名字一样直接使用导入进来的名字（不需要加前缀）。甚至，如果要把模块当中的所有名字都导入到当前名字空间当中可以用*通配符，例如：

```
from div import *
```

但是直接导入所有名字的方式并不推荐，因为 `Python` 是解释执行的语言，导入模块中的名字，意味着模块中涉及到的代码都会被执行一次，如果这些被导入的代码的计算量比较大，就会浪费过多的 `CPU` 时间。

同样地，与对象一样，对于模块同样可以使用 `dir()` 函数列出模块当中的元素，这是在交互模式中进行试验的一种很有效的方式：

```
>>> import math
>>> dir(math)
['_doc_', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
>>>
```

0.3.7 文档

在使用 Python 的时候，除了教程上提到的内容之外，还有大量教程未提及的说明文档你可以使用，比如在交互模式下，可以使用 `help()` 命令来查看 Python 内置模块的信息。

如果只是输入 `help()` 指令，会输出关于 `help()` 指令使用方法的简单信息：

```
>>> help()

Welcome to Python 3.7's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics". Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help>
```

在 `help()` 指令的交互模式当中，输入 `q` 即可退出并返回到普通的 Python 交互模式。

如果要查看具体某个模块或者某个内置函数的文档，可以在 `help` 后面的圆括号当中填入要查看的名字，例如：

```
>>> help(math)
```

这回进入文档阅读模式，默认的阅读模式用的是类似 vim 的键位，按 i 键向上滚动，按 j 键向下滚动，按 q 键退出：

```
NAME
  math
```

```
MODULE REFERENCE
  https://docs.python.org/3.7/library/math
```

```
The following documentation is automatically generated from the Python
source files. It may be incomplete, incorrect or include features that
are considered implementation detail and may vary between Python
implementations. When in doubt, consult the module reference at the
```

```
:
```

除了上述两种方式之外，还可以直接输出对象的 `_doc_` 属性来查看这个对象的文档信息，例如我们查看数学模块当中求平方根的函数的文档：

```
>>> from math import sqrt
>>> print(sqrt._doc_)
Return the square root of x.
>>>
```

第四章 综合练习（二）

0.4.1 习题 1: 使用命令行参数

问题：在命令行当中运行 `python` 程序的时候，如何读取命令行参数？

`python` 程序最常用的一种运行方式，就是在命令行当中运行：

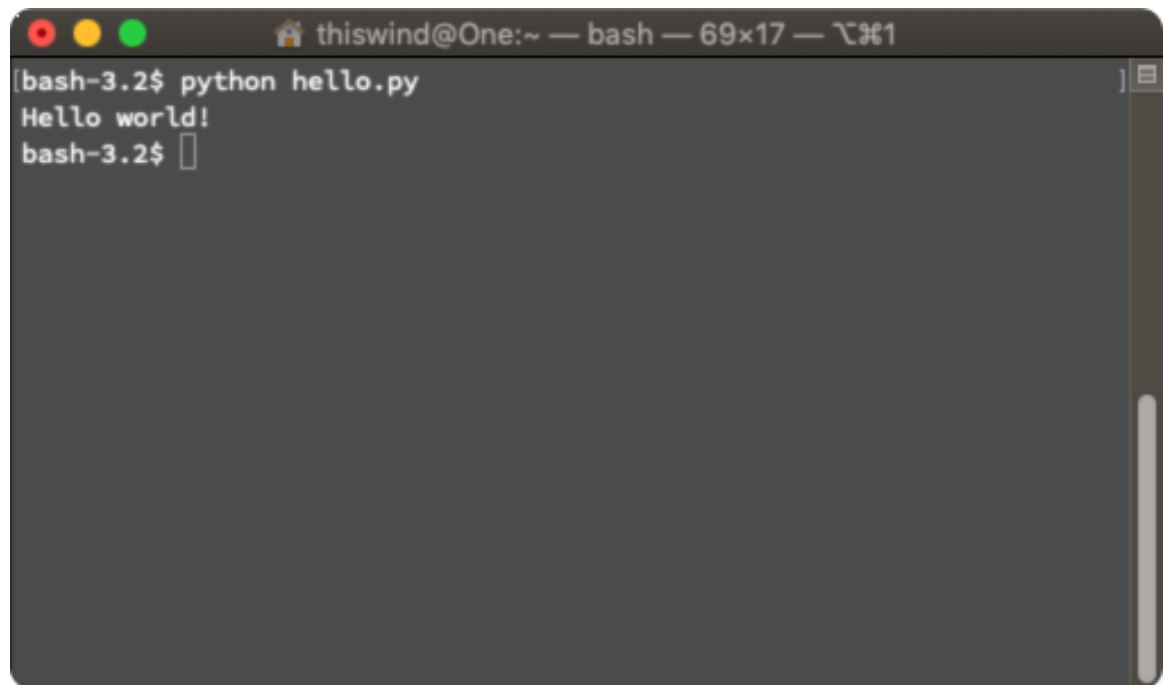
```
python 文件名.py
```

第一个 `python` 是解释器，文件名.py 是你要执行 `python` 源文件的名字

如果有一个 `python` 源文件 `hello.py`:

```
# python.py  
print('Hello world')
```

那么可以在命令行当中运行这个程序：

A terminal window with a dark background and light text. The title bar at the top shows a home icon, the text 'thiswind@One:~', a window icon, 'bash', a separator, '69x17', another separator, and a keyboard icon. The terminal content shows a prompt '[bash-3.2\$' followed by the command 'python hello.py'. The output is 'Hello world!'. The prompt returns to '[bash-3.2\$' with a cursor. There is a scrollbar on the right side of the terminal window.

```
thiswind@One:~ — bash — 69x17 — ￼1
[bash-3.2$ python hello.py
Hello world!
bash-3.2$
```

Python 程序的命令行参数被保存在 `sys.argv` 变量当中

```
# hello.py
import sys

for arg in sys.argv:
    print(f'arg is {arg}')
```

运行结果:

```
$ python hello.py
~/L/P/P/scratches
arg is hello.py
```

可以看到，当没有任何命令行参数的时候，`sys.argv` 当中只有一个元素，就是当前被执行的 `python` 文件的文件名。

如果有多个命令行参数，那么就可以得到:


```
$ python hello.py aaa bbb ccc ddd  
arg is hello.py      # sys.argv[0]  
arg is aaa           # sys.argv[1]  
arg is bbb           # sys.argv[2]  
arg is ccc           # sys.argv[3]  
arg is ddd           # sys.argv[4]
```

0.4.2 习题 2：在命令行直接执行 Python 语句

问题：如何在命令行直接执行 Python 语句

python 解释器有一个 `-c` 参数，接收一个字符串，然后 python 解释器就会解释执行这个字符串。例如：

```
$ python -c "print('hello world')"  
hello world  
$ python -c "a = 3 * 4; print(f'a is {a}')"  
a is 12
```

0.4.3 习题 3：接收命令行输入

问题：从命令行接收用户输入的一个字符串，并重新从命令行输出出来。

我们先新建一个 Python 文件：echo.py:

```
# echo.py  
  
message = input('Please input:')  
print(f'Got input from user "{message}"')
```

接下来我们在命令行当中运行这个程序：

```
$ python echo.py  
Please input:Hello  
Got input from user "Hello"
```

通过内置的 `input()` 函数可以接收用户在命令行当中的输入，可以传递一个字符串参数，作为用户输入的提示。`input()` 函数的返回值是一个字符串，如果需要接收的数字或者其它类型，需要在收到用户输入之后再进行必要的类型转换，例如：

```
# grade.py  
  
yeaar = int(input('您的入学年份: '))  
print(f'今年是 2019 年，您现在是{2019-year}年级')
```

运行结果：

```
$ python grade.py
您的入学年份: 2017
今年是 2019 年, 您现在是 2 年级
```

如果结合 `while` 循环语句, 我们就可以利用 `input()` 函数编写一个交互式的程序, 例如:

```
# hello.py

running = True

while running:
    message = input('your message:')

    if message == 'hello':
        print('greetings :)')
    else:
        print(f'got your message: {message}')

    if message == 'exit':
        running = False # 把循环条件设为 False, 退出循环
```

执行结果如下:

```
$ python hello.py
your message:AAA
got your message: AAA
your message:你好
got your message: 你好
your message:hello
greetings :)
your message:
got your message:
your message:exit
got your message: exit
bash-3.2$
```

0.4.4 习题 4：命令行交互

问题：编写一个备忘录，用户可以输入一些信息，可以通过关键词对用户输入的信息进行查找

解决这个问题，我们需要利用之前介绍的编写命令行交互式程序的技术。在这个问题当中，用户的输入有两种类型，一种是要存入的信息，另一种是检索用的关键字，这意味着存在两种状态，一种记忆状态，一种检索状态。但是为了程序的完整性，我们还需要设置第三种状态用以退出程序。

这个程序有很多种不同的写法，下面是其中一种写法的示例：

```
# memo.py

memory = []

running = True
while running:
    mode = input('M:存入信息,S:查找信息,Q:退出?')

    if mode == 'Q':
        running = False
        print('再见')

    elif mode == 'M':
        one_memo = input('请输入记录:')
        memory.append(one_memo)

    elif mode == 'S':
        keyword = input('关键词:')

        found = []
        for one_memo in memory:
            if keyword in one_memo:
                found.append(one_memo)
```

```
if found:
    print('找到记录:')
    for one in found:
        print(one)
else:
    print('没有找到')

else:
    print('输入错误')
```

运行过程:

```
$ python memo.py
M:存入信息,S:查找信息,Q:退出?M
请输入记录:下班记得买灭蚊器
M:存入信息,S:查找信息,Q:退出?M
请输入记录:下班记得要关灯
M:存入信息,S:查找信息,Q:退出?M
请输入记录:上班记得要开窗户
M:存入信息,S:查找信息,Q:退出?M
请输入记录:小王的电话号码是 12345678
M:存入信息,S:查找信息,Q:退出?S
关键词:下班
找到记录:
下班记得买灭蚊器
下班记得要关灯
M:存入信息,S:查找信息,Q:退出?S
关键词:小王
找到记录:
小王的电话号码是 12345678
M:存入信息,S:查找信息,Q:退出?Q
再见
```

0.4.5 习题 5：访问网页

问题：访问一个网页，并且输出网页的源代码

在 python 当中有很多中访问网址的方式，例如我们可以使用 requests 包：

```
import requests

r = requests.get('http://www.ynu.edu.cn') # 用 GET 方式访问网址
print(r.text)                             # 输出网页源代码
```

要运行这段程序，需要安装 requests 包，你可以打开一个命令行，通过 pip 安装 python 的扩展包：

```
pip install --user requests
```

运行结果：

```
.....
<META content="text/html; charset=UTF-8" http-equiv="Content-Type">
<LINK rel="stylesheet" type="text/css" href="css/test.css">
<LINK rel="stylesheet" type="text/css" href="css/style.css">
<script src="js/jquery-latest.min.js"></script><script src="js/TouchSlide.js"></scri
pt>
<script type="text/javascript" src="js/koala.min.js"></script>
.....
```

如果要从网页当中搜索包含某个关键词的行，可以像这样：

```
import requests
```

```
keyword = '云南大学'
```

```
r = requests.get('http://www.ynu.edu.cn')
```

```
r.encoding = 'utf-8' # 设置字符集为 UTF-8
```

```
lines = (line for line in r.text.split('\n')) # 用换行符切分, 返回一个生成器
```

```
for one_line in lines:
```

```
    if keyword in one_line:
```

```
        print(one_line)
```

输出可能会是这样:

```
<li><a href="http://www.news.ynu.edu.cn/info/1095/25396.htm" target="_blank">云南大学教职工第九届“东陆杯”气排球比赛举行闭...</a> <span>2019-05-06</span></li>
```

```
<li><a href="http://www.news.ynu.edu.cn/info/1109/25395.htm" target="_blank">数学与统计学院举办 2019 年云南大学第十四届女生节</a> <span>2019-05-06</span></li>
```

```
<li><a href="http://www.news.ynu.edu.cn/info/1095/25392.htm" target="_blank">云南大学召开 2019 年上半年组织工作会议暨基层党建...</a> <span>2019-05-05</span></li>
```

```
<li><a href="http://www.news.ynu.edu.cn/info/1103/25390.htm" target="_blank">云南大学王文光教授主持的国家社会科学基金重大项...</a> <span>2019-04-30</span></li>
```

```
<li><a href="http://www.news.ynu.edu.cn/info/1093/25389.htm" target="_blank">云南大学师生积极收听收看纪念五四运动 100 周年大...</a> <span>2019-04-30</span></li>
```

```
<li><a href="http://www.news.ynu.edu.cn/info/1095/25391.htm" target="_blank">云南大学团委组织青年师生集中收听收看纪念五四运...</a> <span>2019-04-30</span></li>
```


第五章 语法规则

本章将介绍 Python 程序的语法规则和一些约定。

0.5.1 代码结构和缩进

在 Python 程序当中，每一条语句默认以行尾的换行符作为语句的结束标志，这不同于 C、Java、等语言当中用英文的分号;表示语句结束。所以一般情况下，在 Python 当中，一行只写一条语句。例如：

```
# 每句一行
a = 1
b = 2
print(f'a + b = {a + b}')
```

但是这并不是强制的，在一些情况下需要把语句写到多行当中也是可以的，可以使用英文的反斜杠\作为续行符：

```
a = '这是一个比较长写在 \
多行的语句'

print(f'a = {a}')
```

输出：

```
a = 这是一个比较长写在多行的语句
```

缩进用来表示不同的代码块，比如函数、条件语句、循环体、类型定义的主体等等。代码块的下一级在上一级的基础上继续缩进，缩进量可以是任意的，但是整个代码块的缩进必须保持一致。例如：

```
if condition:
    statement_1 # 缩进一致，正确
    statement_2
```

```
else:
    statment_3
    statment_4 # 缩进不一致，错误
```

对于函数、条件语句、循环体，如果只包含一条语句，那么也可以直接放在统一行里面，例如：

```
if conditiona: statment_1
else: statment_2
```

如果要表示一个空的代码块，可以用 `pass` 语句：

```
if condition_a:
    pass
else:
    statment
```

如果要在统一行上写多条语句，那么也可以使用英文的分号作为语句的分隔符；，例如：

```
if condition_a: pass; else: statement;
```

最后，程序当中并不是每一行都是需要执行的代码，还需要一些注释来解释程序代码。在 `Python` 当中只有一种注释的格式，就是用英文井号 `#` 进行行注释，例如：

```
# 这是一个注释

a = 100 # a 的值是 100
```

即在井号后面的内容，直到这一行的结束，都是注释的内容，不会被执行。

0.5.2 标识符和保留字

我们在 Python 当中所使用的名字，比如变量名、函数名、类型名、模块名等，正式的叫法是“标识符”，在 Python 当中，标识符的可以包含字母、数字、汉字、除了下划线之外不能包含其它标点符号。字母是区分大小写的。例如下面这些都是合法的标识符：

```
姓名 = '张同学'  
_name = '李同学'  
name3 = '王同学'  
name_4 = '赵同学'  
  
print(f'姓名 is {姓名}')  
print(f'_name is {_name}')
```

输出：

```
姓名 is 张同学  
_name is 李同学  
name3 is 王同学  
name_4 is 赵同学
```

但是，并不是所有的单词都可以用作标识符，有一些特殊的单词是被 Python 所使用，有特殊含义的，这些单词被叫做保留字。可以通过 keyword 模块查看 Python 当中的关键字：

```
import keyword  
print(keyword.kwlist)
```

输出：

['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

在 Python 的约定当中，以英文下划线开头的标识符有特殊的含义：

以一个下划线开头的标识符，例如 `_name`，只应该在本模块内部使用。前后均有两个英文下划线的标识符，例如 `__init__`，是为特殊的方法保留的，只应在类型定义内部使用。前面有两个英文下划线而后面没有的标识符，例如 `_name`，是为类型定义当中的私有成员保留的，只应在类型定义内部使用。

0.5.3 数字的字面量

对于一个数字来说，除了数字的标识符的名字之外，还有数字的字面值。在 Python 当中，数字的字面值有四种类型：

- 布尔值
- 整数
- 浮点数
- 复数

布尔值只有 True 和 False 两种取值。

```
a = True  
b = False
```

在 Python 当中的整数是不受位数限制的，你可以指定任何长度的整数。这一点不同于 C 或者 Java，使用 Python 的时候你不需要为整数的精度操心（数字最大能写多少位）

对于浮点数，在 Python 当中一般会采用科学计数法进行表示，例如：

```
a = 12345.6789  
b = 1.23456789e+4  
  
print(f'a is {a}')  
print(f'b is {b}')  
print(f'a == b is {a == b}')
```

输出：

```
a is 12345.6789  
b is 12345.6789  
a == b is True
```

对于复数，在数字后面加上一个大写或者小写的字母 j，就可以把这个数字指定为复数的虚部。例如：

```
a = 3 + 4j  
  
real = a.real # 实部  
imag = a.imag # 虚部  
r = abs(a)    # 模  
  
print(f'real is {real}')  
print(f'imag is {imag}')  
print(f'r is {r}')
```

输出：

```
real is 3.0  
imag is 4.0  
r is 5.0
```


0.5.4 字符串的字面值

字符串的字面值的表示方式，是把文本放在英文的单引号'、双引号"、三引号'''或"""当中。三种方式没有差别，但是要求两端用相同的引号。

需要注意的是，在 **Python** 的字符串当中用英文的反斜杠\用于转义一些特殊的字符，例如换行符、反斜杠自身、引号以及其它非打印字符。

| 转义字符 | 描述 |
|---------------------|---|
| <code>\</code> | 在行尾时，续行符 |
| <code>\\</code> | 反斜杠符号 |
| <code>\'</code> | 单引号 |
| <code>\“</code> | 双引号 |
| <code>\a</code> | 响铃 |
| <code>\b</code> | 退格(Backspace) |
| <code>\e</code> | 转义 |
| <code>\000</code> | 空 |
| <code>\n</code> | 换行 |
| <code>\v</code> | 纵向制表符 |
| <code>\t</code> | 横向制表符 |
| <code>\r</code> | 回车 |
| <code>\f</code> | 换页 |
| <code>\oyy</code> | 八进制数，yy代表字符，例如： <code>\o12</code> 代表换行 |
| <code>\xyy</code> | 十六进制数，yy代表字符，例如： <code>\x0a</code> 代表换行 |
| <code>\other</code> | 其它的字符以普通格式输出 |

另外，可以在字符串前面加上小写字母 **r** 表示这是原始字符串，对于原始字符串，不会进行转义，例如：

```
a = 'hell\to'    # 转义制表符
b = r'hello\to'  # 原始字符串

print(f'a is {a}')
print(f'b is {b}')
```

输出：

```
a is hell o
b is hello\to
```

0.5.5 容器

在 Python 当中，方括号、圆括号和打括号括起来的内容，分别表示列表、元组、字典，这样的结构也可以叫做容器：

```
a = [1, 2, 3]
b = (4, 5, 6)
c = {'x': 7, 'y': 8, 'z': 10}
```

容器里每一个元素的末尾都需要跟一个英文逗号分隔。在创建容器的时候如果需要换行，可以不使用换行符。例如：

```
a = [
    1,
    2,
    3
]
```

0.5.6 运算符、分隔符和特殊符号

Python 的运算符全部都是英文字符：

```
+ - * ** / // % << >> & |  
^ ~ < > <= >= == != <> +=  
-= *= /= //= %= **= &= |= ^= >>= <<=
```

表达式、列表、字典以及语句当中可以使用如下分隔符，同样，都必须是英文字符：

```
( ) [ ] { } , : . ` = ;
```

另外，在 Python 语句当中也可能会出现下面这些英文符号：

```
' " # \ @
```

最后，英文的美元符号\$和问号?只能用在带引号的字符串字面值当中。

0.5.7 源代码的编码

Python 的源代码默认使用 UTF-8 字符集，一般来说，可以在 Python 源文件案的第一行或者第二行添加一行特殊的注释来标示源代码文件的编码格式：

```
#!/usr/bin/env python  
# -*- coding: UTF-8 -*-
```

在这行特殊的注释当中，只需要包含 `coding:`后面跟着字符集的名称，即可用来标示源代码的编码字符集。

第六章 函数

函数是为了便于维护和更好地实现程序的模块化。程序代码被分解为多个可以被重复利用的函数，可以把相同的定义集中在一处，使得代码的开发和维护都更加方便。相比起 C 和 Java 这样的语言，Python 对函数定义的语法做了极大的简化。

0.6.1 函数

使用 `def` 语句可以定义一个函数，例如：

```
def add(a, b):  
    return a + b # 两个数相加
```

第一行 `def add(a, b)` 称为函数声明，在这个例子当中，声明了一个名叫 `add` 的函数，同时声明了这个函数所接收的参数列表。

函数声明下面缩进的部分，是函数的函数体，函数体里面的语句，定义了这个函数将进行怎样的操作。

在这个例子当中，函数体的最后一句是一个 `return` 语句，表示这个函数将执行结束之后将会返回一个值。在函数的定义当中 `return` 语句并不是必须的，如果函数执行结束之后不需要返回某个值，就不需要写 `return` 语句。

在调用函数的时候，只需要在函数名后面用圆括号把参数传递进去即可，例如：

```
定义函数  
def add(a, b):  
    c = a + b # 两个数相加  
    return c  
  
y = add(4, 5) # 调用这个函数，传递参数 a:3, b:4  
  
print(f'y is {y}')
```

