

Python编程从入门到实践

第一部分 基础知识

第1章 起步

配置Python环境教程（采用conda 创建虚拟环境 防止环境混乱 因为不同的项目需要涉及不同版本的安装包）：

最新版最详细Anaconda新手安装+配置+环境创建教程 anaconda配置-CSDN博客

VS Code配置Python环境

10分钟搞定！VS Code配置Python开发环境指南（2025新版） - 知乎

可能会出现的问题：

Warning

PS E:\Python编程 从入门到实践> (D:\Anaconda\shell\condabin\conda-hook.ps1) ; (conda activate streamlit)

>>>>>>>>>>>>>>>>> ERROR REPORT <<<<<<<<<<<<<<<<

```
Traceback (most recent call last): File "D:\Anaconda\Lib\site-  
packages\conda\exception_handler.py", line 16, in call return func(*args, **kwargs) ^^^^^^  
File "D:\Anaconda\Lib\site-packages\conda\cli\main.py", line 111, in main_sourced  
print(activator.execute(), end="") UnicodeEncodeError: 'gbk' codec can't encode character  
\u202a' in position 410: illegal multibyte sequence
```

因为系统环境变量被Unicode控制字符污染 可能是之前复制网页中的系统路径时无意中带入

成功截图：

```
example > helloworld.py
```

问题 输出 调试控制台 终端 端口

```
(streamlit) PS E:\Python编程 从入门到实践 & D:\Anaconda\envs\streamlit\python.exe "e:/Python编程 从入门到实践/example/helloWorld.py"
● Hello World!
○ (streamlit) PS E:\Python编程 从入门到实践 >
```

第1章总结到此结束！！！

第2章 变量和简单的数据类型

之后会采用pycharm来进行实例运行 因为用习惯了😊

变量

💡 非常的浅显易懂

```
message = "Hello Python world!"  
print(message)  
message = "Hello Python Crash Course world!"  
print(message)  
#在程序中，可随时修改变量的值，而Python将始终记录变量的最新值
```

```
D:\Anaconda\envs\streamlit\python.exe "E:\Python编程 从入门到实践\example\2. 变量和简单的数据类型.py"  
Hello Python world!  
Hello Python Crash Course world!
```

message只是一个名字 而赋给它的值定义了它的类型 如果是5那就是整型 如果是""那就是字符串

Python是动态类型但**强类型语言**

```
10 + "a" # ✗ 直接报错
```

因此Python的变量不是“容器”，而是“标签” => Python用来做AI是非常合适的 因为AI需要随时调参讲究的是一个动态的变化 模型结构本身就是运行期对象，**语言必须允许结构动态变化**

💡 Tip

变量的命名规范

1. 变量名只能包含字母、数字和下划线。变量名能以字母或下划线开头，但不能以数字开头。

例： message_1 ✅ 1_message ✗

2. 变量名不能包含空格

3. 不要将Python关键字和函数名用作变量名 例如： print、 input等等

⚠ (剩余是工程上的规范 一是变量名应既简短又具有描述性，二是慎用小写字母l和大写字母O，因为它们可能被人错看成数字1和0)

#2.2.2 使用变量时命名错误

```
message = "Hello Python Crash Course reader!"  
print(mesage)
```

```
D:\Anaconda\envs\streamlit\python.exe "E:\Python编程 从入门到实践\example\2. 变量和简单的数据类型.py"  
Traceback (most recent call last):  
  File "E:\Python编程 从入门到实践\example\2. 变量和简单的数据类型.py", line 9, in <module>  
    print(mesage)  
           ^^^^^^  
NameError: name 'mesage' is not defined. Did you mean: 'message'?
```

⌚ Caution

上面的代码块我只截取了相关的部分 具体的整体章节代码在example文件夹下的2.变量和简单的数据类型.py中

通过Python解释器提供的报错信息可以发现是代码的第九行 一般变量xxx未定义的报错都是Python无法识别你提供的变量名 **根本在于使用变量前没有给它赋值** 这里的message可以当作一个新变量

字符串

字符串 (string) 就是一系列字符。在Python 中，用引号引起的都是字符串（可以是单引号也可以是双引号）

比较复杂的字符串

```
text = '''老师说: "今天的作业主题是'Python' 的字符串处理'，请大家认真完成，并在作业中写明: '我已经理解了双引号"和单引号'的区别'。" '''
print(text)

#这里采用了三引号 能够实现以下两个功能（一般用不上） PS: 中文的“和英文的”是有区别的
#1.可以跨行
#2.里面可以随便放'和"

#如果不想用'''三引号却又想在字符串里用"如何解决？
#采用转义字符
message = "The language \"Python\" is named after Monty Python, not the snake."
print(message)
输出: The language "Python" is named after Monty Python, not the snake.
#这里的\就是告诉解释器不需要扫描匹配"字符直接输出=>也就是\"在解释器看来就是直接输出"
```

Important

字符串相关操作函数

1.title()函数

```
# title() 将每个单词的首字母转换为大写
name = "tian lin ying"
print(name.title())

#输出: Tian Lin Ying
```

2.upper()/lower()函数

```
# upper()/lower() 将每个单词大写/小写
name = "Tian Lin Ying"
print(name.upper())
print(name.lower())

#输出: TIAN LIN YING
tian lin ying
```

在字符串中使用变量

```
first_name = "lin ying"
last_name = "tian"
fullname = f"{first_name} {last_name}"
print(fullname)

#输出: lin ying tian
```

Tip

要在字符串中插入变量的值，可先在左引号前加上字母f，再将要插入的变量放在花括号内。

这种字符串称为**f字符串** (format-设置格式) Python通过把花括号内的变量替换为其值来设置字符串的格式

```
first_name = "lin ying"
last_name = "tian"
fullname = f"{first_name} {last_name}"
print(f"Hello, {fullname.title()}!")

#输出: Hello, Lin Ying Tian!
```

也可以使用**f字符串**来创建消息

```
first_name = "lin ying"
last_name = "tian"
fullname = f"{first_name} {last_name}"
message = f"Hello, {fullname.title()}!"
print(message)

#输出: Hello, Lin Ying Tian!
```

#将消息赋给了一个变量 让最后的函数调用print()简单得多

实际案例：

```
out_name = out_nameitems[attribute_box.currentIndex()]
sql_el = textwrap.dedent(f'''
    SELECT
        '当前表' as 表名,
        COUNT(*) as 记录数,
        SUM(a.{attribute_box.currentText()}) as {out_name}总数
    FROM {textbox_30.text()} a
    WHERE a.init_date between 20251001 and 20251031
    UNION ALL
    SELECT
        '历史表' as 表名,
        COUNT(*) as 记录数,
        SUM(a.{attribute_box.currentText()}) as {out_name}总数
    FROM {textbox_20.text()}@uf20_his a
    WHERE a.init_date between 20251001 and 20251031
'''')
```

```
out_sql.setPlainText(sql_e1)
```

这里的代码有一点复杂 因为涉及到了pyqt的知识 可以把花括号里的内容当作变量 因为很多部分基本上差不多的 因此脚本只需要替换变的地方即可=>用变量来实现，最后用f字符串进行拼接

使用制表符或换行符来添加空白

```
#制表符和换行符
print("Python")
print("\tPython")

#输出:
#Python
#      Python
#\t缩进4个字符

print("Languages:\nPython\nC\nJavaScript")

#输出:
#Languages:
#Python
#C
#JavaScript
#\n换行

print("Languages:\n\tPython\n\tC\n\tJavaScript")
#输出:
#Languages:
#      Python
#      C
#      Javascript
```

Important

字符串相关函数

1.rstrip()函数

```
#rstrip() 删除字符串右端空白
favorite_language = ' python '
print(favorite_language.rstrip())

#输出: (此处为空格) python
```

2.lstrip()函数

```
#lstrip() 删除字符串左端空白
favorite_language = ' python '
print(favorite_language.lstrip())

#输出: python (此处为空格)
```

3.strip()函数

```
#strip() 删除字符串两端空白  
favorite_language = ' python '  
print(favorite_language.strip())
```

#输出: python

4.removeprefix()函数

```
#removeprefix() 删除前缀  
nostarch_url = 'https://nostarch.com'  
simple_url = nostarch_url.removeprefix('https://')  
print(simple_url)
```

#输出: nostarch.com

5.removesuffix()函数

```
#removesuffix() 删除后缀  
filename = "python_notes.txt"  
print(filename.removesuffix('.txt'))
```

#输出: python_notes

数

Tip

整数

加减乘除

**表示乘方运算

浮点数

```
print(0.2 + 0.1)  
#输出: 0.30000000000000004
```

#结果包含的小数位数可能是不确定的 所有编程语言都存在这种问题，没有什么可担心的 后续会讲处理多余小数位的方式

整数和浮点数

1.将任意两个数相除，结果总是浮点数

2.只要有操作数是浮点数，默认得到的就是浮点数，即便结果原本为整数

数中的下划线

在书写很大的数时，可使用下划线将其中的位分组，使其更清晰易读

```
universe_age = 14_000_000_000
print(universe_age)

#输出: 140000000000
#在存储这种数时, Python会忽略其中的下划线
```