输出：


```
y is 9
```

在声明函数的参数列表的时候，函数的参数可以带有默认值，例如：

```
def add(a=10, b=20):  
    c = a + b  
    return c  
  
y = add(a=15) # b 取默认值 20  
  
print(f'y is {y}')
```

输出：

```
y is 35
```

如果在函数声明当中加上英文的星号*，就可以在函数当中传入任意数量的参数，例如：

```
def add(*numbers):  
    sum = 0  
  
    for x in numbers: # 参数会被放到列表当中  
        sum = sum + x  
  
    return sum  
  
# 用随意数量的参数调用  
y = add(1, 2, 3, 4, 5, 6, 7, 8, 9)  
  
print(f'y is {y}')
```

输出：

y is 45

0.6.2 参数传递与返回值

调用函数的时候，除了要指明函数名之外，还需要指明要传入的参数，或者参数的名称。

如果传递的参数是不可变对象，那么在函数内部对参数的修改不会影响原对象的值，例如：

```
a = 10

def foo(x):
    x = x + 100  # 修改参数的值
    return x

## 调用
b = foo(a)

print(f'a is {a}')
print(f'b is {b}')
```

输出：

```
a is 10
b is 110
```

我们看到，函数调用之后，参数原先的值并没有发生变化。

但是对于可变对象，在函数内部对参数的修改，就会影响到原对象本身：

```
a = [1, 2, 3]

def foo(x):
```

```
x.append(100)
return x

# 调用
b = foo(a)

print(f'a is {a}')
print(f'b is {b}')
```

输出：

```
a is [1, 2, 3, 100]
b is [1, 2, 3, 100]
```

一般来说，不推荐在函数当中对参数的原始值进行修改，随着程序的规模和复杂度的增加，这样的修改会给代码的维护和调试带来越来越多的困难。

在函数定义当中 **return** 语句从函数返回一个值，在前面的例子介绍过，这个值可以是一个普通的数字、字符串、布尔值，也可以是一个列表、元组、字典。如果返回的是一个元组，在调用的时候 Python 会自动解包：

```
def divide(a, b):
    q = a // b
    r = a % b
    return (q, r)    # 返回一个元组

# 调用的时候可以自解包
x, y = divide(10, 3) # 自解包

print(f'x is {x}')
print(f'y is {y}')
```

输出:

```
x is 3  
y is 1
```

0.6.3 作用域

在函数体里面定义的变量，其作用域只在函数体内部，这样的变量称为局部变量，例如我们在函数体内部也定义一个同名的变量 **a**，虽然同名，但是函数体内部的局部变量 **a** 和函数外部的全局变量 **a** 是两个不同的变量，对局部变量的任何操作，都不会影响到外部的全局变量。

```
a = 1

def foo():
    a = 1000 # 修改的只是局部变量，只是名字相同而已
    print(f'a in foo is {a}')

foo() # 调用函数
print(f'a out of foo is {a}')
```

输出：

```
a in foo is 1000
a out of foo is 1
```

但是相反，在函数内部可以直接使用在函数体外部定义的全局变量，例如：

```
a = 1

def foo():
    print(f'a in foo is {a}')

foo() # 调用函数
```

```
a = 500 # 修改全局变量的值
foo()
```

输出：

```
a in foo is 1
a in foo is 500
```

可以看到，全局变量的修改在函数内部同样是生效的。在 Python 当中，上一级的代码块里定义的名字的作用域可以延伸到下一级，如果下一级的代码块里有同名的名字，那么会在局部覆盖掉全局变量。但是相反，在下一级代码块里的所有名字，作用域都是无法超出本代码块的。

如果一定要在函数体内部操作函数外部定义的变量，那么需要用 `global` 关键字声明一个同名的变量，这样在函数体当中就可以直接对作用域外部的全局变量进行操作，例如：

```
a = 1

def foo():
    global a # 用 global 关键字声明一个同名的变量
    a = 100

print(f'before call: a is {a}')

foo()
print(f'after call: a is {a}')
```

输出：

```
before call: a is 1  
after call: a is 100
```

过多地使用全局变量，在程序的规模和复杂度增加之后，会造成很多调试和维护上的困难，如果不是必须的，尽量不要使用全局变量

。

第七章 面向对象编程

随着编程语言的更新发展，越来越多的编程方式被提出用来改善和解决编程效率、对问题的建模方式等问题。例如早期的面向过程编程方式、之后出现的面向对象编程方式以及现在新出现的函数式编程方式。在这些编程方式中，最流行以及应用的最广泛的的就是面向对象编程方式。

面向对象就是直接对现实世界中的问题通过一个一个类进行建模的，一个类即表示一类实体，包括各种各样的属性(内部数据)以及可以执行的操作(行为)，类之间通过继承或者组合来形成新的类，从而实现对现实问题比较直观的建模。例如人(Person)具有一些属性如姓名、性别、年龄，还有一些行为例如睡觉、吃饭，公司里面的员工(Employee)同样属于人这一类别，因此可以员工可以继承人，在其基础上添加上属于员工特有的属性和操作，例如员工编号属性和工作行为。公司管理者(Employer)也属于员工，因此可以可以继承员工，然后添加专有的属性和行为。公司的正常运转是需要员工和管理者共同配合完成的，这在面向对象里面叫做组合。正如上面这个例子，通过面向对象的方式对实际问题通过类建模之后，相比面向过程式的建模来说就多了一层抽象层次，更容易分析和建模。

以下是一些面向对象的特性：

- 类: 用来描述具有相同的属性和方法的对象的集合。它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。

- 类变量：类变量在整个实例化的对象中是公用的。类变量定义在类中且在函数体之外。类变量通常不作为实例变量使用。
- 数据成员：类变量或者实例变量, 用于处理类及其实例对象的相关的数据。
- 方法重写：如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫方法的覆盖（**override**），也称为方法的重写。
- 局部变量：定义在方法中的变量，只作用于当前实例的类。
- 实例变量：在类的声明中，属性是用变量来表示的。这种变量就称为实例变量，是在类声明的内部但是在类的其他成员方法之外声明的。
- 继承：即一个派生类（**derived class**）继承基类（**base class**）的字段和方法。继承也允许把一个派生类的对象作为一个基类对象对待。例如，有这样一个设计：一个 **Dog** 类型的对象派生自 **Animal** 类，这是模拟“是一个（**is-a**）”关系（例图，**Dog** 是一个 **Animal**）。
- 实例化：创建一个类的实例，类的具体对象。
- 方法：类中定义的函数。
- 对象：通过类定义的数据结构实例。对象包括两个数据成员（类变量和实例变量）和方法。

Python 从设计之初就已经是一门面向对象的语言，正因为如此，在 Python 中创建一个类和对象是很容易的。下面我们分别介绍如何创建类、类的实例对象、访问类的属性和方法、继承类以及在继承的类中重写方法实现不同的行为等内容。

0.7.1 创建类

在 Python 中使用 `class` 关键字来创建一个类，`class` 之后为类的名称并以冒号结尾，类的名称一般使用首字母大写的驼峰(CamelCase)命名方式，例如 `Dog`、`AdvancedManager` 等：

```
class ClassName:
    '类的帮助信息' #类文档字符串
    class_suite    #类体
```

类的帮助信息可以通过 `ClassName.__doc__` 来查看，`class_suite` 由类成员，方法组成。以下是一个简单的 Python 类的例子：

```
#!/usr/bin/python
# coding=utf-8

class Employee:
    '所有员工的基类'
    empCount = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1

    def displayCount(self):
        print(f"Total Employee {Employee.empCount}")

    def displayEmployee(self):
        print(f"Name : {self.name}, Salary: {self.salary}")
```

其中 `empCount` 变量是一个类成员变量（Class Attributes），它的值将在这个类的所有实例之间共享。你可以在内部类或外部类使用 `Employee.empCount` 访问；`__init__()` 方法是一种特殊的方法，被称为类的

构造函数或初始化方法，当创建了这个类的实例时就会调用该方法，需要注意的是所有的实例成员变量都必须在这个方法中声明或者初始化；**self** 代表类的实例，**self** 在定义类的方法时是必须有的，但在调用时不需要传入这个参数，Python 会自动把当前对象的实例作为第一个参数传递进去。

类的方法与普通的函数只有一个区别——类方法必须有一个额外的**第一个参数**，按照惯例它的名称是 **self**，表示当前类的实例。

```
class Test:
    def prt(self):
        print(self)
        print(self.__class__)

t = Test()
t.prt()
```

以上实例执行结果为：

```
<__main__.Test instance at 0x10d066878>
__main__.Test
```

0.7.2 创建实例对象

实例化类在其他编程语言如 Java、C++ 中一般用关键字 `new`，但是在 Python 中并没有这个关键字，类的实例化类似函数调用方式。以下使用 `Employee` 类来实例化两个此类的对象。

```
"创建 Employee 类的第一个对象"  
emp1 = Employee("Zara", 2000)  
"创建 Employee 类的第二个对象"  
emp2 = Employee("Manni", 5000)
```

0.7.3 访问类成员

您可以使用点号来访问对象的属性。如下所示，通过类的实例名来访问实例成员以及通过类名来访问类成员。

```
emp1.displayEmployee()  
emp2.displayEmployee()  
print(f"Total Employee {Employee.empCount}")
```

完整实例：

```
#!/usr/bin/python  
# coding=utf-8  
  
class Employee:  
    '所有员工的基类'  
    empCount = 0  
  
    def __init__(self, name, salary):  
        self.name = name  
        self.salary = salary  
        Employee.empCount += 1  
  
    def displayCount(self):  
        print(f"Total Employee {Employee.empCount}")  
  
    def displayEmployee(self):  
        print(f"Name : {self.name}, Salary: {self.salary}")  
  
"创建 Employee 类的第一个对象"  
emp1 = Employee("Zara", 2000)  
"创建 Employee 类的第二个对象"  
emp2 = Employee("Manni", 5000)  
emp1.displayEmployee()  
emp2.displayEmployee()  
print(f"Total Employee {Employee.empCount}")
```

执行以上代码输出结果如下：

```
Name : Zara ,Salary: 2000  
Name : Manni ,Salary: 5000  
Total Employee 2
```

你可以添加，删除，修改类的属性，如下所示：

```
emp1.age = 7 # 添加一个 'age' 属性  
emp1.age = 8 # 修改 'age' 属性  
del emp1.age # 删除 'age' 属性
```

你也可以使用以下函数的方式来访问属性：

- `getattr(obj, name[, default])` : 访问对象的属性。
- `hasattr(obj,name)` : 检查是否存在一个属性。
- `setattr(obj,name,value)` : 设置一个属性。如果属性不存在，会创建一个新属性。
- `delattr(obj, name)` : 删除属性。

```
hasattr(emp1, 'age') # 如果存在 'age' 属性返回 True。  
getattr(emp1, 'age') # 返回 'age' 属性的值  
setattr(emp1, 'age', 8) # 添加属性 'age' 值为 8  
delattr(emp1, 'age') # 删除属性 'age'
```


0.7.4 类的继承

面向对象的编程带来的主要好处之一是代码的重用，实现这种重用的方法之一是通过继承机制。通过继承创建的新类称为子类或派生类，被继承的类称为基类、父类或超类。

继承语法

```
class 派生类名(基类名)
...
```

在 python 中继承中的一些特点：

1. 如果在子类中需要父类的构造方法就需要显示的调用父类的构造方法，或者不重写父类的构造方法。详细说明可查看：[python 子类继承父类构造函数说明](#)。
2. 在调用基类的方法时，需要加上基类的类名前缀，且需要带上 `self` 参数变量。区别在于类中调用普通函数时并不需要带上 `self` 参数
3. Python 总是首先查找对应类型的方法，如果它不能在派生类中找到对应的方法，它才开始到基类中逐个查找。（先在本类中查找调用的方法，找不到才去基类中找）。

如果在继承元组中列了一个以上的类，那么它就被称作“多重继承”。派生类的声明，与他们的父类类似，继承的基类列表跟在类名之后，如下所示：

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
...

#!/usr/bin/python
# coding=utf-8

class Parent:    # 定义父类
```

```

parentAttr = 100

def __init__(self):
    print("调用父类构造函数")

def parentMethod(self):
    print('调用父类方法')

def setAttr(self, attr):
    Parent.parentAttr = attr

def getAttr(self):
    print(f"父类属性 : {Parent.parentAttr}")

class Child(Parent): # 定义子类
    def __init__(self):
        print("调用子类构造方法")

    def childMethod(self):
        print('调用子类方法')

c = Child()      # 实例化子类
c.childMethod()  # 调用子类的方法
c.parentMethod() # 调用父类方法
c.setAttr(200)   # 再次调用父类的方法 - 设置属性值
c.getAttr()      # 再次调用父类的方法 - 获取属性值

```

以上代码执行结果如下：

```

调用子类构造方法
调用子类方法
调用父类方法
父类属性 : 200

```

你可以继承多个类

```
class A:    # 定义类 A
.....

class B:    # 定义类 B
.....

class C(A, B): # 继承类 A 和 B
.....
```

你可以使用 `issubclass` 或者 `isinstance` 方法来检测。

- `issubclass(sub,sup)` - 判断一个类是另一个类的子类或者子孙类。
- `isinstance(obj, Class)` 如果 `obj` 是 `Class` 类的实例对象或者是一个 `Class` 子类的实例对象则返回 `true`。

0.7.5 方法重写

如果父类的方法不满足业务逻辑，可以在子类中进行重写（Overriding）此方法，添加上自己专有的业务逻辑，通常情况下还会通过 `super()` 调用父类的此方法。

实例：

```
#!/usr/bin/python
# coding=utf-8

class Rectangle:
    def __init__(self, length, breadth):
        self.length = length
        self.breadth = breadth

    def area(self):
        return self.length * self.breadth

    def printArea(self):
        print(f"{self.area()} is area of rectangle")

class Square(Rectangle):
    def __init__(self, side):
        Rectangle.__init__(self, side, side)

    def printArea(self):
        print(f"{self.area()} is area of square")

s = Square(4)
r = Rectangle(2, 4)
s.printArea()
r.printArea()
```

执行以上代码输出结果如下：

16 is area of square
8 is area of rectangle

0.7.6 类方法和静态方法

类方法（class method）通常是用@classmethod 装饰器定义的方法，和实例方法（instance method）类似，同样有第一个隐含的参数，一般命名为 cls，表示的类。此类方法一般是通过类名来调用的。

类方法（static method）通常是用@staticmethod 装饰器定义的方法，和类方法（instance method）类似，一般是通过类名来调用的。但是没有第一个隐含的参数。

如下示例所示：

```
#!/usr/bin/python
# coding=utf-8

from datetime import date

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # 通过 name 和 year 创建实例的类方法
    @classmethod
    def fromBirthYear(cls, name, year):
        return cls(name, date.today().year - year)

    # 判断一个人是否成年的静态方法
    @staticmethod
    def isAdult(age):
        return age > 18

person1 = Person('mayank', 21)
person2 = Person.fromBirthYear('mayank', 1996)

print(person1.age)
```

```
print(person2.age)
print(Person.isAdult(22))
```

运行结果是：

```
21
23
True
```

通常我们这样来区别使用类方法和静态方法：

- 通常使用类方法来创建工厂方法（**factory methods**），工厂方法在多种场景下返回这个类的实例，类似于构造函数。
- 通常使用静态方法来创建一些工具函数（**utility functions**）来辅助实现类的一些功能。

0.7.7 基础重载方法

上面的例子中当我们 `print` 输出一个对象的时候，通常显示的是 `<__main__.Employee object at 0x0000023BCCFC8438>` 这样的内容，这样很不人性化，只能看出它是一个 `Employee` 对象，其内存虚拟地址是 `0x0000023BCCFC8438`，这些信息实际上没有多少意义，如果可以显示其成员属性值就更有价值了。在 `Python` 中我们可以通过重载 `__str__(self)` 方法来实现此目的，添加的方法如下所示：

```
def __str__(self):
    return f'Name : {self.name}, Salary: {self.salary}'

def __repr__(self):
    return f'Employee('{self.name}','{self.salary}')
```

上面的代码中还有一个特殊的中 `__repr__(self)`，它和 `__str__(self)` 类似，`__repr__(self)` 侧重于生成一个没有歧义的描述信息，通常是一个可以直接构造此对象的代码字符串，在交互式环境中输出一个对象实例或显示调用 `repr` 会调用此方法；`__str__(self)` 则侧重于当使用 `print/str` 的时候需要一个可读性的字符串描述信息时会调用此方法，在没有 `__str__(self)` 方法的时候，`print/str` 会自动调用 `__repr__(self)` 作为替代。

以下是在交互式环境下面一个简单的例子：

```
>>> import datetime
>>> today = datetime.datetime.now()
>>> str(today)
'2019-04-06 18:17:27.946784'
```



```

>>> today
datetime.datetime(2019, 4, 6, 18, 17, 27, 946784)
>>> repr(today)
'datetime.datetime(2019, 4, 6, 18, 17, 27, 946784)'
>>> eval(repr(today))
datetime.datetime(2019, 4, 6, 18, 17, 27, 946784)
>>> anotherDate = eval(repr(today))
>>> anotherDate == today
True
>>>

```

从上面的代码可以看出 `repr` 返回的字符串是一个可执行的可以直接构建当前对象的可执行代码字符串。

下表列出了一些通用的功能，你可以在自己的类重写：

| 序号 | 方法，描述 & 简单的调用 |
|----|---|
| 1 | <code>__init__(self [,args...])</code> 构造函数 简单的调用方法： <code>obj = className(args)</code> |
| 2 | <code>__del__(self)</code> 析构方法，删除一个对象 简单的调用方法： <code>del obj</code> |
| 3 | <code>__repr__(self)</code> 转化为供解释器读取的形式 简单的调用方法： <code>repr(obj)</code> |
| 4 | <code>__str__(self)</code> 用于将值转化为适于人阅读的形式 简单的调用方法： <code>str(obj)</code> |
| 5 | <code>__cmp__(self, x)</code> 对象比较 简单的调用方法： <code>cmp(obj, x)</code> |

0.7.8 运算符重载

我们在学习列表的时候知道可以使用+合并两个列表，如下面的例子：

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

如果使用一个未定义的运算，例如-，或者在不同数据类型之间进行运算会有如下错误提示：

```
>>> a-b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'list' and 'list'
>>> 1+a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'list'
>>>
```

从上面的例子中可以知道内置的 list 数据类型实现了+运算，没有实现-运算。

如果是一个自定义的类型，例如在二维坐标系统中表示点的一个类，其定义可能如下：

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
```

```
p1 = Point(2, 3)
p2 = Point(-1, 2)
print(p1 + p2)
```

执行上面的代码同样会有类似的错误提示信息：

```
Traceback (most recent call last):
  File "Point.py", line 9, in <module>
    p1 + p2
TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

为了让上面的代码可以正确运行，我们需要在 `Point` 类中定义+运算，也就是去实现一个`__add__()`特殊的方法如下：

```
def __add__(self, other):
    x = self.x + other.x
    y = self.y + other.y
    return Point(x, y)
```

完整的代码如下：

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return f"({self.x},{self.y})"

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)
```

```
p1 = Point(2, 3)
p2 = Point(-1, 2)
print(p1 + p2)
```

运行结果如下：

```
(1,5)
```

除了+加法对应`_add_`，Python 中还有如下一些运算符可以重载

:

| 运算 | 表达式 | 内部调用 |
|-------------------------|-----------------------------|----------------------------------|
| 加(Addition) | <code>p1 + p2</code> | <code>p1.__add__(p2)</code> |
| 减(Subtraction) | <code>p1 - p2</code> | <code>p1.__sub__(p2)</code> |
| 乘(Multiplication) | <code>p1 * p2</code> | <code>p1.__mul__(p2)</code> |
| 幂(Power) | <code>p1 ** p2</code> | <code>p1.__pow__(p2)</code> |
| 除(Division) | <code>p1 / p2</code> | <code>p1.__truediv__(p2)</code> |
| 向下取整除法(Floor Division) | <code>p1 // p2</code> | <code>p1.__floordiv__(p2)</code> |
| 取余数(Remainder/modulo) | <code>p1 % p2</code> | <code>p1.__mod__(p2)</code> |
| 左移(Bitwise Left Shift) | <code>p1 << p2</code> | <code>p1.__lshift__(p2)</code> |
| 右移(Bitwise Right Shift) | <code>p1 >> p2</code> | <code>p1.__rshift__(p2)</code> |
| 按位与(Bitwise AND) | <code>p1 & p2</code> | <code>p1.__and__(p2)</code> |
| 按位或(Bitwise OR) | <code>p1 p2</code> | <code>p1.__or__(p2)</code> |
| 异或(Bitwise XOR) | <code>p1 ^ p2</code> | <code>p1.__xor__(p2)</code> |
| 非(Bitwise NOT) | <code>~p1</code> | <code>p1.__invert__()</code> |

类似的还可以重载关系运算符：

| 运算 | 表达式 | 内部调用 |
|--------------------------------|--------------------------|----------------------------|
| 小于(Less than) | <code>p1 < p2</code> | <code>p1.__lt__(p2)</code> |
| 小于等于(Less than or equal to) | <code>p1 <= p2</code> | <code>p1.__le__(p2)</code> |
| 等于(Equal to) | <code>p1 == p2</code> | <code>p1.__eq__(p2)</code> |
| 不等于(Not equal to) | <code>p1 != p2</code> | <code>p1.__ne__(p2)</code> |
| 大于(Greater than) | <code>p1 > p2</code> | <code>p1.__gt__(p2)</code> |
| 大于等于(Greater than or equal to) | <code>p1 >= p2</code> | <code>p1.__ge__(p2)</code> |

第八章 模块与包

模块(Module)与包(Package)是任何大新程序的核心，一个复杂的项目通常要使用到很多外部模块和内部模块，模块化开发也是现代软件开发的常用方式，通常一个模块的最佳实践是高内聚低耦合，减少不同模块之间的耦合性，这对开发、测试来说都是非常有帮助的。

随着项目的代码越来越多，功能也越来越复杂，不可能把所有的代码都放在一个源代码文件中，我们通常会分隔到不同的文件中，每个文件是一组完成特定任务的代码集合，可以包括相关变量、常量、函数、类等，这样一个文件在 Python 中也叫做模块。再往更高层次设计，有时候一些特定的模块可以组织起来来完成一个更高层面上的任务，这样一系列相关模块的组合在 Python 中就叫做包。

0.8.1 为什么使用模块与包

使用模块最大的好处是大大提高了代码的可维护性。其次，编写代码不必从零开始。当一个模块编写完毕，就可以被其他地方引用。我们在编写程序的时候，也经常引用其他模块，包括 Python 内置的模块和来自第三方的模块。使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中，因此，我们自己在编写模块时，不必考虑名字会与其他模块冲突。但是也要注意，尽量不要与内置函数名字冲突。

如果不同的人编写的模块名相同怎么办？为了避免模块名冲突，Python 又引入了按目录来组织模块的方法，称为包（Package）。

例如一个 `module1.py` 的文件就是一个名字叫 `module1` 的模块，一个 `module2.py` 的文件就是一个名字叫 `module2` 的模块。现在，假设我们的 `module1` 和 `module2` 这两个模块名字与其他地方的模块冲突了，于是我们可以通过包来组织模块，避免冲突。方法是选择一个顶层包名，比如 `mypackage`，按照如下目录存放：

```
mypackage
├─ __init__.py
├─ module1.py
└─ module2.py
```

引入了包以后，只要顶层的包名不与别人冲突，那所有模块都不会与别人冲突。现在，`module1.py` 模块的名字就变成了 `mypackage.module1`，类似的，`module2.py` 的模块名变成了 `mypackage.module2`。每一个包目录下面都会有一个 `__init__.py` 的文件，这个文件是必须存在

的，否则，Python 就把这个目录当成普通目录，而不是一个包。`__init__.py` 可以是空文件，也可以有 Python 代码，因为 `__init__.py` 本身就是一个模块，而它的模块名就是 `mypackage`。

一个包所对应的文件夹下可以有多级子目录，组成多级层次的包结构，比如如下的目录结构：

```
mypackage
├── web
│   ├── __init__.py
│   ├── utils.py
│   └── www.py
├── __init__.py
├── module1.py
└── module2.py
```

此时 `www.py` 的模块名就是 `mypackage.web.www`。

上面的包结构中，如果 `module1.py` 文件内容如下：

```
print(f"running in module: {_name_}")
PI = 3.1415926

def circleArea(radius):
    return PI * radius * radius
```

`module2.py` 文件内容如下：

```
from module1 import PI, circleArea

if __name__ == "__main__":
    print(f"running in module: {_name_}")
    print(f"PI: {PI}")
```



```
radius = 3
print(f"circleArea of radius {radius}: {circleArea(radius)}")
```

则运行 `python module2.py` 运行结果如下所示：

```
running in module: module1
running in module: __main__
PI: 3.1415926
circleArea of radius 3: 28.274333400000003
```

`module2.py` 中模块引用方式如果直接使用 `import` 的方式，则需要改写成如下：

```
import module1

if __name__ == "__main__":
    print(f"running in module: {__name__}")
    print(f"PI: {module1.PI}")
    radius = 3
    print(f"circleArea of radius {radius}: {module1.circleArea(radius)}")
```

模块属性 `__name__`，它的值由 Python 解释器设定，如果脚本文件是作为主程序调用，其值就设为 `__main__`，如果是作为模块被其他文件导入，它的值就是当前模块名即文件名。通过 `import moduleName` 方式引入模块的时候，这个模块下所有的变量、函数、类等都通过 `moduleName` 来访问，此时 `moduleName` 类似于命名空间，也可以理解为名字限定符。通过 `from moduleName import someVariable, someFunction` 方式引入模块的时候，即把需要用到的变量、函数、类引入到当前命名空间下，使用的时候不用加 `moduleName` 名字限定符。还可以通过 `from moduleName import *` 方式引入模块所有的变量、函数、类等

，但这种方式容易造成同名的变量、函数、类等被覆盖，造成不可预知的问题，不建议使用。

0.8.2 import 和 from 的使用

可以从包中导入模块或者模块里面的变量、函数、类等：

- `import PackageA.SubPackageA.ModuleA`，使用时必须用全路径名，例如要使用 `ModuleA` 中的某个函数 `func`，则需要通过 `PackageA.SubPackageA.ModuleA.func` 使用。
- `from PackageA.SubPackageA import ModuleA`，使用时直接使用模块名而不用加上包前缀。例如通过 `ModuleA.func` 来使用 `ModuleA` 中的某个函数 `func`。
- `from PackageA.SubPackageA.ModuleA import func`，也可以通过这种方式直接导入 `ModuleA` 模块中的 `func` 函数。

0.8.3 模块搜索路径

当导入一个模块时，解释器先在当前包中查找模块，若找不到，然后在内置的 `built-in` 模块中查找，找不到则按 `sys.path` 给定的路径找对应的模块文件(模块名.py)

`sys.path` 的初始值来自于以下地方：

1. 包含脚本当前的路径，当前路径
2. `PYTHONPATH` 环境变量
3. 默认安装路径

`sys.path` 初始化完成之后可以更改

上面的例子中，当 `module1` 被导入后，python 解释器就在当前目录下寻找 `module1.py` 的文件，然后再从环境变量 `PYTHONPATH` 寻找，如果这环境变量没有找到，解释器还会在安装预先设定的的一些目录寻找。

这些搜索目录可在运行时动态改变，比如将 `module1.py` 不放在当前目录，而放在其他地方。这里你就需要通过某种途径，如添加 `module1.py` 的路劲到 `sys.path` 中来告知 Python 从哪里来搜索模块。

`sys.path` 返回的是模块搜索列表，通过前后的输出对比和代码，应能理悟到如何增加新路径的方法了吧。非常简单，就是使用 list 的 `append()`或 `insert()`增加新的目录。

```

import sys
import os
from pprint import pprint

pprint(f"sys.path: size {len(sys.path)}, {sys.path}")
workpath = os.path.dirname(os.path.abspath(sys.argv[0]))
sys.path.insert(0, os.path.join(workpath, 'modules'))
pprint(f"sys.path: size {len(sys.path)}, {sys.path}")

```

运行上面的代码结果如下：

```

{'sys.path: size 9, '
 '['D:\\\\code\\\\python\\\\python-ebook\\\\code\\\\mymodule', "
 '"C:\\\\Users\\\\Liu.D.H\\\\Anaconda3\\\\python37.zip', "
 '"C:\\\\Users\\\\Liu.D.H\\\\Anaconda3\\\\DLLs', "
 '"C:\\\\Users\\\\Liu.D.H\\\\Anaconda3\\\\lib', "
 '"C:\\\\Users\\\\Liu.D.H\\\\Anaconda3', "
 '"C:\\\\Users\\\\Liu.D.H\\\\Anaconda3\\\\lib\\\\site-packages', "
 '"C:\\\\Users\\\\Liu.D.H\\\\Anaconda3\\\\lib\\\\site-packages\\\\win32', "
 '"C:\\\\Users\\\\Liu.D.H\\\\Anaconda3\\\\lib\\\\site-packages\\\\win32\\\\li
 b', "
 '"C:\\\\Users\\\\Liu.D.H\\\\Anaconda3\\\\lib\\\\site-packages\\\\Pythonwin'
 "]
 'sys.path: size 10, '
 '['D:\\\\code\\\\python\\\\python-ebook\\\\code\\\\mymodule\\\\modules',
 "
 '"D:\\\\code\\\\python\\\\python-ebook\\\\code\\\\mymodule', "
 '"C:\\\\Users\\\\Liu.D.H\\\\Anaconda3\\\\python37.zip', "
 '"C:\\\\Users\\\\Liu.D.H\\\\Anaconda3\\\\DLLs', "
 '"C:\\\\Users\\\\Liu.D.H\\\\Anaconda3\\\\lib', "
 '"C:\\\\Users\\\\Liu.D.H\\\\Anaconda3', "
 '"C:\\\\Users\\\\Liu.D.H\\\\Anaconda3\\\\lib\\\\site-packages', "
 '"C:\\\\Users\\\\Liu.D.H\\\\Anaconda3\\\\lib\\\\site-packages\\\\win32', "
 '"C:\\\\Users\\\\Liu.D.H\\\\Anaconda3\\\\lib\\\\site-packages\\\\win32\\\\li
 b', "
 '"C:\\\\Users\\\\Liu.D.H\\\\Anaconda3\\\\lib\\\\site-packages\\\\Pythonwin'
 "]

```

可以看出第二次输出的时候 `sys.path` 最前面添加了我们自定义的路劲，由于环境不同，每个人运行上述代码所得到的结果是不同的。

0.8.4 打包

上面提到的包其实是安装之后的的目录结构，通常我们使用的第三方包（库）是从 PyPI(Python Package Index 的缩写，<https://pypi.org/>)上下载下来安装的，一般下载的可以是 whl 编译之后的安装包或者源码安装包，这是 `pip install packageName` 时候自动判定的，优先下载 whl 的安装包，如果没有则下载源码然后本地编译安装。我们在本地把我们的包制作成 whl 或者源码压缩包的过程就叫做打包，打包生成的文件可以很方便的拷贝到其他电脑上安装。一般为了让所有 Python 开发者都可以使用，我们会上传到 pypi 上。

要实现打包，需要在前面讲到的包目录结构的基础上，在包目录的同级添加一个 `setup.py` 文件。`setup.py` 文件是程序包的构建脚本文件。Setuptools 的 `setup` 函数可构建要上传到 PyPI 的程序包。Setuptools 里有程序包的信息，版本号以及用户所需要的其他程序包的信息。

以下是一个示例的目录结构：

```
.
├── README.md
├── mymodule
│   ├── __init__.py
│   ├── module1.py
│   ├── module2.py
│   ├── module3.py
│   ├── module4.py
│   └── web
│       ├── __init__.py
│       ├── utils.py
│       └── www.py
├── requirements.txt
└── setup.py
```

其中 `setup.py` 文件的内容如下：

```
from setuptools import setup, find_packages

# 一般会有一个 README 文件描述这个包的信息
with open("README.md", "r") as readme_file:
    readme = readme_file.read()

# 自己写的包可能也会依赖于一些其他包
requirements = ["ipython>=6", "nbformat>=4", "nbconvert>=5", "requests>=2"]

setup(
    name="mymodule",
    version="0.0.5",
    author="donghua liu",
    author_email="liudonghua123@gmail.com",
    description="A package to show how to create",
    long_description=readme,
    long_description_content_type="text/markdown",
    url="https://github.com/your_package/homepage/",
    packages=find_packages(),
    install_requires=requirements,
    classifiers=[
        "Programming Language :: Python :: 3.7",
        "License :: OSI Approved :: GNU General Public License v3 (GPLv3)",
    ],
)
```

主要是配置一些包的属性信息，例如名称、版本号、作者、作者邮箱、描述、详细描述以及依赖库等信息，都比较直接。

在有上述目录结构之后，进入到 `setup.py` 所在目录，执行 `python setup.py sdist bdist_wheel` 就可以同时制作源码安装包和 `whl` 二进制安装包，打包之后的文件在 `dist` 子目录，其内容类似于：

```
dist/
├── mymodule-0.0.5-py3-none-any.whl
└── mymodule-0.0.5.tar.gz
```

打包之后我们就可以上传到 PyPI 上，这是通过 `twine`(通过 `pip install twine` 来安装)工具来上传的，最好我们先上传到 [Test PyPI](#) 上测试之后没有问题再正式发布到 PyPI 上，需要现在 PyPI 上注册一个用户才能执行上传操作。下面是上传的命令：

```
# 上传到 Test PyPI 为了测试
twine upload --repository-url https://test.pypi.org/legacy/ dist/*
# 从 Test PyPI 上进行安装测试
pip install --index-url https://test.pypi.org/simple/ --extra-index-url https://pypi.org/simple mymodule

# 测试没有问题之后正式上传到 PyPI 上
twine upload dist/*
```


第九章 GUI 图形界面

GUI 是 Graphical User Interface 的简称，即图形用户接口，准确来说 GUI 就是屏幕产品的视觉体验和互动操作部分。GUI 是一种结合计算机科学、美学、心理学、行为学，及各商业领域需求分析的人机系统工程，强调人一机—环境三者作为一个系统进行总体设计。

GUI 图形界面包含的领域非常广泛，可以是操作系统桌面应用程序、浏览器里的网页或者是手机、Pad 等移动设备上的应用程序，本章中介绍的 GUI 图形界面是在桌面操作系统上的图形界面程序设计。在 Python 中进行桌面图形界面程序开发非常方便，除了可以使用自带的 Tkinter 外，还可以使用很多第三方库如 PyQt5、PyGObject、wxPython、Pyside 等。几乎所有的这些 GUI 图形界面开发库都有一些共同的特性：基于消息队列事件循环，图形界面程序是通过事件来驱动图形界面的变化，内部有一个无限循环的消息队列，点击按钮、鼠标左右键、键盘输入等都会产生相对应的事件，事件包括事件源即发起事件的控件、事件类型例如鼠标左键点击、事件接收者例如在一个文本输入框中输入内容则事件接收者是文本输入框，产生的事件会加入到一个消息队列中，存在一些列消息处理函数（也称为 Handler）来不断地从消息队列中获取其能处理的事件然后处理；每个类库都提供一些相似外观的组件/控件（View、Widget、Component）以及用来对组件进行布局的容器（Container、Layout），组件和容器是图形界面设计中非常基础并且也非常重要的，为了便于组件和容器的方便、灵活、可扩展使用，一般使用面向对象的方式封装这些组件和容器，形成一

颗层次结构清晰的继承组件树，如果类库自带的组件和容器不满足实际用途要求，可以通过继承的方式实现自定义的组件和容器。

0.9.1 Tkinter

一个简单的 hello world 例子如下：

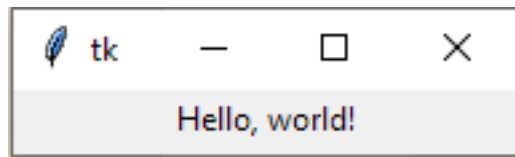
```
# 注意：Python3.x 版本使用的库名为tkinter (2.x 版本中库的名字为Tkinter)
from tkinter import *

root = Tk()

w = Label(root, text="Hello, world!")
w.pack()

root.mainloop()
```

运行后的效果如下图所示：



一开始我们执行 `from tkinter import *` 从 `tkinter` 包中导入所有的模块，这些模块有各种组件、容器以及相关函数，我们也可以仅仅导入需要用到的，这里也可以写成 `from tkinter import Tk, Label`。通过执行 `root = Tk()` 我们对 `tkinter` 进行初始化，我们需要创建一个 `Tk` 类型的根容器，它是一个普通的窗口组件，有标题栏，通过窗口管理器管理的最大最小化以及关闭按钮，每个 `tkinter` 应用程序都需要创建一个这样的顶层窗口容器，并且需要在创建其他子组件或子容器之前创建它。接着我们执行 `w = Label(root, text="Hello, world!")` 创建一个标签组件（`Label`），标签组件仅仅用来显示一些文本，可以附带一些图标（`icon`）或者图像（`image`），这里我们仅仅是显示一个“Hello, world!”字符串，需要注意第一个参数需要指定父容器，即把标签组件放在

什么地方。后面我们通过 `w.pack()` 调用标签组件的 `pack` 方法，这个方法是用来做布局调整的——设置他自己的大小适应标签组件的内容的大小，设置为可见状态。但是这时候整个窗口还没有显示出来，直到我们调用 `root.mainloop()` 让 `tkinter` 进入事件循环，这时才会显示应用程序的窗口以及其中的组件。

这个程序的窗口会一直显示着直到我们关闭窗口，因为事件循环是一个无限循环的消息处理队列。事件循环不仅处理来自用户的事件（例如鼠标点击、键盘按键点击等事件）、窗口系统的事件（例如拖动改变窗口尺寸之后由窗口管理系统触发的重绘事件），还会处理 `tkinter` 自身生成的事件。

`Tkinter` 的提供各种控件，如按钮，标签和文本框，目前有 15 种 `Tkinter` 的部件。我们提出这些部件以及一个简短的介绍，在下面的表：

| 控件 | 描述 |
|-----------------------------|--|
| Button | 按钮控件；在程序中显示按钮。 |
| Canvas | 画布控件；显示图形元素如线条或文本 |
| Checkbutton | 多选框控件；用于在程序中提供多项选择框 |
| Entry | 输入控件；用于显示简单的文本内容 |
| Frame | 框架控件；在屏幕上显示一个矩形区域，多用来作为容器 |
| Label | 标签控件；可以显示文本和位图 |
| Listbox | 列表框控件；在Listbox窗口小部件是用来显示一个字符串列表给用户 |
| Menubutton | 菜单按钮控件，由于显示菜单项。 |
| Menu | 菜单控件；显示菜单栏,下拉菜单和弹出菜单 |
| Message | 消息控件；用来显示多行文本，与label比较类似 |
| Radiobutton | 单选按钮控件；显示一个单选的按钮状态 |
| Scale | 范围控件；显示一个数值刻度，为输出限定范围的数字区间 |
| Scrollbar | 滚动条控件，当内容超过可视化区域时使用，如列表框。 |
| Text | 文本控件；用于显示多行文本 |
| Toplevel | 容器控件；用来提供一个单独的对话框，和Frame比较类似 |
| Spinbox | 输入控件；与Entry类似，但是可以指定输入范围值 |
| PanedWindow | PanedWindow是一个窗口布局管理的插件，可以包含一个或者多个子控件。 |
| LabelFrame | labelframe 是一个简单的容器控件。常用与复杂的窗口布局。 |
| tkMessageBox | 用于显示你应用程序的消息框。 |

在 tkinter 中有一些标准通用属性，例如大小、颜色、字体设置等，例如上面的例子中我们可以通过 `root.configure(cursor="dotbox green")`来设置整个窗口内的光标形状颜色或者通过 `w.configure(cursor="dotbox green")`来仅仅设置标签组件的光标。下面是这些标准通用属性：

| 属性 | 描述 |
|-----------|-------|
| Dimension | 控件大小； |
| Color | 控件颜色； |
| Font | 控件字体； |
| Anchor | 锚点； |
| Relief | 控件样式； |
| Bitmap | 位图； |
| Cursor | 光标； |

上面提到的 `pack` 方法是用作布局调整使用的，类似的还有 `grid`、`place`，他们也称作几何管理，`grid` 是网格布局，`place` 是相对来说用的较少的位置布局（可以设置相对或绝对位置）

以下是一个简单的 `grid` 布局的例子：



```
from tkinter import Tk, Button, messagebox
root = Tk()

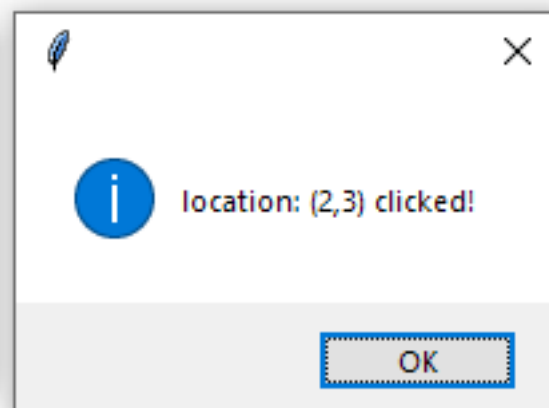
def handleClick(location):
    messagebox.showinfo(
        message=f"location: ({location[0]},{location[1]}) clicked!")

for r in range(3):
    for c in range(4):
        btn = Button(root, text=f"({r},{c})", command=lambda: handleClick(
            (r, c)), padx=5, pady=5)
        btn.grid(row=r, column=c)

root.mainloop()
```

运行后的效果是

| | | | |
|---|-------|---|-------|
|  | — |  | × |
| (0,0) | (0,1) | (0,2) | (0,3) |
| (1,0) | (1,1) | (1,2) | (1,3) |
| (2,0) | (2,1) | (2,2) | (2,3) |



0.9.2 PyQt5

PyQt 是一个创建 GUI 应用程序的工具包，它是 Qt 的 Python 比较流行的封装库，Qt 则是 C++ 领域非常著名的一个跨平台 GUI 应用程序的工具包，Linux 中 KDE 桌面就是通过 Qt 来开发的。PyQt 是由 Phil Thompson 开发，PyQt 实现了一个 Python 模块集。它有超过 300 类，将近 6000 个函数和方法。它是一个多平台的工具包，可以运行在所有主要操作系统上，包括 UNIX，Windows 和 Mac。目前 PyQt4 和 Qt4 已经停止更新维护了，建议使用最新版本的 PyQt5，其对应使用的是 Qt5。官方网址是 <https://www.riverbankcomputing.com/software/pyqt/intro>。