同时给多个变量赋值

```
x, y, z = 0, 0, 0
```

用逗号将变量名分开; 对于要赋给变量的值, 也需要做同样的处理。Python将按顺序将每个值赋给对应的变量。只要变量数和值的**个数相同**, Python就能正确地将变量和值**关联起来**。

常量

Python没有内置的常量类型, 但Python程序员会使用全大写字母来指出应将某个变量视为常量

```
MAX_CONNECTIONS = 5000
```

注释

在Python中, 注释用井号 (#) 标识。井号后面的内容都会被Python解释器忽略

第3章 列表

3.1 列表介绍

列表 (list) 由一系列按特定顺序排列的元素组成

在Python中, 用方括号 ([]) 表示列表, 用逗号分隔其中的元素。

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles)

#输出: ['trek', 'cannondale', 'redline', 'specialized']
#Python将打印列表的内部表示, 包括方括号
```

💡 Tip

1. 访问列表元素

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0])

#输出: trek
#就当C语言的数组用! 直接访问下标 不同点是C语言的数组是单一数据类型而列表能放很多种数据类型
#例如lists = ['trek', 'cannondale', 'redline', 'specialized', 1]也可行!

#小测试 🐱 (思考一下这个代码输出的结果是啥):
print(bicycles[0].title())
```

2.索引从0而不是1开始

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[1])
print(bicycles[3])

#输出:
#cannondale
#specialized
```

#bingo! 也是跟C语言一模一样

特殊语法

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[-1])
```

```
#输出:
#specialized
```

100 可以实现在不知道列表长度的情况下访问最后的元素 这种语法也适用于其他负数索引
#索引-2返回倒数第二个列表元素，索引-3返回倒数第三个列表元素，依此类推

3.使用列表中的各个值

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
message = f"My first bicycle was a {bicycles[0].title()}."
print(message)

#输出: My first bicycle was a Trek.
```

3.2 修改、添加和删除元素

创建的大多数列表将是动态的，这意味着列表创建后，将随着程序的运行增删元素



1.修改列表元素

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)
motorcycles[0] = 'ducati'
print(motorcycles)

#输出:
#[ 'honda', 'yamaha', 'suzuki']
#[ 'ducati', 'yamaha', 'suzuki']
```

😊 相信学过C语言的你能理解
#你可以修改任意列表元素的值，而不只是第一个元素的值

2.在列表中添加元素

```
#在列表末尾添加元素
```

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)
motorcycles.append('ducati')
print(motorcycles)
```

#输出:
#['honda', 'yamaha', 'suzuki']
#['honda', 'yamaha', 'suzuki', 'ducati']

#append()方法将元素'ducati'添加到列表末尾

```
#append()方法让动态地创建列表易如反掌
motorcycles = []
motorcycles.append('honda')
motorcycles.append('yamaha')
motorcycles.append('suzuki')
print(motorcycles)
```

#输出: ['honda', 'yamaha', 'suzuki']

```
#在列表中插入元素
#使用insert()方法可在列表的任意位置添加新元素
#需要指定新元素的索引和值
motorcycles = ['honda', 'yamaha', 'suzuki']
motorcycles.insert(0, 'ducati')
print(motorcycles)
#输出: ['ducati', 'honda', 'yamaha', 'suzuki']
```

3.从列表中删除元素

```
#使用del语句删除元素
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)
del motorcycles[0]
print(motorcycles)
```

#输出:
#['honda', 'yamaha', 'suzuki']
#['yamaha', 'suzuki']

⌚ #使用del可删除任意位置的列表元素，只需要知道其索引即可

```
#使用pop()方法删除元素
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)
popped_motorcycle = motorcycles.pop()
print(motorcycles)
print(popped_motorcycle)

#输出: ['honda', 'yamaha', 'suzuki']
#[ 'honda', 'yamaha']
#suzuki

#pop()默认删除列表末尾的元素
```

```
#删除列表中任意位置的元素
#使用pop()删除列表中任意位置的元素，只需要在括号中指定要删除的元素的索引即可
motorcycles = ['honda', 'yamaha', 'suzuki']
first_owned = motorcycles.pop(0)
print(f"The first motorcycle I owned was a {first_owned.title()}.")

#输出: The first motorcycle I owned was a Honda.