在使用 PyQt5 之前需要通过 pip 来安装相应的包：

```
pip3 install pyqt5 pyqt5-tools
```

下面是一个简单的 hello world 小例子：

```
import sys
from PyQt5 import QtCore, QtWidgets
from PyQt5.QtWidgets import QMainWindow, QLabel, QGridLayout, QWidget
from PyQt5.QtCore import QSize

class HelloWorld(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        self.setMinimumSize(QSize(320, 240))
        self.setWindowTitle("Hello world")

        centralWidget = QWidget(self)
```



```
self.setCentralWidget(centralWidget)

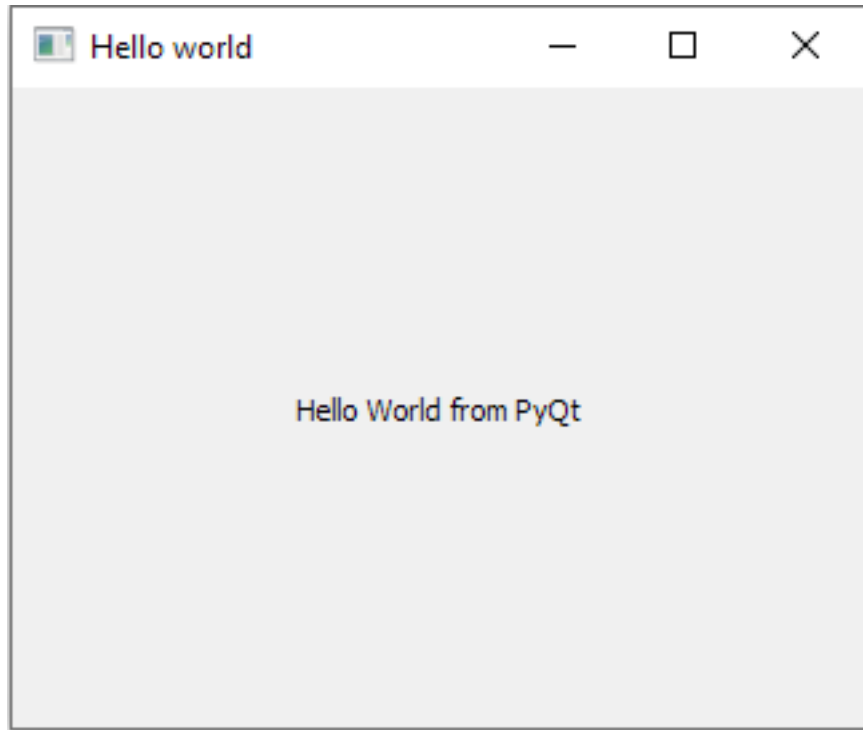
gridLayout = QGridLayout(self)
centralWidget.setLayout(gridLayout)

title = QLabel("Hello World from PyQt", self)
title.setAlignment(QtCore.Qt.AlignCenter)
gridLayout.addWidget(title, 0, 0)

if __name__ == "__main__":
    app = QtWidgets.QApplication(sys.argv)
    mainWin = HelloWorld()
    mainWin.show()
    sys.exit(app.exec_())
```

上面的这个程序和 Tkinter 的还是有很多相似点的，首先我们通过继承 QMainWindow 的方式创建一个自己的窗口，在窗口中创建中心组件用来显示内容，然后设置中心组件的布局为网格布局，在网格布局的中心位置显示一个标签组件，最后显示我们创建的窗口并且进入一个类似的事件循环。

显示效果如下：



通常在代码里面去创建需要用到的组件的方式非常麻烦，容易出错，而且和业务逻辑代码混合起来不利于开发和维护，Qt 程序支持使用 xml 格式的文件描述用户界面组件、布局以及设置各种属性（例如文本、字体颜色、前景色、背景色等）和事件处理函数（在 Qt 中也叫作信号/槽）。通过 Qt 提供的 **designer** 工具进行图形界面设计可以很方便的通过拖拽等方式快速的设计好图形界面的 UI 部分，然后在 Python 中加载前面设计好的 UI，添加例如菜单点击、按钮点击等事件的处理函数，这样的实现方式可以使 UI 和业务逻辑代码分离，减少耦合性，有利于协同开发合作。**designer** 工具的使用本章不做介绍，下面通过在代码里面创建组件的方式以及在 **designer** 里设计用户界面然后在加载界面和添加业务逻辑代码的方式展示一个计算器小程序的例子给大家参考学习。

在代码里面创建组件的方式：

```
#coding: utf-8
```

```
import sys
from PyQt5.QtWidgets import QWidget, QGridLayout, QLineEdit, QPushButton, QApplication
```

```
class Calculator(QWidget):
```

```
    def __init__(self):
        super().__init__()
        self.initUI()
```

```
    def initUI(self):
        # 使用grid 布局显示界面
        grid = QGridLayout()
        self.setLayout(grid)
        # 定义按钮显示内容
        names = [' ', ' ', ' ', 'cls',
                  '7', '8', '9', '/',
                  '4', '5', '6', '*',
                  '1', '2', '3', '-',
                  '0', '.', '=', '+']
```

```
        positions = [(i, j) for i in range(5) for j in range(4)]
```

```
        for position, name in zip(positions, names):
            if name == "":
                continue
            button = QPushButton(name)
            # 设置button 的点击事件处理方法(signal/slot)
            button.clicked.connect(self.on_button_clicked)
            grid.addWidget(button, *position)
```

```
        # 设置显示结果的输入框
        self.resultLineEdit = QLineEdit()
        self.resultLineEdit.setReadOnly(True)
        grid.addWidget(self.resultLineEdit, 0, 0, 1, 3)
```

```
        self.move(300, 150)
        self.setWindowTitle('简易计算器')
```

```

self.show()

def on_button_clicked(self):
    button = self.sender()
    text = button.text()
    if text == "":
        return
    elif text in "0123456789+-*/.":
        self.resultLineEdit.insert(text)
    elif text in "=":
        try:
            self.resultLineEdit.insert('={result}'.format(
                result=eval(self.resultLineEdit.text())))
        except:
            self.resultLineEdit.setText("some error occurs, press cls!")
    elif text in "cls":
        self.resultLineEdit.clear()

if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = Calculator()
    sys.exit(app.exec_())

```

在 designer 里设计用户界面然后在加载界面和添加业务逻辑代码的方式：

通过 designer 设计的用户界面文件 calculator.ui（xml 格式的，内容比较长）

```

<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
<class>MainWindow</class>
<widget class="QMainWindow" name="MainWindow">
<property name="geometry">
<rect>
<x>0</x>
<y>0</y>
<width>309</width>
<height>192</height>

```

```

</rect>
</property>
<property name="windowTitle">
  <string>MainWindow</string>
</property>
<widget class="QWidget" name="centralwidget">
  <layout class="QGridLayout" name="gridLayout">
    <property name="sizeConstraint">
      <enum>QLayout::SetMaximumSize</enum>
    </property>
    <property name="leftMargin">
      <number>0</number>
    </property>
    <property name="topMargin">
      <number>0</number>
    </property>
    <property name="rightMargin">
      <number>0</number>
    </property>
    <property name="bottomMargin">
      <number>0</number>
    </property>
    <property name="spacing">
      <number>0</number>
    </property>
    .....
  <connection>
    <sender>actionResetStyle</sender>
    <signal>triggered()</signal>
    <receiver>MainWindow</receiver>
    <slot>on_action_triggered()</slot>
    <hints>
      <hint type="sourcelabel">
        <x>-1</x>
        <y>-1</y>
      </hint>
      <hint type="destinationlabel">
        <x>154</x>
        <y>95</y>
      </hint>
    </hints>
  </connection>
</connections>
<slots>
  <slot>on_button_clicked()</slot>
  <slot>on_action_triggered()</slot>

```

```
</slots>
</ui>
```

```
#coding: utf-8
```

```
from PyQt5.uic import loadUi
from PyQt5.QtCore import pyqtSlot
from PyQt5.QtWidgets import QWidget, QGridLayout, QLineEdit, QPushButton, QAp
plication, QMainWindow, QActionGroup, QAction
import sys
import os
import qdarkstyle
import qdarkgraystyle
# import PyQt5_stylesheets
app = None
```

```
class Window(QMainWindow):
```

```
    def __init__(self):
```

```
        super(Window, self).__init__()
        loadUi('calculator.ui', self)
```

```
        actionGroupDefaultStyle = QActionGroup(self)
        actionGroupDefaultStyle.addAction(self.actionFusion)
        actionGroupDefaultStyle.addAction(self.actionWindows)
        actionGroupDefaultStyle.addAction(self.actionWindowsVista)
        actionGroupDefaultStyle.setExclusive(True)
```

```
        actionGroupStyle = QActionGroup(self)
        actionGroupStyle.addAction(self.actionQdarkstyle)
        actionGroupStyle.addAction(self.actionQdarkgraystyle)
        actionGroupStyle.addAction(self.actionMyStyle)
        actionGroupStyle.addAction(self.actionResetStyle)
        actionGroupStyle.setExclusive(True)
```

```
        self.show()
```

```
@pyqtSlot()
```

```
def on_action_triggered(self):
```

```
    action = self.sender()
    text = action.text()
```

```
    if text == "Exit":
```

```
        app.quit()
```

```
    elif text == "Fusion":
```

```
        app.setStyle('Fusion')
```

```
    elif text == "Windows":
```

```

    app.setStyle('Windows')
elif text == "WindowsVista":
    app.setStyle('WindowsVista')
elif text == "Qdarkstyle":
    # https://github.com/ColinDuquesnoy/QDarkStyleSheet
    self.setStyleSheet(qdarkstyle.load_stylesheet_pyqt5())
elif text == "Qdarkgraystyle":
    # https://github.com/mstuttgart/qdarkgraystyle
    self.setStyleSheet(qdarkgraystyle.load_stylesheet())
elif text == "MyStyle":
    self.setStyleSheet(open('style/material/material-blue.qss').read())
elif text == "ResetStyle":
    self.setStyleSheet("")

@pyqtSlot()
def on_button_clicked(self):
    button = self.sender()
    text = button.text()
    if text == "":
        return
    elif text in "0123456789+-*/.":
        self.lineEditResult.insert(text)
    elif text in "=":
        try:
            self.lineEditResult.insert('={result}'.format(
                result=eval(self.lineEditResult.text())))
        except:
            self.lineEditResult.setText("some error occurs, press cls!")
    elif text in "cls":
        self.lineEditResult.clear()

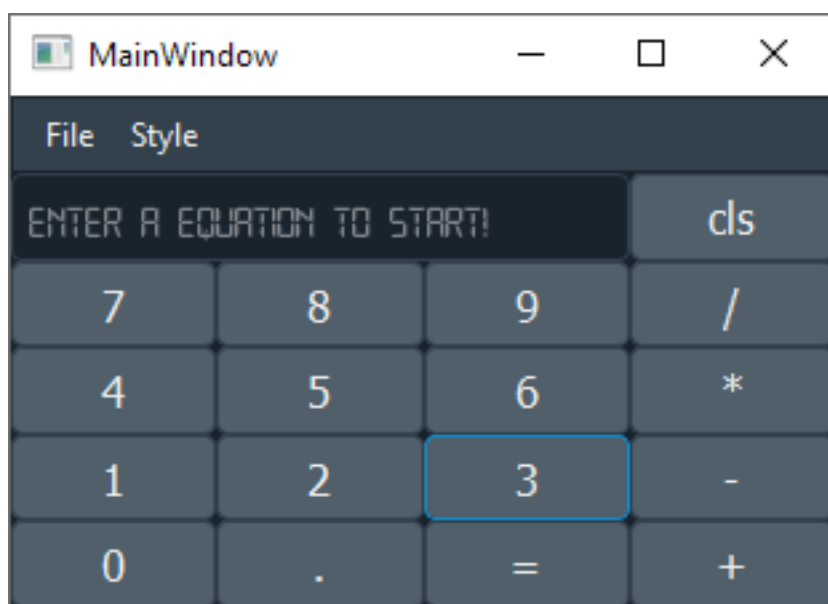
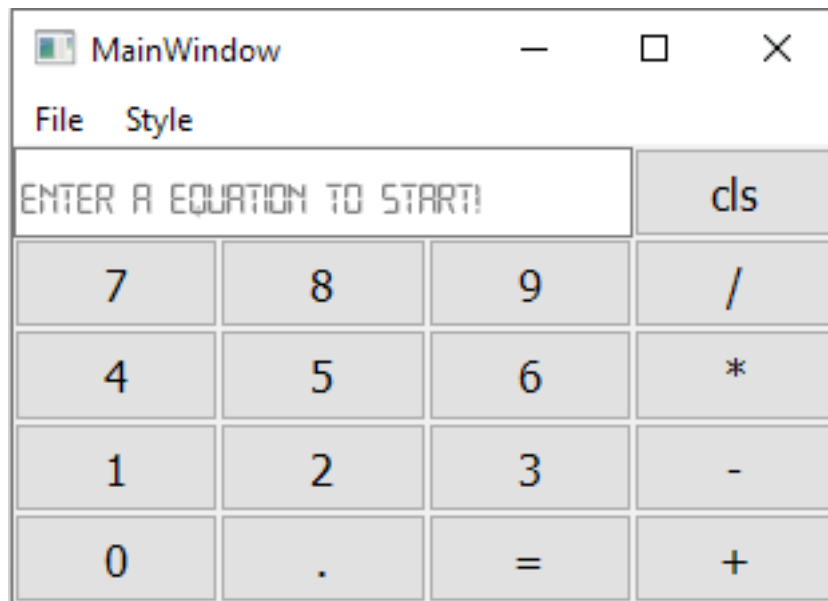
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = Window()
    sys.exit(app.exec_())

```

后面提供的这个版本的程序还通过 `css` 提供不同的 UI 样式，需要使用一些第三方库，这些定义在一个 `requirements.txt` 文件中：

```
pyqt5
pyqt5-tools
qdarkstyle
qdarkgraystyle
```

运行之后的界面如下：



MainWindow

File

Style

ENTER A EQUATION TO START!

cls

| | | | |
|---|---|---|---|
| 7 | 8 | 9 | / |
| 4 | 5 | 6 | * |
| 1 | 2 | 3 | - |
| 0 | . | = | + |

0.9.3 PyGObject

PyGObject 的前身也叫作 PyGTK+, 它和 PyQt 类似, 也是 Python 对 GTK+ 的封装, GTK+ 是一个在 Linux 图形界面编程领域非常出名的一个库, 它最开始用于开发 GIMP 的, 之后分离出来作为通用的图形界面开发库, 在 Linux 中 Gnome 图形窗口系统也是采用 GTK+ 开发的, GTK+ 的编程语言是 C, PyGObject 则是使用 Python 封装 GTK3+ 的 API, 方便用户快速创建基于 GTK3+ 的桌面图形界面应用程序。

在不同平台上安装配置 PyGObject 略微有些不同, 详细可参考 https://pygobject.readthedocs.io/en/latest/getting_started.html#window-s-getting-started, 其中在 Windows 下安装配置需要在 msys2 环境下面进行:

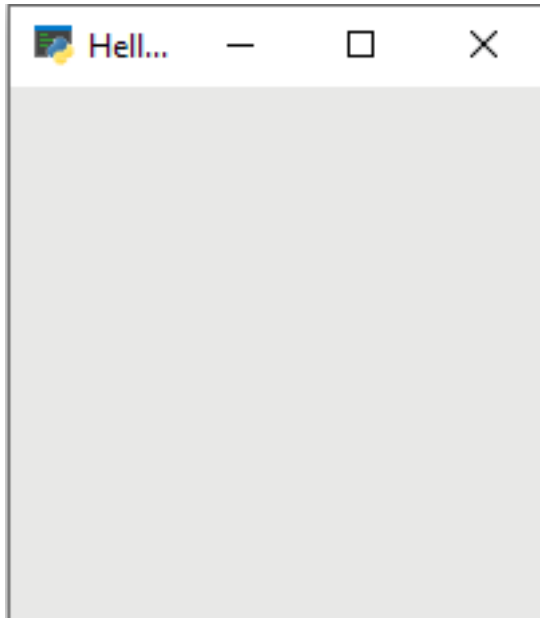
1. 从 <http://www.msys2.org/> 上下载 msys2 的安装程序
2. 运行 msys2 安装程序, 一步一步安装完成
3. 运行 C:\msys32\mingw32.exe 打开 msys2 的终端
4. 运行 `pacman -Suy` 更新 msys2
5. 运行 `pacman -S mingw-w64-i686-gtk3 mingw-w64-i686-python3-gobject` 安装 PyGObject 相关软件开发包

下面是一个简单的 PyGObject 版本的 hello world 小程序:

```
import gi
gi.require_version("Gtk", "3.0")
from gi.repository import Gtk
```

```
window = Gtk.Window(title="Hello World")
window.show()
window.connect("destroy", Gtk.main_quit)
Gtk.main()
```

运行上面的程序的截图如下：



另一个通过继承自定义窗口组件以及添加事件处理函数的例子如下：

```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk

class MyWindow(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self, title="Hello World")

        self.button = Gtk.Button(label="Click Here")
        self.button.connect("clicked", self.on_button_clicked)
        self.add(self.button)
```

```
def on_button_clicked(self, widget):
    print("Hello World")

win = MyWindow()
win.connect("destroy", Gtk.main_quit)
win.show_all()
Gtk.main()
```

同样 PyGObject 和 PyQt 类似，可以通过界面设计工具 glade 把 UI 部分的代码分离到一个独立的文件，与业务逻辑代码分开来写。下面是和上面相同的计算器程序使用这两种方式来做的示例代码。

在代码中创建组件的方式：

```
#coding: utf-8

import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk

class Window(Gtk.Window):

    def __init__(self):
        Gtk.Window.__init__(self, title="简易计算器")
        self.initUI()

    def initUI(self):
        # 使用 grid 布局显示界面
        grid = Gtk.Grid()
        self.add(grid)
        # 定义按钮显示内容
        names = [' ', ' ', ' ', 'cls',
                  '7', '8', '9', '/',
                  '4', '5', '6', '*',
                  '1', '2', '3', '-',
                  '0', '.', '=', '+']

        positions = [(i, j) for i in range(5) for j in range(4)]
```

```

for position, name in zip(positions, names):
    if name == "":
        continue
    button = Gtk.Button(label=name)
    button.set_hexpand(True)
    button.set_vexpand(True)
    # 设置 button 的点击事件处理方法
    button.connect("clicked", self.on_button_clicked)
    # 这里的位置需要逆转, 依次是 left top width height
    position = position[::-1] + (1, 1)
    grid.attach(button, *position)

# 设置显示结果的输入框
self.resultEntry = Gtk.Entry()
self.resultEntry.set_editable(False)
self.resultEntry.set_hexpand(True)
self.resultEntry.set_vexpand(True)
grid.attach(self.resultEntry, 0, 0, 3, 1)

def on_button_clicked(self, widget):
    text = widget.get_label()
    if text == "":
        return
    elif text in "0123456789+-*/.":
        self.resultEntry.set_text(self.resultEntry.get_text() + text)
    elif text in "=":
        try:
            self.resultEntry.set_text('{origin}={result}'.format(
                origin=self.resultEntry.get_text(),
                result=eval(self.resultEntry.get_text()))
        except:
            self.resultEntry.set_text("some error occurs, press cls!")
    elif text in "cls":
        self.resultEntry.set_text("")

if __name__ == "__main__":
    window = Window()
    window.connect("destroy", Gtk.main_quit)
    window.show_all()
    Gtk.main()

```

通过界面设计工具 glade 设计 UI 和分离业务逻辑代码的方式:

```

<?xml version="1.0" encoding="UTF-8"?>
<!-- Generated with glade 3.22.1 -->
<interface>
  <requires lib="gtk+" version="3.20"/>
  <object class="GtkImage">
    <property name="visible">True</property>
    <property name="can_focus">False</property>
    <property name="stock">gtk-close</property>
  </object>
  <object class="GtkWindow" id="window">
    <property name="can_focus">False</property>
    <signal name="destroy" handler="on_window_destroy" swapped="no"/>
    <child>
      <placeholder/>
    </child>
    .....
    <child>
      <object class="GtkButton" id="btnAdd">
        <property name="label" translatable="yes">+</property>
        <property name="visible">True</property>
        <property name="can_focus">True</property>
        <property name="receives_default">True</property>
        <property name="hexpand">True</property>
        <property name="vexpand">True</property>
        <signal name="clicked" handler="on_button_clicked" swapped="no"/>
      </object>
      <packing>
        <property name="left_attach">3</property>
        <property name="top_attach">4</property>
      </packing>
    </child>
  </object>
  <packing>
    <property name="expand">True</property>
    <property name="fill">True</property>
    <property name="position">1</property>
  </packing>
</child>
</object>
</child>
</object>
</interface>

```

#coding: utf-8

```
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk, Gio
```

```
resultEntry = None
window = None
```

```
class Handler:
```

```
    def on_window_destroy(self, *args):
        Gtk.main_quit()
```

```
    def on_menuitem_activate(self, widget):
        text = widget.get_label()
        if text == 'StyleElementary':
            provider = Gtk.CssProvider()
            provider.load_from_data(open("style/elementary/gtk.css").read())
            apply_css(window, provider)
            window.show_all()
        elif text == 'StyleElementaryDark':
            provider = Gtk.CssProvider()
            provider.load_from_data(open("style/elementary/gtk-dark.css").read())
            apply_css(window, provider)
            window.show_all()
        elif text == 'Exit':
            Gtk.main_quit()
```

```
    def on_button_clicked(self, widget):
        text = widget.get_label()
        if text == "":
            return
        elif text in "0123456789+-.*/.":
            resultEntry.set_text(resultEntry.get_text() + text)
        elif text in "=":
            try:
                resultEntry.set_text('{origin}={result}'.format(
                    origin=resultEntry.get_text(),
                    result=eval(resultEntry.get_text()))
            except:
                resultEntry.set_text("some error occurs, press cls!")
        elif text in "cls":
            resultEntry.set_text("")
```

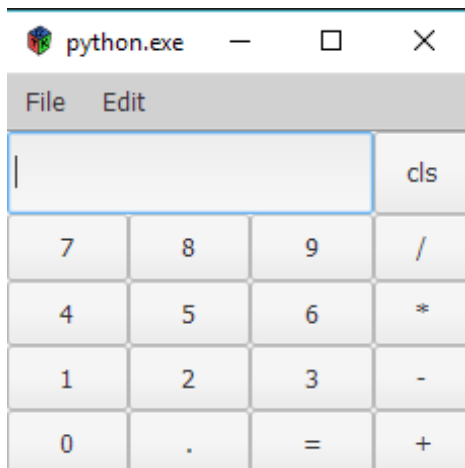
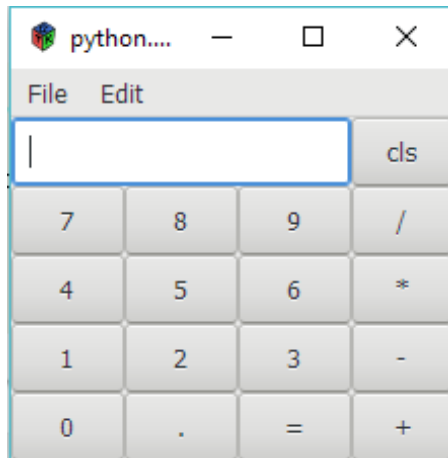
```

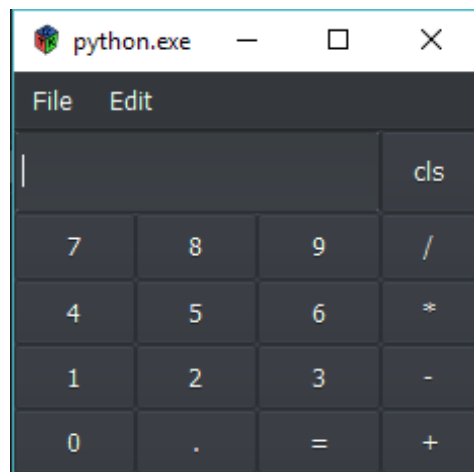
def apply_css(widget, provider):
    Gtk.StyleContext.add_provider(widget.get_style_context(),provider,Gtk.STYLE_PROVIDER_PRIORITY_APPLICATION)
    if isinstance(widget, Gtk.Container):
        widget.forall(apply_css, provider)

if __name__ == "__main__":
    builder = Gtk.Builder()
    builder.add_from_file("calculator.glade")
    builder.connect_signals(Handler())
    window = builder.get_object("window")
    resultEntry = builder.get_object("resultEntry")
    window.show_all()
    Gtk.main()

```

运行后的截图如下所示：





第十章 Web 开发

本章将简要介绍 Python 中的 Web 框架——Flask(<http://flask.pocoo.org/>)，Flask 是 Python 中非常流行的“微框架”，相比另一个主流的 Django(<https://www.djangoproject.com/>)框架相比，它更加轻量级和简单。Flask 框架非常小，框架本身代码也不多，在你使用 Flask 开发 Web 程序的过程中，可以阅读它的源码，对于学习 Python 来说是非常有益的，既可以深入学习 Web 编程技巧，又可以学习 Flask 里面优秀的编程思想和设计模式。

1.0.1 Flask 概要

Flask 主要有两个依赖：路由、调试和 Web 服务器网关接口(Web Server Gateway Interface, WSGI)子系统由 Werkzeug(<http://werkzeug.pocoo.org/>)提供，模板系统则是由 Python 中的经典模板引擎 Jinja2(<http://jinja.pocoo.org/>)提供。

Flask 并不直接支持数据库访问、Web 表单验证、用户认证等高级功能，这些功能通常是以 Flask 扩展包的形式提供，开发者可以直接使用已有的一些扩展包，例如针对数据库访问的 Flask-SQLAlchemy(<http://flask-sqlalchemy.pocoo.org/>)，针对 Web 表单验证的 Flask-WTF(<http://pythonhosted.org/Flask-WTF/>)，针对用户认证的 Flask-User(<https://flask-user.readthedocs.io/>)或 Flask-Login(<https://flask-login.readthedocs.io/>)，针对现在流行的前端框架——Bootstrap(<https://getbootstrap.com/>)封装的 Flask-Bootstrap(<https://pythonhosted.org/Flask-Bootstrap/>)。这些 Flask 扩展包可以帮我们做很多常见 Web 开发中需要用到的功能模块。如果没有找到适合自己场景或功能的扩展包，可以自己开发一个。

1.0.1.1 Flask 安装

大多数 Python 的第三方包(库)都是通过自带的 pip 来安装的，Flask 及其相关的扩展包也不例外，例如我们安装 Flask 可以执行下面的命令。

```
C:\Users\Liu.D.H>pip install flask # python 的包是不区分大小写的
```

执行上述命令之后，你就在自己电脑上全局安装了 Flask 及其依赖了，现在我们简单测试一下是否安装成功，在 Python 的交互式环境中，执行 `import flask` 如果没有报错提示包不存在则说明已经安装成功了。

```
C:\Users\Liu.D.H>python
Python 3.6.4 |Anaconda, Inc.| (default, Jan 16 2018, 10:22:32) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import flask
>>>
```

1.0.1.2 Flask 的一个简单例子

我们先写一个简单的 hello world 例子程序，后面再来介绍它是如何工作的。

```
# flask_hello.py

from flask import Flask
app = Flask(__name__)

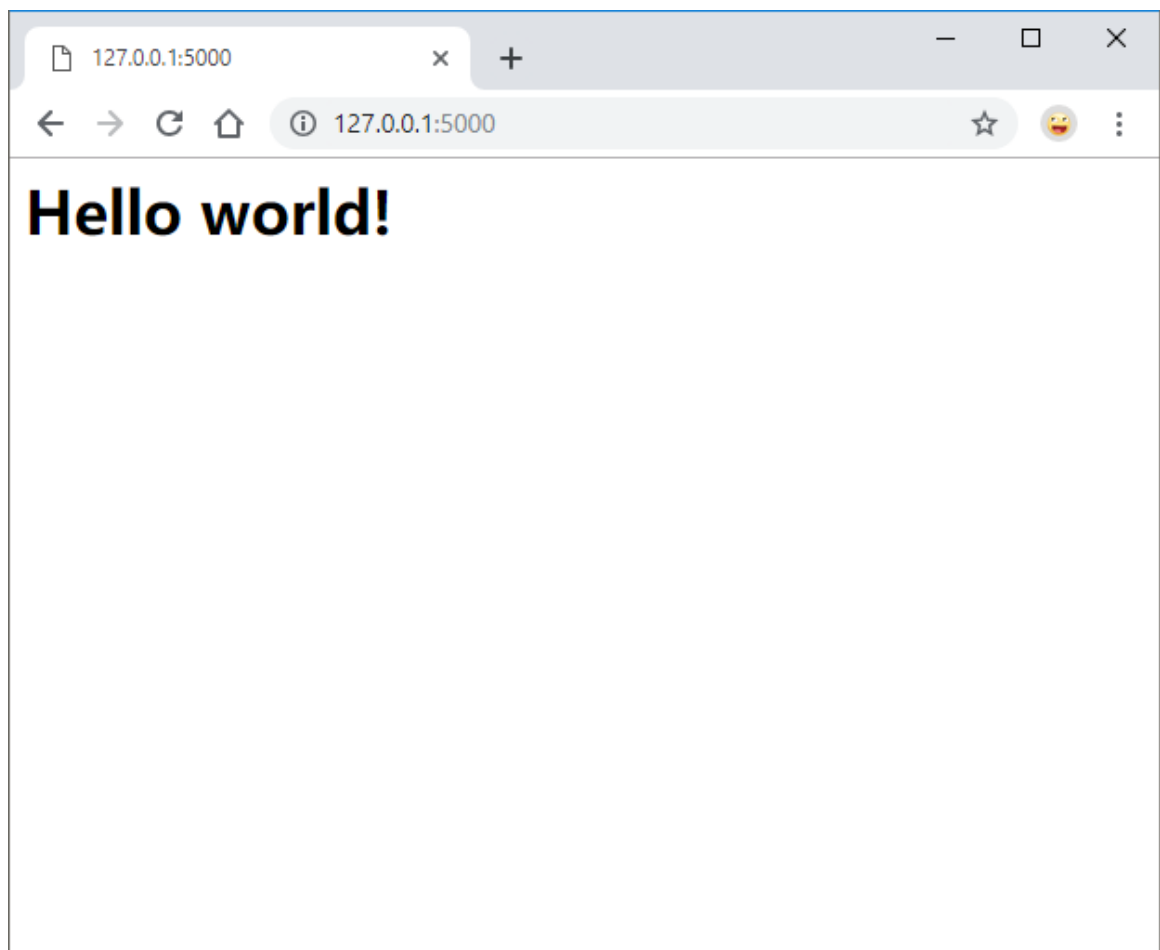
@app.route('/')
def index():
    return '<h1>Hello world!</h1>'

if __name__ == "__main__":
    app.run(debug=True)
```

我们把这个程序保存为 `flask_hello.py`，再 cmd 中通过 `cd` 切换到当前文件所在目录，然后通过 `python flask_hello.py` 来运行，运行输出类似于下面的内容。

```
D:\Documents\python\ebook\code>python flask_hello.py
* Serving Flask app "flask_hello" (lazy loading)
* Environment: production
  WARNING: Do not use the development server in a production environment.
  Use a production WSGI server instead.
* Debug mode: on
* Restarting with stat
* Debugger is active!
* Debugger PIN: 234-857-604
('127.0.0.1', 5000)
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

这时候我们打开浏览器，访问 <http://127.0.0.1:5000/>，例如再 Chrome 浏览器中的效果如下图所示。



到目前为止我们就完整的编写和运行了第一个 Python Web 程序。

这个程序包括三部分：初始化、路由和启动服务器。

```
from flask import Flask  
app = Flask(__name__)
```

这两行代码是做初始化的，每个 Flask 程序都必须创建一个 flask 实例，用来处理来自客户端发送过来的 HTTP 请求，flask 对象实例的构造函数通过传递进去的 `__name__` 参数告诉 Flask 程序运行的根目录是在哪里，Flask 程序的初始化一般都是这样写的。

```
@app.route('/')  
def index():  
    return '<h1>Hello world!</h1>'
```

这三行代码定义了 `/'` 路由，这里是通过 Python 里面的装饰器来定义路由的，装饰器在 Python 里常用于定义一些事件回调函数，在 GUI 编程中也有类似用法。如果启动的 Web 服务是在本地的默认 5000 端口上，那我们访问 `http://localhost:5000/` 就是访问 `/'` 路由，如果部署到一个绑定的域名(例如 `www.example.com`)和 http 默认 80 端口上，则这个路由对应的访问网址是 `http://www.example.com/`，一般我们在做 Web 开发的时候在程序里是不需要关注访问的域名或者 IP 的。

像 `index()` 这样的函数叫做视图函数(view function)，视图函数返回的响应可以是简单的 HTML 字符串或者是一些复杂的 JSON/XML 字符串，我们后面会给大家在介绍。

```
if __name__ == "__main__":  
    app.run(debug=True)
```

这两行代码是用来启动 Flask 集成的 Web 服务器的，`if __name__ == "__main__"` 是 Python 中程序的常见用法，当直接运行这个脚本的时候这个条件为 `True`，如果这个脚本（模块）是通过其他脚本引入的(`import`)，则这个条件为 `False`，不会执行 `app.run(debug=True)`，`app.run` 方法可以参数设置调整 Web 服务器的运行模式，比如这里的 `debug=True` 参数设置在开发程序的时候就非常好用——当我们保存程序文件的时候，Flask 检测到文件有变化，会自动重新运行这个脚本，这就不需要我们使用 `Ctrl+C` 先停止程序运行，然后再 `python flask_hello.py` 运行。

1.0.2 请求-响应循环

现在你已经写了一个简单的 Web 程序，接下来，我们进一步了解 Flask 的运行方式，这对于我们学习后面的只是是很有帮助的。Web 程序运行后会一直运行，接受请求-处理-返回响应，这样一直不停循环下去，直到我们停止 Web 程序。

1.0.2.1 程序和请求上下文

Web 程序一般都是无状态的，即每次访问，只要是相同的 URL，一般响应也是相同的。Flask 从客户端收到请求后，要让视图函数能够访问客户端发送过来的请求，这些请求除了明显的 URL，可能还包括一些隐藏在背后的例如 Cookie、Accept-Language 等，这些 HTTP 请求信息在 Flask 中都封装在请求**对象**里面。为了在视图函数中能够访问**请求对象**，可以通过视图函数参数的方式传递进去，但是这样每个视图函数都要写这些参数，很不方便，在 Flask 中，使用上下文临时把某些对象变成**全局变量**，这样就方便在视图函数中使用，例如下面的程序。

```
from flask import request

@app.route('/')
def index():
    user_agent = request.headers.get('User-Agent')
    return '<p>Your browser is {user_agent}</p>'.format(user_agent=user_agent)
```

在这个视图函数中，request 像是一个全局变量，实际上它不可能是全局变量，在多线程服务器中，每个线程同时处理不同客户端请求的时候，request 的值是不同的。Flask 使用上下文让特定的变量在一

个线程中全局可访问，同时也不会影响其他线程。Flask 的上下文有程序上下文和请求上下文，如下表所示。

| 变量名 | 上下文 | 说明 |
|--------------------------|-------|-------------------------|
| <code>current_app</code> | 程序上下文 | 当前程序的实例 |
| <code>g</code> | 程序上下文 | 处理请求的临时对象，每次请求都会重新创建 |
| <code>request</code> | 请求上下文 | 请求对象，封装HTTP请求信息 |
| <code>session</code> | 请求上下文 | 会话对象，用于存储相关联请求之间需要共享的信息 |

Flask 在分发请求的时候激活（或推送）程序和请求上下文，处理请求完成后再将其删除，程序上下文激活后就可一再线程中安全的使用 `current_app` 和 `g` 变量；请求上下文被激活后就可一使用 `request` 和 `session` 变量。下面在 Python 交互式环境下演示上下文的使用。

```
>>> from flask_hello import app
>>> from flask import current_app
>>> current_app
<LocalProxy unbound>
>>> app_ctx = app.app_context()
>>> app_ctx.push()
>>> current_app
<Flask 'flask_hello'>
>>> app_ctx.pop()
>>> current_app
<LocalProxy unbound>
>>>
```

再这个例子中，如果没有激活程序上下文直接访问 `current_app` 的时候，是显示<LocalProxy unbound>的，当激活了之后再次访问的时候即显示了 Flask 的实例<Flask 'flask_hello'>。

1.0.2.2 请求调度

程序收到来自客户端的请求之后，如果知道应该调用哪个视图函数来处理这个请求呢？再 Flask 中，是通过在 **URL 映射** 中通过请求 URL 来查找其对应的视图函数。下面在 Python 交互式环境下演示 **URL 映射** 的使用。

```
>>> from flask_hello import app
>>> app.url_map
Map([<Rule '/' (OPTIONS, HEAD, GET) -> index>,
<Rule '/static/<filename>' (OPTIONS, HEAD, GET) -> static>])
>>>
```

这里，‘/’路径的 URL 的 OPTIONS, HEAD, GET 方法对应的视图函数是 index，另一个‘/static/’路径的 URL 的 OPTIONS, HEAD, GET 方法对应的视图函数是 static，这个不是在我们程序里面定义的，它是 Flask 自动注册的，用来处理静态资源请求的。

1.0.2.3 请求钩子

有时候在请求处理之前或之后需要做一些处理，例如在请求之前检测用户是否登陆、用户权限，这些处理一般在特定的一系列 URL 请求之前都需要处理，如果都在各个视图函数中去调用，则代码写起来会有些冗余，不怎么优雅，而且后期如果需要修该则需要改动多个地方，容易出错。在 Flask 中，我们可以通过其提供的通用函数注册功能，在每个请求处理之前或之后调用特定的函数（也叫做钩子，hooks）。钩子一般是通过装饰器来定义的，主要有下面四种。

- `before_first_request`: 注册一个函数，在第一请求处理之前调用
- `before_request`: 注册一个函数，在每次请求之前调用
- `after_request`: 注册一个函数，如果没有异常抛出，在每次请求处理之后调用。
- `teardown_request`: 注册一个函数，不管时候又异常抛出，在每次请求处理之后调用。

例如下面的代码片段

```
# flask_hello_hooks.py

from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return '<h1>Hello world!</h1>'

@app.before_first_request
def before_first_request():
    print('before_first_request executed')

@app.before_request
def before_request():
    print('before_request executed')

@app.after_request
def after_request(response):
    print('after_request executed')
    return response
```

```
@app.teardown_request
def teardown_request(response):
    print('teardown_request executed')
    return response

if __name__ == "__main__":
    app.run(debug=True)
```

运行时候访问两次 <http://127.0.0.1:5000/> 的终端输出信息如下所示。注意第二次没有“before_first_request executed”的输出。

```
D:\Documents\python\ebook\code>python flask_hello_hooks.py
* Serving Flask app "flask_hello_hooks" (lazy loading)
* Environment: production
  WARNING: Do not use the development server in a production environment.
  Use a production WSGI server instead.
* Debug mode: on
* Restarting with stat
* Debugger is active!
* Debugger PIN: 234-857-604
('127.0.0.1', 5000)
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
before_first_request executed
before_request executed
after_request executed
teardown_request executed
127.0.0.1 - - [19/Mar/2019 13:13:34] "GET / HTTP/1.1" 200 -
before_request executed
after_request executed
teardown_request executed
127.0.0.1 - - [19/Mar/2019 13:13:38] "GET / HTTP/1.1" 200 -
```

1.0.3 模板

在前面的例子中，视图函数只是返回一个简单的 HTML 片段(

Hello world!

)，在真实环境中，一般需要返回相对来说稍微复杂一点的 HTML 内容，里面一般有动态的内容，如用户名，这时候如果在视图通过字符串拼接的方式十分低效而且还容易出错。这时候就需要使用模板，简单来说模板就是一个文本内容，里面有变量、控制语句如判断循环、子模版（包含或继承其他模板，便于模块化编程）等，可以方便灵活的生成最终的 HTML 内容。模板的文本内容一般是保存到一个文件中，这样方便复用。使用真实值替换模板中的变量，这个过程叫做模板渲染。Flask 中提供了一个功能强大的模板引擎，叫做 Jinja2。

1.0.3.1 Jinja2 模板引擎

默认情况下，Flask 会在当前的 `templates` 子文件夹下寻找模板。以下是一个简单的例子，有两个模板文件，`templates/index.html` 和 `templates/user.html`，其内容分别是：

```
<h1>Hello, World!</h1>
<h1>Hello, {{ name }}!</h1>
```

`flask_hello_template.py` 文件内容是：

```
from flask import Flask, render_template
app = Flask(__name__)
```

```
@app.route('/')
def index():
    return render_template('index.html')

@app.route('/user/<name>')
def user(name):
    return render_template('user.html', name=name)

if __name__ == "__main__":
    app.run(debug=True)
```

在这里我们使用 flask 中的 `render_template` 函数，这个函数第一个参数是模板文件在 `templates` 文件夹下的相对路径，后面的参数是传递给模板的变量内容，例如传递给 `user.html` 模板的 `name` 变量，它的值是请求 URL 中的 `path` 变量。



← → ↻ 🏠 ⓘ 127.0.0.1:5000/user/eric

Hello, eric!

1.0.3.2 模板变量

在模板中的 `{{ name }}` 表示一个变量，是一个占位符，在渲染模板的时候会使用 `name` 变量的值替换这一块内容，Jinja2 能识别 Python 中所有常用类型的变量，甚至是一些复杂的数据类型，例如列表、字典和对象，如下所示。

```
<p>render dict: {{ mydict['key'] }}</p>  
<p>render list: {{ mylist[0] }}</p>  
<p>render object: {{ myobject.someVariable }}, {{myobject.someMethod() }}</p>
```

可以使用过滤器（**filters**）修改变量值，过滤器是在变量后使用|管道符，然后加上过滤器名，例如下面的模板一首字母大写的形式显示变量 **name** 的值：

```
Hello, {{ name | capitalize }}
```

下面列举 Jinja2 提供的部分常用过滤器：

| 过滤器名 | 说明 |
|------------|-------------------|
| safe | 渲染时不对内容转义 |
| capitalize | 首字母大写，其他字母小写 |
| lower | 全部字母小写 |
| upper | 全部字母大写 |
| title | 每隔单子的首字母都大写 |
| trim | 去除首尾的空白符，例如空格、Tab |
| striptags | 去除所有的HTML标签，只保留内容 |

默认情况下，处于 Web 安全考虑，Jinja2 会转义所有的变量，例如一个变量值的内容是<h1>hello</h1>，Jinja2 会将其渲染成<h1>hello</h1>，在 HTML 里面显示的就是

hello

，而不是一个格式化之后的一级标题 **hello**，这能避免一些恶意的 js 代码执行。如果你在特定的场景下不需要这种自动转义功能，在确

认信任、安全时，可以使用 **safe** 过滤器。完整的内置过滤器可以在 Jinja2 文档中（<http://jinja.pocoo.org/docs/2.10/templates/#builtin-filters>）查看。

1.0.3.3 模板控制语句

在模板中通常还会配合控制语句用来增强模板功能，例如下面的模板中使用条件控制语句：

```
{% if user %}
    Hello, {{ user }}
{% else %}
    Hello, Stranger!
{% endif %}
```

例如通过 **for** 循环渲染一个列表里的内容：

```
<ul>
    {% for comment in comments %}
    <li>{{ comment }}</li>
    {% endfor %}
</ul>
```

Jinja2 支持宏，类似于 Python 中的函数，例如：

```
{% macro render_comment(comment) %}
<li>{{ comment }}</li>
{% endmacro %}

<ul>
    {% for comment in comments %}
    {{
        render_comment(comment)
    }}
</ul>
```



```
{% endfor %}  
</ul>
```

一般可以把可复用的宏保存到一个单独的文件中，例如 `macros.html`，然后在需要使用的模板中 `import` 这个宏文件即可：

```
{% import 'macros.html' as macros %}  
<ul>  
  {% for comment in comments %}  
    {{  
      macros.render_comment(comment)  
    }}  
  {% endfor %}  
</ul>
```

和 `import` 宏类似，我们可以把一些会重复使用的模板片段保存到一个单独的文件中，在 `include` 到使用到的模板中，以便于代码复用，例如：

```
{% include 'header.html' %}  
{% include 'footer.html' %}
```

另一种复用模板代码的方式是类似于面向对象里面的继承，例如首先我们可以定义一个基础模板 `base.html`：

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
  <meta charset="UTF-8">  
  <meta name="viewport" content="width=device-width, initial-scale=1.0">  
  <meta http-equiv="X-UA-Compatible" content="ie=edge">  
{% block head %}  
  <title>{% block title %}{% endblock title %}</title>
```

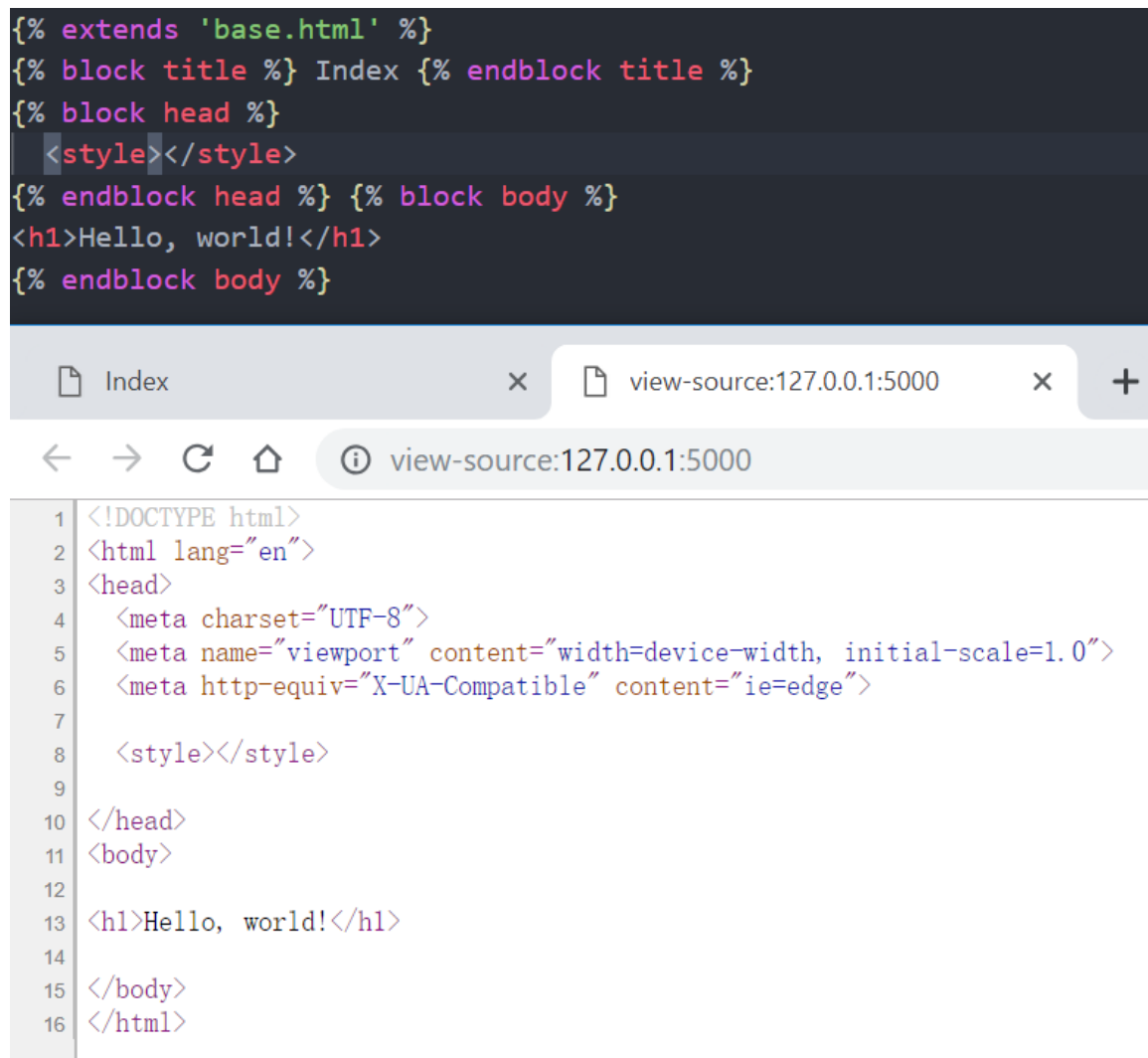
```
{% endblock head %}  
</head>  
<body>  
    {% block body %}  
    {% endblock body %}  
</body>  
</html>
```

`block` 标签定义的内容可以在继承的模板中进行修改，在上面的 `base.html` 中，我们定义了名为 `head`、`title`、`body` 的块，其中 `title` 是包含在 `head` 里的，下面我们继承上面的基础模板：

```
{% extends 'base.html' %}  
{% block title %} Index {% endblock title %}  
{% block head %}  
    {{ super() }}  
    <style></style>  
{% endblock head %} {% block body %}  
<h1>Hello, world!</h1>  
{% endblock body %}
```

`extends` 声明这个模板继承 `base.html`，在 `head` 块中，由于基础模板中的内容不是空的，所以需要使用 `super()` 获取原来的内容，这样 `title` 块中的内容才能正确的插入到模板中，如果没有 `super()`，则最终生成的 `html` 内容如下图所示：

```
{% extends 'base.html' %}
{% block title %} Index {% endblock title %}
{% block head %}
    <style></style>
{% endblock head %} {% block body %}
<h1>Hello, world!</h1>
{% endblock body %}
```



```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <meta http-equiv="X-UA-Compatible" content="ie=edge">
7
8   <style></style>
9
10 </head>
11 <body>
12
13 <h1>Hello, world!</h1>
14
15 </body>
16 </html>
```

1.0.3.4 使用 Flask-Bootstrap

Bootstrap 是 Twitter 开发的一套前端框架，提供了一套样式统一、友好、漂亮的用户组件，用于快速创建现代的 Web 用户界面。它是属于客户端浏览器层面上的，不会直接涉及 Web 服务端，但是创建一个网站是一系列网页构成的，这些网页应该具有一致性，所以通过模板来定义一个基础模板，然后再在其基础上显示内容是 Web 开发的一个比较好的最佳实践。在 Flask 中使用 Bootstrap 可以使用 Flask-Bootstrap 扩展库。

在使用 Flask-Bootstrap 扩展库之前需要对 app 实例进行初始化，如下代码所示：

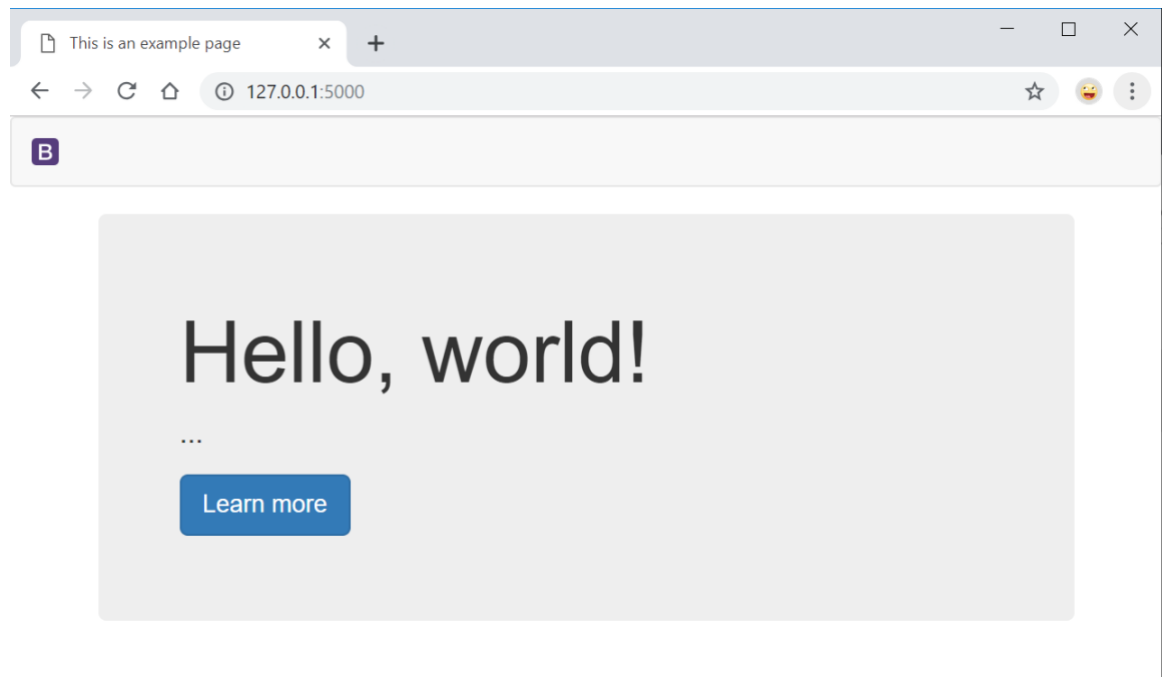
```
#.....  
from flask_bootstrap import Bootstrap  
  
#.....  
Bootstrap(app)
```

这样就可以使用 Flask-Bootstrap 的模板，以下是一个简单的例子：

```
{% extends "bootstrap/base.html" %}  
{% block title %}This is an example page{% endblock %}  
  
{% block navbar %}  
<nav class="navbar navbar-default">  
  <div class="container-fluid">  
    <div class="navbar-header">  
      <a class="navbar-brand" href="#">  
          
      </a>  
    </div>  
  </div>  
</nav>  
{% endblock %}
```

```
{% block content %}  
<div class="container">  
  <div class="jumbotron">  
    <h1>Hello, world!</h1>  
    <p>...</p>  
    <p><a class="btn btn-primary btn-lg" href="#" role="button">Learn more</a><  
/p>  
  </div>  
</div>  
{% endblock %}
```

这时候网页的效果如下图所示：



可以看到现在这个网页的显示效果已经改善了很多。具体更多内容可以参考 Bootstrap(<https://getbootstrap.com/docs/>)和 Flask-Bootstrap(<https://pythonhosted.org/Flask-Bootstrap>)的文档。

Flask-bootstrap 的 base.html 模板还定义了很多其他块，都可以在继承的模板中使用，下表列出了所有可用的块(<https://pythonhosted.org/Flask-Bootstrap/basic-usage.html#available-blocks>)。

| 块名称 | 外层块名称 | 用途 |
|--------------|-------|---|
| doc | | 最外层的块 |
| html | doc | 包括整个 <code><html></code> 标签的内容 |
| html_attribs | doc | <code><html></code> 标签的属性 |
| head | doc | 包括整个 <code><head></code> 标签的内容 |
| body | doc | 包括整个 <code><body></code> 标签的内容 |
| body_attribs | body | <code><body></code> 标签的属性 |
| title | head | 包括整个 <code><title></code> 标签的内容 |
| styles | head | 所有的head中的 <code><link></code> 标签的内容 |
| metas | head | 包括整个 <code><meta></code> 标签（元素据标签）的内容 |
| navbar | body | body中的内容之上的导航栏内容 |
| content | body | body中的内容 |
| scripts | body | 所有的body中的 <code><script></code> 标签的内容 |

1.0.3.5 模板链接

在模板中的链接直接写链接相对/绝对路径虽然可以，但是容易出错，例如前面例子中的 `index()` 视图函数对应的相对路径是 `/`，如果后面调整这个路径为 `/home`，则需要修改所有模板中路径为 `/` 的链接，非常的不方便，而且还容易出错。

为了解决上述问题，Flask 提供了 `url_for()` 函数，用于根据视图函数名生成对应的正确路径，例如 `flask_hello.py` 中调用 `url_for('index')` 返回的是 `/`，调用 `url_for('index', _external=True)` 返回的是绝对路径 `http://localhost:5000/`。`url_for()` 函数还可以将动态的内容或者查询参数

通过关键字参数的形式传入，例如 `url_for('user', name='eric')` 返回的是 `/user/eric`，`url_for('index', page=1)` 返回的是 `/?page=1`。

在前面的 URL 映射学习的时候，有一个 `static` 路由，路径是 `/static/<filename>`，这是 Flask 对静态文件的特殊处理的路由，例如调用 `url_for('static', filename='favicon.ico')`，返回的路径是 `/static/favicon.ico`，这个路由在每个页面中几乎都会用到，其他静态资源如 `css`、`javascript` 文件的路径一般也是通过这种方式来引用的。

第十一章 综合案例

1.1.1 网页爬虫

网页爬虫就是通过程序自动的从互联网上抓取特定的 HTML 以及相关内容，然后做一些分析处理，例如比较经典的场景是搜索引擎，它不停的抓取互联网上的网页信息，对他们进行分析、索引、存储等处理，最后提供一个搜索网站，根据用户输入的关键词在搜索引擎数据库中去查找相关网页信息，并且按照搜索结果匹配度从高到底优先级方式展示给用户，为用户提供便捷快速的搜索服务。

在 Python 中在写网页爬虫是非常方便的，通常结合 requests 和 BeautifulSoup4,复杂一点的场景下会用到 Selenium 或者 scrapy 等框架。

下面通过一个抓取天气预报的实际案例来讲解网页爬虫。网页爬虫是通过 HTTP 请求获取到网页的 html 源代码，是纯文本字符串，为了方便解析里面的数据，可以使用 BeautifulSoup4 来解析 html 文本内容，这样就可以通过 css 选择器灵活的定位和获取到指定元素的文本或者属性值等内容。

1.1.2 抓取天气的爬虫示例

我们获取的天气信息是在中国天气 (<http://www.weather.com.cn/>) 网站上的, 假设我们只是想获取昆明及周边的天气信息, 经过初步分析, 这些信息都在 <http://www.weather.com.cn/weather/101290101.shtml> 这个页面上, 如下图所示:



周边地区的天气信息同样是在这个网页上，如下图所示：

周边地区 | 周边景点

2019-05-05 11:30更新

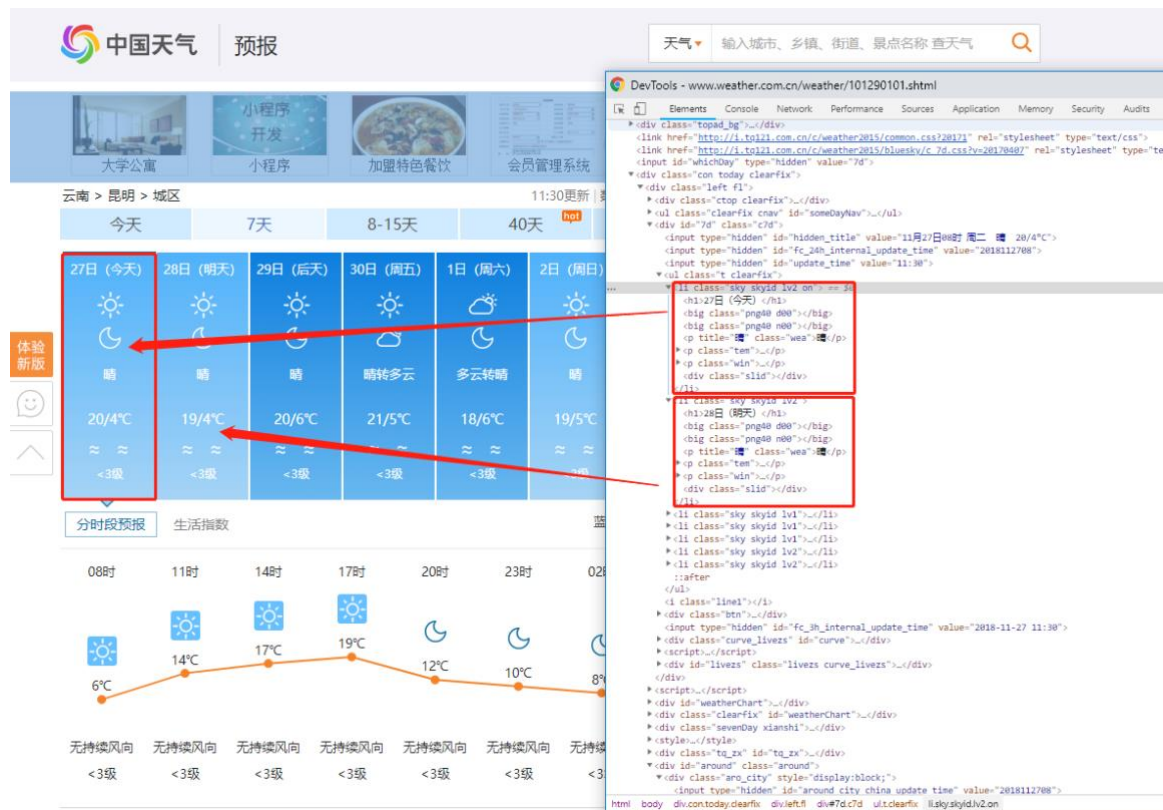
| | | | |
|----|---|-----|---|
| 晋宁 |  30/16°C | 黔西南 |  25/15°C |
| 曲靖 |  28/15°C | 寻甸 |  30/15°C |
| 安宁 |  29/17°C | 禄劝 |  31/15°C |
| 嵩明 |  29/15°C | 攀枝花 |  36/23°C |
| 富民 |  31/16°C | 楚雄 |  30/18°C |
| 呈贡 |  28/17°C | 玉溪 |  30/16°C |

高清图集

>>

现在我们在通过查看网页源代码的方式分析一下这些天气信息内容是否就在静态的 `html` 文本中，如果在其中，那么通过使用 `Beautiful Soup4` 来解析就可以，如果不在其中，那说明网页内容是通过 `ajax` 动态获取和渲染的，这时候就需要使用 `Selenium` 来模拟浏览器加载网页完成，然后解析完成之后的 `html`（我们也可以称之为动态的 `html`）内容来提取相关信息。

在这个天气示例中，这些天气信息已经包含在静态的 `html` 文本中，因此处理相对简单。通过 `Chrome` 的开发者工具可以查看 `html` 的，定位到相关元素，就可以通过 `css` 选择器定位到这一元素，例如通过 `id` 选择器、`class` 选择器或者元素名和 `class` 的组合选择器。



例如上图昆明主城区的一周七天的天气信息都是在包含 sky 和 sky id 的 class 选择器对应的 li 元素中，其 css 选择器可以是 li.sky.skyid，下面周边天气信息经过分析其选择器可以使 div.aro_city ul li，下面是完整的代码：

#coding: utf8

```
import requests
from bs4 import BeautifulSoup
from pprint import pprint
from win10toast import ToastNotifier
```

```
class KunmingWeather:
```

```
    def __init__(self):
```

```
        page = requests.get("http://www.weather.com.cn/weather/101290101.shtml")
        self.soup = BeautifulSoup(page.content, 'html.parser')
```

```

def get_latest_7_days(self):
    weather_kunming_7days = []
    for day in range(7):
        weather_kunming_7days.append({
            "date": self.soup.select("li.sky.skyid")[day].find_all("h1")[0].text,
            "weather": self.soup.select("li.sky.skyid")[day].select('p.wea')[0].text.strip(),
            "temperature": self.soup.select("li.sky.skyid")[day].select('p.tem')[0].text
        })
    return weather_kunming_7days

def get_around(self):
    weather_kunming_around = []
    for index in range(len(self.soup.select("div.aro_city ul li"))):
        liTag = self.soup.select("div.aro_city ul li")[index]
        weather_kunming_around.append({
            "name": liTag.select("span")[0].text,
            "temperature": liTag.select("i")[0].text
        })
    return weather_kunming_around

if __name__ == "__main__":
    kunmingWeather = KunmingWeather()
    weather_latest_7_days = kunmingWeather.get_latest_7_days()
    weather_around = kunmingWeather.get_around()
    pprint(weather_latest_7_days)
    pprint(weather_around)
    # 汇总 7 天的天气情况
    weather_latest_7_days_content = ", ".join(map(lambda weather_info: "日期: {date}, 天气: {weather}, 温度: {temperature}".format(**weather_info), weather_latest_7_days))
    # 汇总周边天气的情况
    weather_around_content = ", ".join(map(lambda weather_info: "地点: {name}, 温度: {temperature}".format(**weather_info), weather_around))
    toaster = ToastNotifier()
    # 汇总整体 Toast 信息
    weather_contents = ""
    昆明最近七天的天气如下:
    {weather_latest_7_days_content}
    昆明周边的天气如下:
    {weather_around_content}
    """.format(weather_latest_7_days_content=weather_latest_7_days_content, weather_around_content=weather_around_content)
    # 显示 Toast 弹窗
    toaster.show_toast("Kunming Weather Notification", weather_contents)

```

运行上面代码的程序，其结果如下图所示：

```
C:\Users\Liu.D.H\python_notebooks\crawler>python C:\Users\Liu.D.H\python_notebooks\crawler\kunming_weather.py
[{'date': '5日(今天)', 'temperature': '\n30/17°C\n', 'weather': '多云'},
 {'date': '6日(明天)', 'temperature': '\n28/17°C\n', 'weather': '多云'},
 {'date': '7日(后天)', 'temperature': '\n28/15°C\n', 'weather': '晴'},
 {'date': '8日(周三)', 'temperature': '\n29/15°C\n', 'weather': '多云转晴'},
 {'date': '9日(周四)', 'temperature': '\n28/15°C\n', 'weather': '晴转多云'},
 {'date': '10日(周五)', 'temperature': '\n27/14°C\n', 'weather': '多云'},
 {'date': '11日(周六)', 'temperature': '\n27/14°C\n', 'weather': '多云'}]
[{'name': '晋宁', 'temperature': '30/16°C'},
 {'name': '黔西南', 'temperature': '25/15°C'},
 {'name': '曲靖', 'temperature': '28/15°C'},
 {'name': '寻甸', 'temperature': '30/15°C'},
 {'name': '安宁', 'temperature': '29/17°C'},
 {'name': '禄劝', 'temperature': '31/15°C'},
 {'name': '嵩明', 'temperature': '29/15°C'},
 {'name': '攀枝花', 'temperature': '36/23°C'},
 {'name': '富民', 'temperature': '31/16°C'},
 {'name': '楚雄', 'temperature': '30/18°C'},
 {'name': '呈贡', 'temperature': '28/17°C'},
 {'name': '玉溪', 'temperature': '30/16°C'}]
C:\Users\Liu.D.H\python_notebooks\crawler>
```

当然要运行次程序还需要运行 `pip install requests beautifulsoup4 win10toast` 安装第三方依赖包。win10toast (<https://pypi.org/project/win10toast/>) 是一个 Windows 10 上的一个显示通知的库，可以很方便的通过 Python 代码在桌面上发送通知。

1.1.3 下载学堂在线网站视频示例

在这个示例中，我们想要下载学堂在线网站上一门已选课程的相关视频，经过对网站网页结构的前期分析可以知道，这个网站在进入视频页面之前是需要登陆的，并且里面的内容是动态生成的，即不能仅仅通过抓取静态的 `html` 文本内容就获取到这些视频链接地址的。所以我们选择使用 `Selenium` 来模拟浏览器加载网页完成，然后解析完成之后的 `html`。`Selenium` 的安装配置这里就不再展开说明了，可以参考其官方文档：<https://www.seleniumhq.org/>。

这个实例仅仅适用于教学示例目的，而且是针对旧版的学堂在线网站。

一些敏感信息及数据是通过环境变量传入的，完整的源码如下：

```
#!/usr/bin/env python
# coding=utf-8

from selenium import webdriver
from selenium.webdriver.support.wait import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.by import By
from selenium.common.exceptions import NoSuchElementException, TimeoutException
from selenium.webdriver.chrome.options import Options
import re
import os
import urllib.request

# construct the regex to normalize filename
regex = re.compile(r"[\s\\\/]+", re.IGNORECASE)
def normalize_filename(filename):
    return regex.sub("_", filename)

def check_exists_by_xpath(driver, xpath):
```

```

    try:
        WebDriverWait(driver, 5).until(EC.visibility_of_element_located((By.XPATH, xpath)))
    except (NoSuchElementException, TimeoutException) as e:
        return False
    return True

def check_exists_by_tag_name(driver, tag_name):
    try:
        WebDriverWait(driver, 5).until(EC.visibility_of_element_located((By.TAG_NAME, tag_name)))
    except (NoSuchElementException, TimeoutException) as e:
        return False
    return True

def get_downloadInfo(courseUrl):
    # create driver
    chrome_options = Options()
    # read HEADLESS environment to detect whether to use --headless argument
    if 'HEADLESS' in os.environ:
        chrome_options.add_argument('--headless')
    driver = webdriver.Chrome(chrome_options=chrome_options)

    # create WebDriverWait instance
    wait = WebDriverWait(driver, 10)
    # open course home page
    driver.get(courseUrl)

    # now in home page
    button = wait.until(EC.element_to_be_clickable((By.ID, 'join_free'))))
    button.click()

    # wait for loading page
    username = wait.until(EC.element_to_be_clickable((By.NAME, 'username'))))
    password = driver.find_element_by_name('password')

    # fill username with USERNAME environment value
    username.send_keys(os.environ['USERNAME'])
    # fill password with USERNAME environment value
    password.send_keys(os.environ['PASSWORD'])
    # login
    password.submit()
    # re-click join_free button
    button = wait.until(EC.element_to_be_clickable((By.LINK_TEXT, '进入课程'))))
    button.click()

```



```

# get all the chapter links
atags = driver.find_elements_by_xpath("//div[@id='accordion']/nav/div/ul/li/a")
atagsUrl = [atag.get_attribute('href') for atag in atags]
downloadInfo = []
for atagUrl in atagsUrl:
    # print(atagUrl)
    driver.get(atagUrl)
    # get all tabs which contains video
    tabs = driver.find_elements_by_xpath("//ol[@id='sequence-list']/li/a")
    for tab in tabs:
        tab.click()
        # check if there is a video tag
        if check_exists_by_tag_name(driver, 'video'):
            # get chapter\subChapter\title information
            chapter = re.split('\s+', driver.find_element_by_xpath("//div[@class='chapter is-open']/h3/a[1]").text)[0]
            subChapter = re.split('\s+', driver.find_element_by_xpath("//div[@class='chapter is-open']/ul/li/a/p[1]").text)[0]
            title = driver.find_element_by_xpath("//div[@class='xblock xblock-student-view xmodule_display xmodule_VideoModule xblock-initialized']/h2").text
            # get video element
            video = driver.find_element_by_tag_name('video')
            downloadInfo.append(("{chapter} {subChapter} {title}".format(chapter=chapter, subChapter=subChapter, title=title), video.get_attribute('src')))
    return downloadInfo

def downloadFile(downloadInfo):
    count = len(downloadInfo)
    for index, entry in enumerate(downloadInfo):
        filename = normalize_filename(entry[0])
        filePath = "videos/{filename}.mp4".format(filename=filename)
        url = entry[1]
        print("{index}/{count} downloading {filename}.mp4".format(index=index, count=count, filename=filename))
        urllib.request.urlretrieve(url, filePath)

if __name__ == "__main__":
    # get course url, if not set in environment then use a default one
    downloadInfo = get_downloadInfo(os.environ['COURSEURL'] if 'COURSEURL' in os.environ else 'http://www.xuetangx.com/courses/course-v1:CAU+08110140_1x+sp/about')
    downloadFile(downloadInfo)

```

1.1.4 Python 处理 excel 数据

在 Python 中对 office 文档的处理还是非常方便的，例如对 word 文档可以使用 python-docx (<https://python-docx.readthedocs.io/en/latest/>) 进行处理，对于 excel 文档可以使用 pyexcel、openpyxl、xlrd、xlwt（可以在这里 <http://www.python-excel.org/> 查看）进行处理，对于 ppt 文档可以使用 python-pptx (<https://python-pptx.readthedocs.io/en/latest/>) 进行处理。

通过这些第三方 office 文档处理包，我们可以很方便的对各种 office 文档进行读取、处理、另存为等操作，我们不用关心这些底层 office 文件的存储结构，我们需要关注的就只是我们的业务逻辑，例如需要读取那里的处理，要进行哪些处理操作，最后是否要保存结果以及结果格式及其内容。

下面我们通过对一个 excel 文档中的成绩进行统计来演示如果对 excel 文档进行处理。

假设原始的 excel 数据文件是 student_scores.xlsx，其内容是：

| name | math_score | english_score |
|------|------------|---------------|
| 张三 | 78 | 85 |
| 李四 | 76 | 87 |
| 王五 | 89 | 91 |

我们想统计数学和英语的平均成绩，经过对 student_scores.xlsx 读取、计算处理之后保存为 summary.xlsx，其内容是：

| average_math_score | average_english_score |
|--------------------|-----------------------|
| 81 | 87.66666667 |

这个简单的示例的完整源码如下：

```
# coding=utf-8

import pyexcel

records = pyexcel.get_records(file_name="student_scores.xlsx")

math_scores=0
english_scores = 0
student_count = 0

for record in records:
    print(type(record))
    math_scores += record['math_score']
    english_scores += record['english_score']
    student_count += 1

average_math_score= math_scores / student_count
average_english_score= english_scores / student_count

data = [['average_math_score', 'average_english_score'], [average_math_score, average_english_score]]

pyexcel.save_as(array=data, dest_file_name="summary.xlsx")
```

可以看到使用 pyexcel 对 excel 文件进行处理是非常简洁高效的，当然要运行次程序还需要运行 `pip install pyexcel pyexcel-xls pyexcel-xl` 安装第三方依赖包。

下载安装 Anaconda

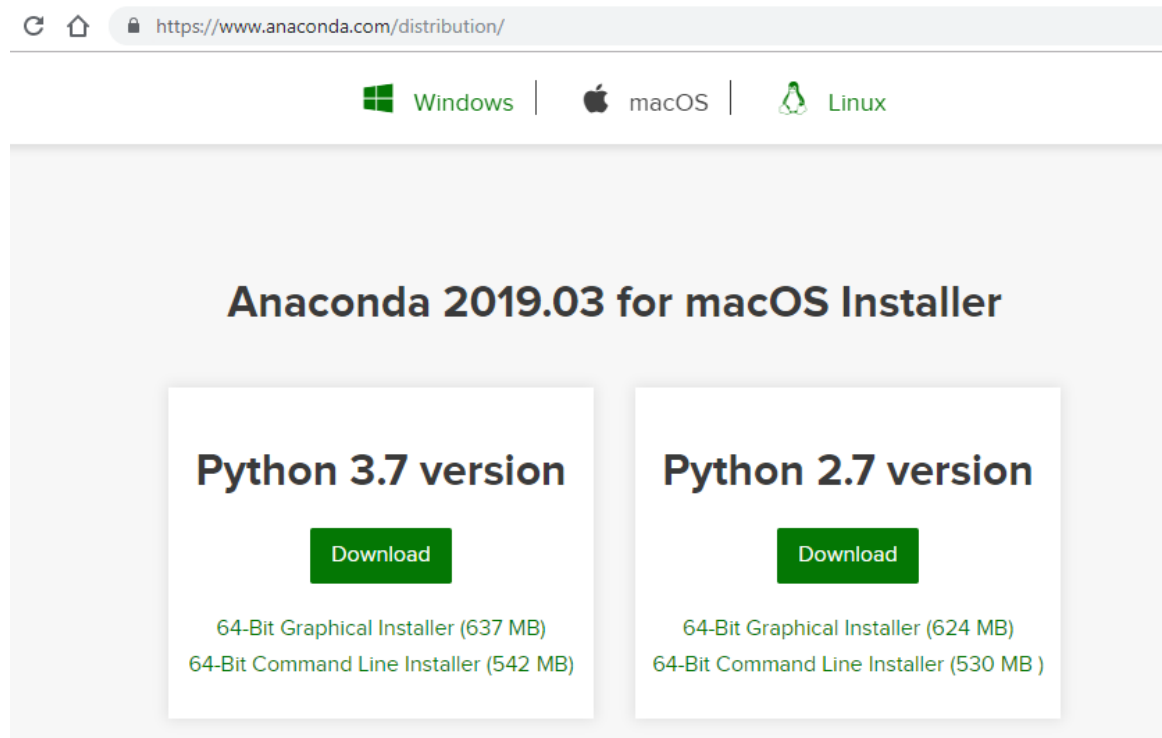
背景

一般使用 Python，可以下载官网（<https://www.python.org/>）的 Python 安装包，但一些常用的第三方库（例如 matplotlib、numpy、scipy、sympy 等）都需要自己使用 pip 去安装，比较麻烦，甚至有时候这些第三方库之间会有版本依赖等问题。Anaconda 很好的解决了这些问题，它集成了常用的 Python 第三方库，主要用于科学计算以及在线图表分析等。

安装图文简易教程

1. 下载

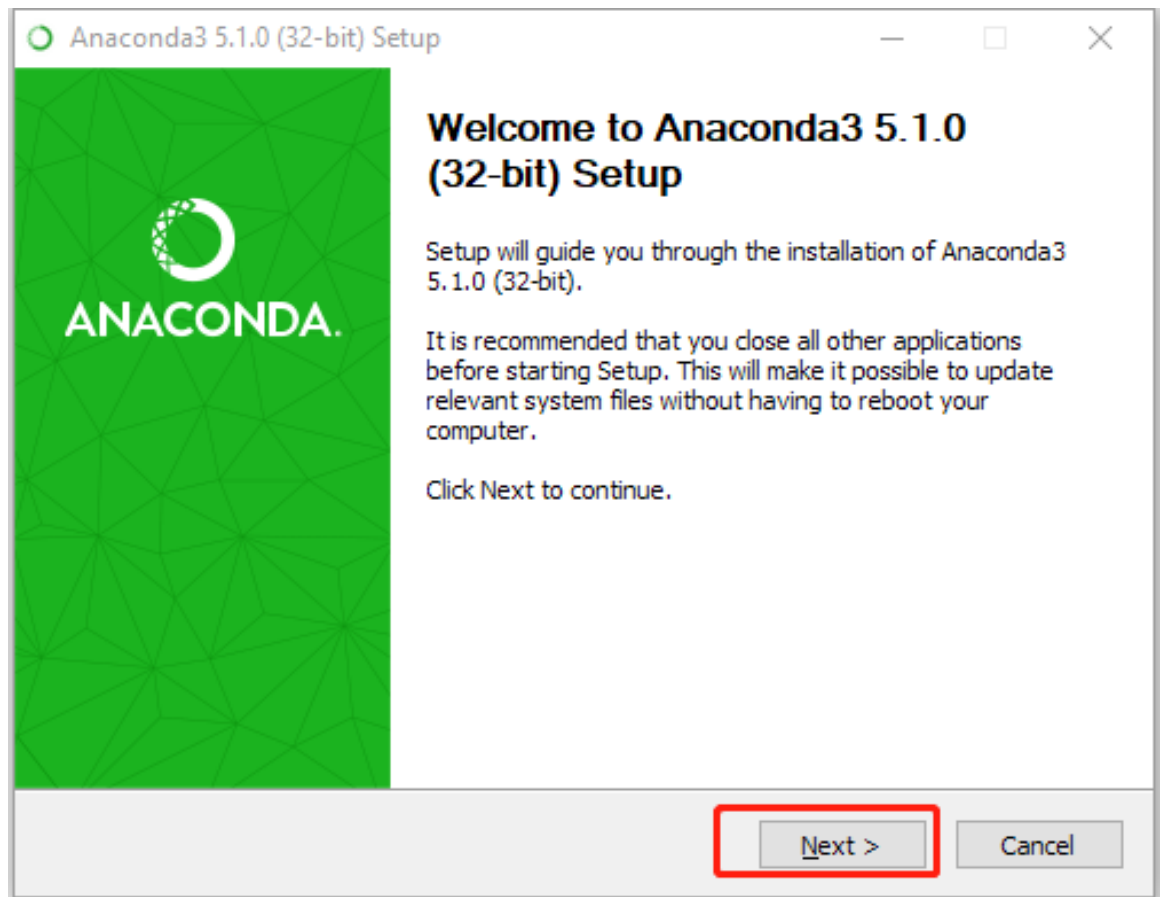
打开浏览器，访问官方网站下载（<https://www.anaconda.com/download/>）。选择对应的操作系统，有 32 位和 64 位，建议安装 32 位即可，32 位的程序在 64 位的系统上也可以正常运行。



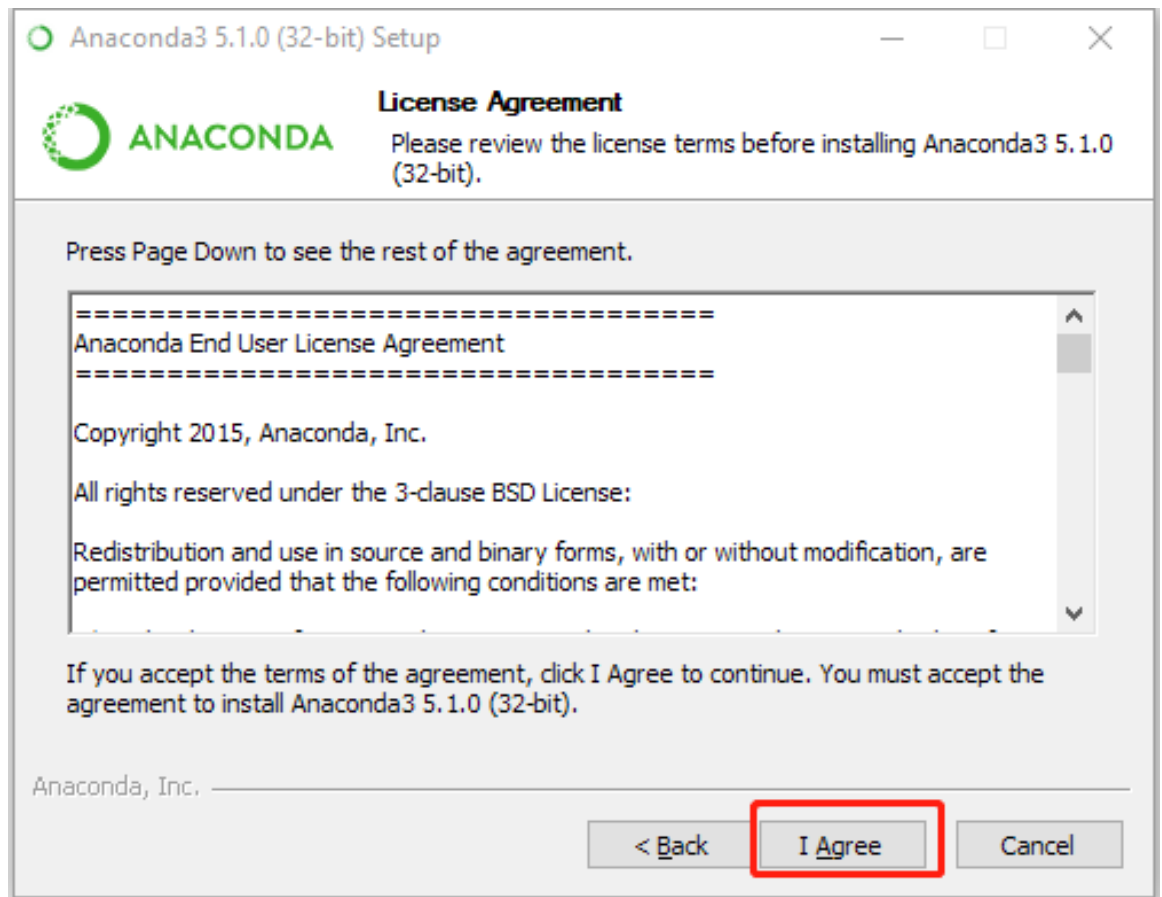
2. 安装

以下以 Windows 为例，安装 32 位 Anaconda，您安装的版本会比以下示例中的新，但基本步骤是一样的。

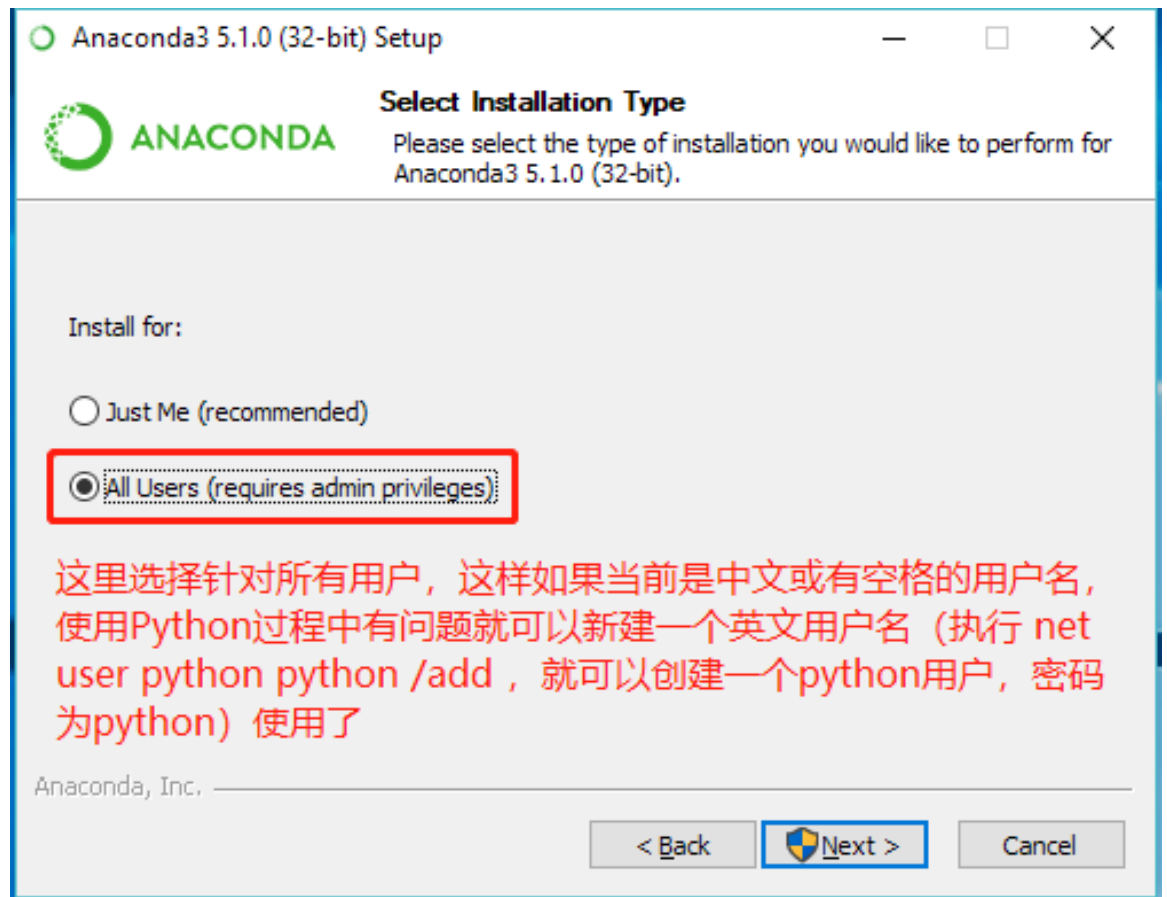
1. 下面是安装的欢迎页面，点击 Next



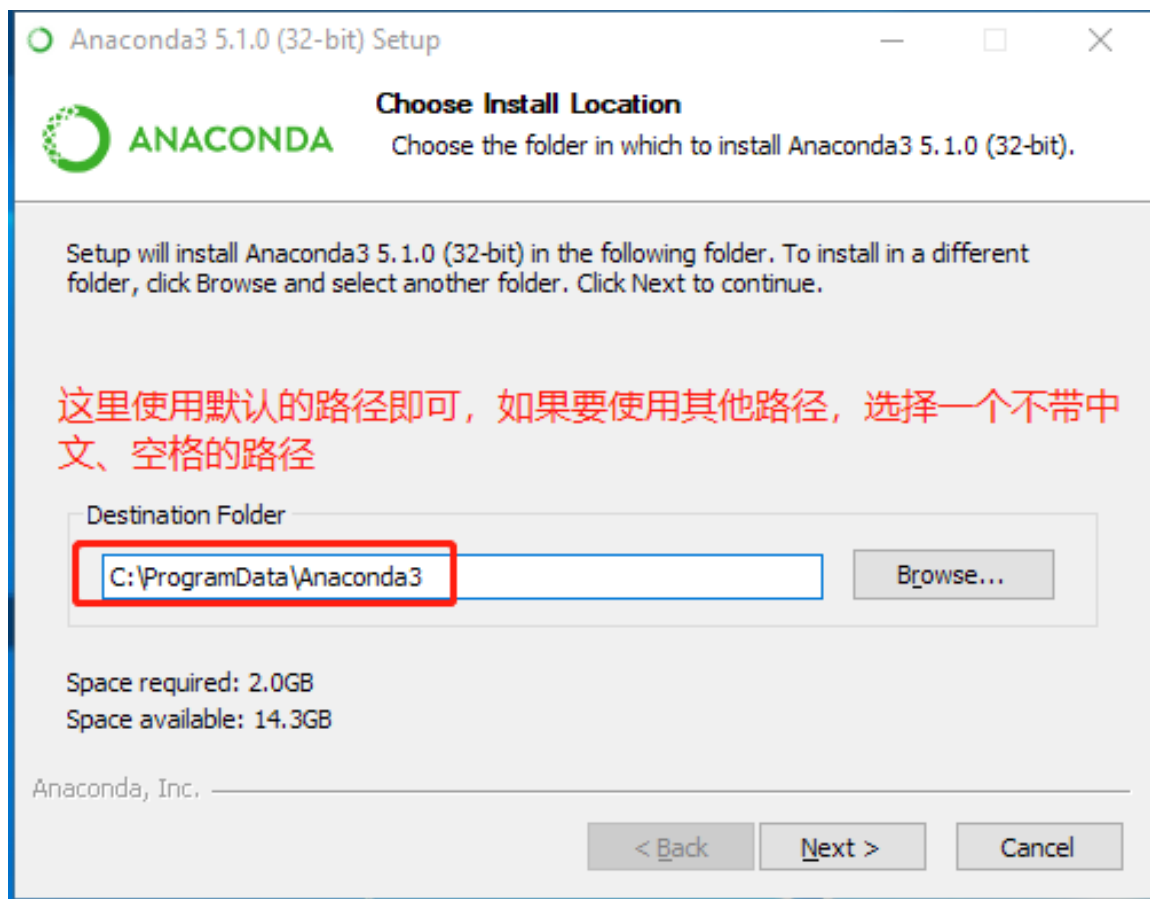
2. 下面是授权同意界面，直接点 Next



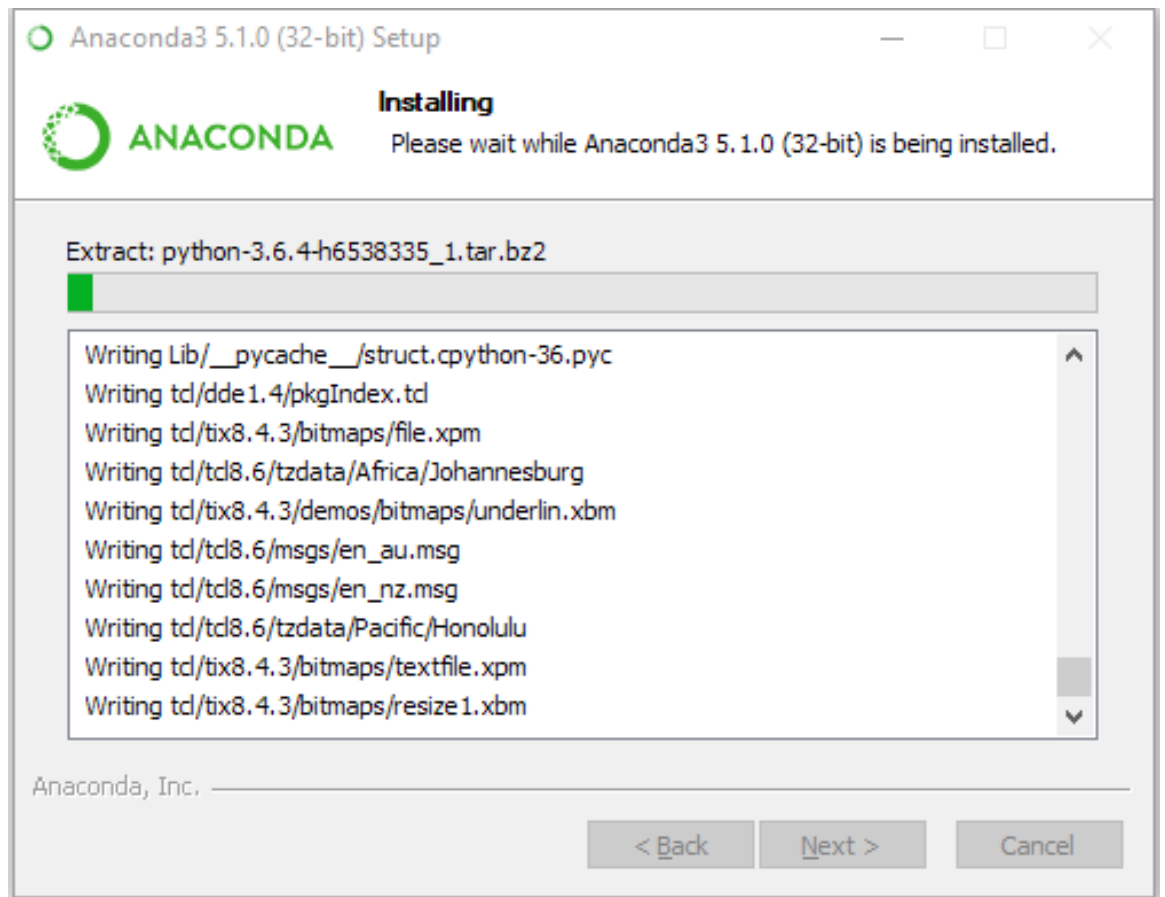
3. 下面是选择 Anaconda 要用仅用于自己还是所有电脑上用户，这里选择第二项针对所有用户



4. 下面选择安装路径，这里需要注意尽量不要安装在中文或者带有空格的路径下，后面容易出现一些奇怪的问题



5. 下面是一些高级选项，建议把第一项 Add Anaconda to my PATH environment variable 选中，这样安装程序自动帮我们把 Anaconda 相关的路径添加到 PATH 环境变量中



8. 下面是安装完的界面

Anaconda3 5.1.0 (32-bit) Setup



ANACONDA

Installation Complete

Setup was completed successfully.

Completed

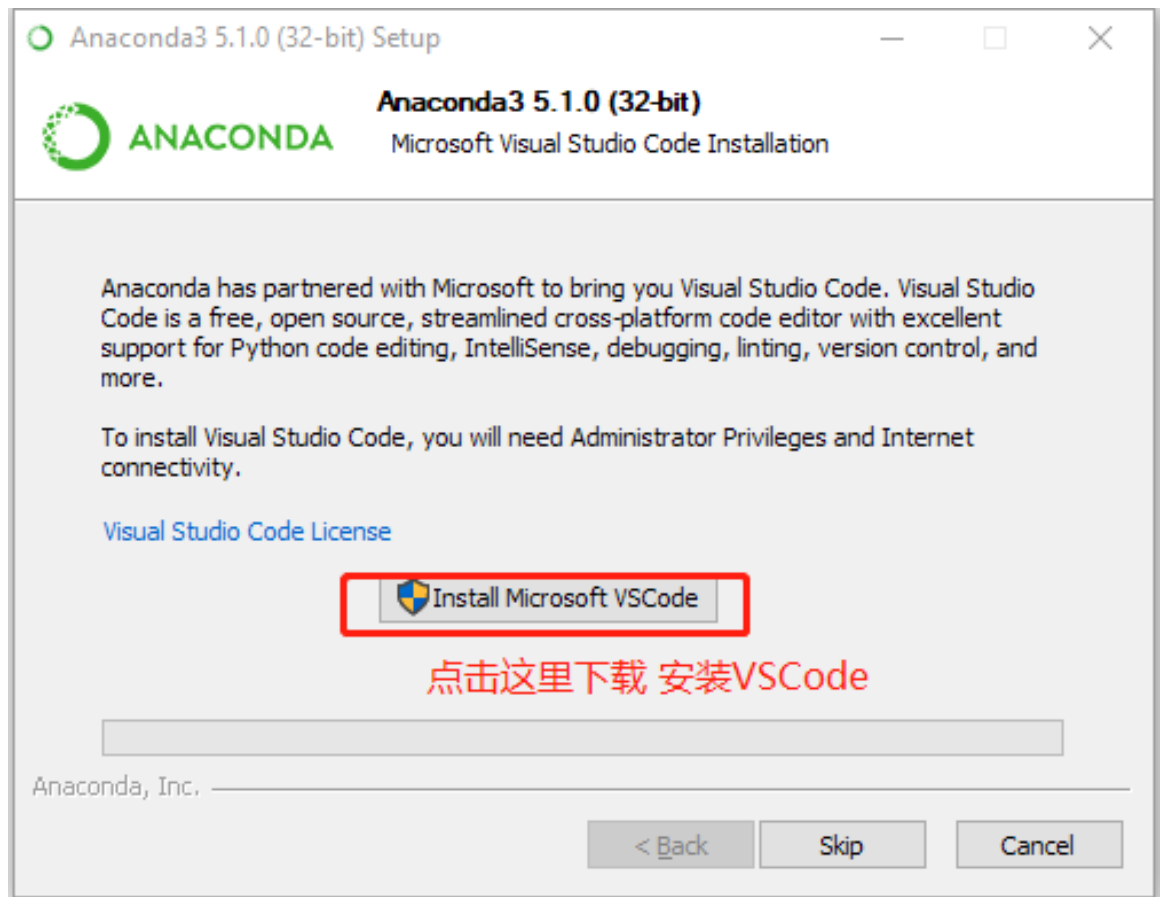
Output folder: C:\Users\Liu.D.H\Anaconda3
Creating Anaconda3 menus...
Execute: "C:\Users\Liu.D.H\Anaconda3\pythonw.exe" -E -s "C:\Users\Liu.D.H\Anaco...
Execute: "C:\Users\Liu.D.H\Anaconda3\pythonw.exe" -E -s "C:\Users\Liu.D.H\Anaco...
Running post install...
Execute: "C:\Users\Liu.D.H\Anaconda3\pythonw.exe" -E -s "C:\Users\Liu.D.H\Anaco...
Execute: "C:\Users\Liu.D.H\Anaconda3\pythonw.exe" -E -s "C:\Users\Liu.D.H\Anaco...
Execute: "C:\Users\Liu.D.H\Anaconda3\pythonw.exe" -E -s "C:\Users\Liu.D.H\Anaco...
Created uninstaller: C:\Users\Liu.D.H\Anaconda3\Uninstall-Anaconda3.exe
Completed

Anaconda, Inc.

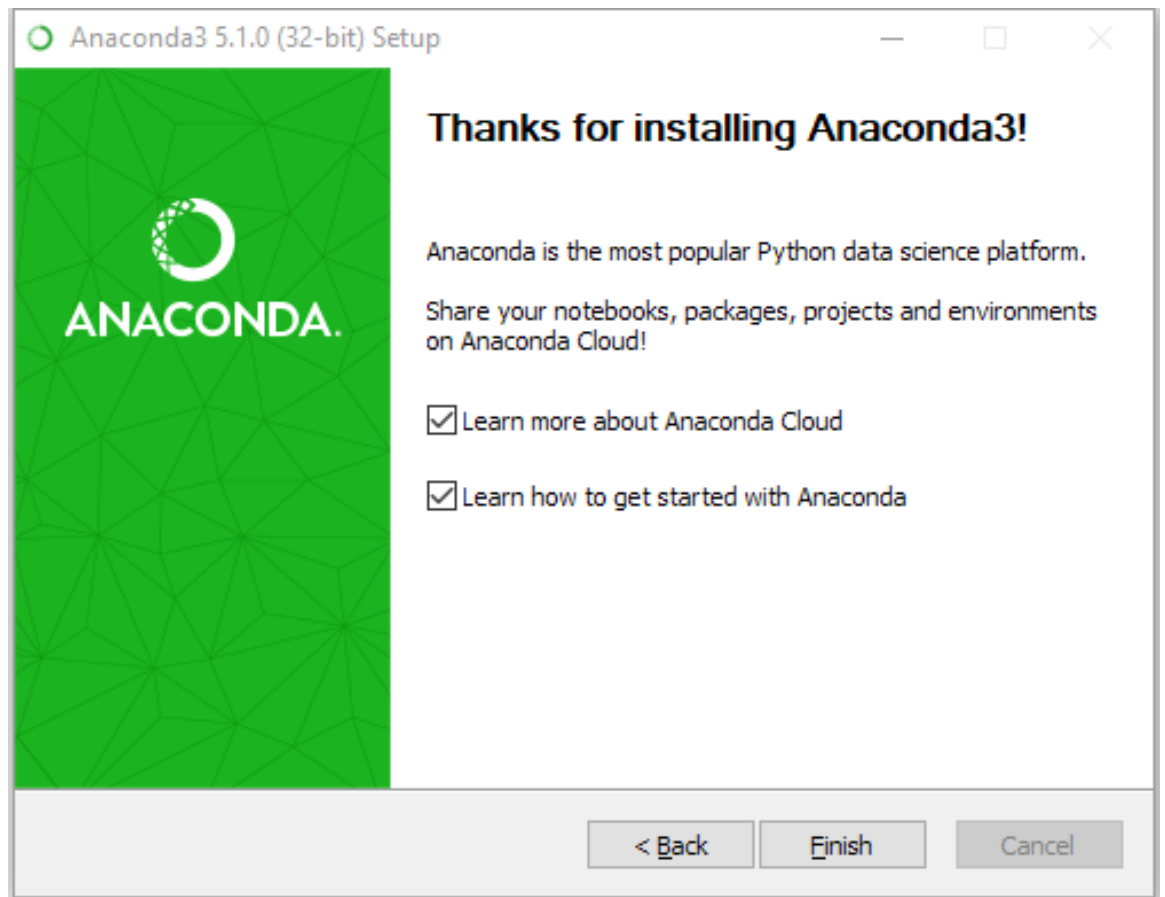
< Back

Next >

Cancel

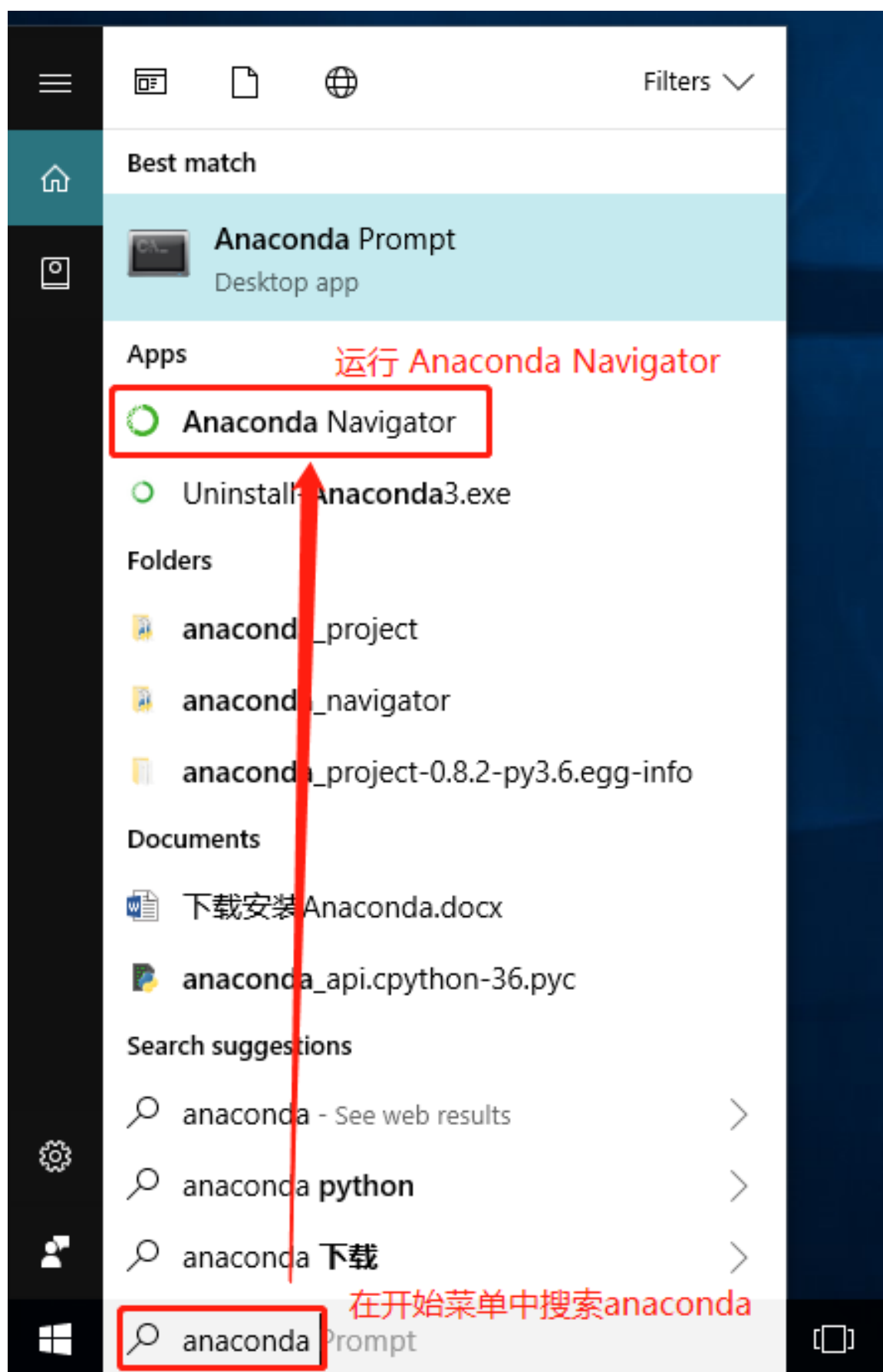


9. 这里也可以选择 **Skip** 跳过安装 VSCode，后面也可以安装，但是需要安装和 Anaconda 相应版本（32 位或 64 位）的 VSCode，这样 Anaconda 里面才能识别到



3. 运行

在开始菜单中搜索 Anaconda 可以找到 Anaconda Navigator（图形界面的 Anaconda）以及 Anaconda Prompt（配置了相关 PATH 的命令行界面）



运行 Anaconda Navigator 之后可以在主界面上看到 Jupyterlab 以及 notebook 等