#⚠ 注意注意:
#!!!每当你使用pop()时，被弹出的元素就不再在列表中了!!!
PS:如何区分使用del还是pop()
需要使用删除的值就用pop();其余两者问题都不大
```

4.根据值删除元素

```
motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
print(motorcycles)
motorcycles.remove('ducati')
print(motorcycles)

#输出:
#[ 'honda', 'yamaha', 'suzuki', 'ducati']
#[ 'honda', 'yamaha', 'suzuki']

PS:
remove()方法只删除第一个指定的值。如果要删除的值可能在列表中出现多次，就需要使用循环
当然remove()函数中的内容可以用变量代替
too_expensive = 'ducati'
motorcycles.remove(too_expensive)
```

3.3 管理列表

① Note

1. 使用sort()方法对列表进行永久排序

```
#sort()方法能永久地修改列表元素的排列顺序
cars = ['bmw', 'audi', 'toyota', 'subaru']
cars.sort() #这里其实默认了reverse=False 等同于用cars.sort(reverse=False)
print(cars)

#输出: ['audi', 'bmw', 'subaru', 'toyota']

cars = ['bmw', 'audi', 'toyota', 'subaru']
cars.sort(reverse=True) #与字母顺序相反的顺序排列列表元素
print(cars)

#输出: ['toyota', 'subaru', 'bmw', 'audi']
```

2. 使用sorted()函数对列表进行临时排序

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
print("Here is the original list:")
print(cars)
print("\nHere is the sorted list:")
print(sorted(cars))
print("\nHere is the original list again:")
print(cars)

#输出:
#Here is the original list:
#[['bmw', 'audi', 'toyota', 'subaru']

#Here is the sorted list:
#[['audi', 'bmw', 'subaru', 'toyota']

#Here is the original list again:
#[['bmw', 'audi', 'toyota', 'subaru']]
```

PS: 通过这个案例可以发现sorted()函数是不会对列表发生改变的

3. 反向打印列表

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
print(cars)
cars.reverse()
print(cars)

#输出:
#[['bmw', 'audi', 'toyota', 'subaru']
#[['subaru', 'toyota', 'audi', 'bmw']]
```

PS: reverse()方法会永久地修改列表元素的排列顺序，但可随时恢复到原来的排列顺序，只需对列表再次调用reverse()即可
且reverse()不是按与字母顺序相反的顺序排列列表元素，只是反转列表元素的排列顺序

4. 确定列表的长度

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
print(len(cars))
```

#输出: 4

第4章 操作列表

4.1 遍历整个列表

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(magician)
```

#输出:

```
#alice
#david
#carolina
```

💡 Tip

刚开始使用循环时请牢记，不管列表包含多少个元素，每个元素都将被执行循环指定的步骤。如果列表包含100万个元素，Python就将重复执行指定的步骤100万次，而且通常速度非常快。

#在for循环中执行更多的操作

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
```

#输出:

```
#Alice, that was a great trick!
#David, that was a great trick!
#Carolina, that was a great trick!
```

4.2 避免缩进错误

❗ Caution

注意循环语句的缩进问题

#1. 忘记缩进

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(magician)
```

```
D:\Anaconda\envs\streamlit\python.exe "E:\Python编程 从入门到实践\example\4.操作列表.py"
  File "E:\Python编程 从入门到实践\example\4.操作列表.py", line 14
      print(magician)
      ^
IndentationError: expected an indented block after 'for' statement on line 13
```

#2. 不必要的缩进

```
message = "Hello Python world!"  
print(message)
```

```
D:\Anaconda\envs\streamlit\python.exe "E:\Python编程 从入门到实践\example\4.操作列表.py"  
File "E:\Python编程 从入门到实践\example\4.操作列表.py", line 18  
    print(message)  
IndentationError: unexpected indent
```

#3. 遗漏冒号

```
magicians = ['alice', 'david', 'carolina']  
for magician in magicians  
    print(magician)
```

#这里需要注意的是Python的循环或者条件语句基本上都会用到：get到这个点就好了！

```
D:\Anaconda\envs\streamlit\python.exe "E:\Python编程 从入门到实践\example\4.操作列表.py"  
File "E:\Python编程 从入门到实践\example\4.操作列表.py", line 22  
    for magician in magicians  
    ^  
SyntaxError: expected ':'
```

4.3 创建数值列表

Important

1. 使用range()函数

```
for value in range(1, 5):  
    print(value)
```

#输出：
#1
#2
#3
#4

#看到这个结果可能有些疑惑

#range()函数让Python从指定的第一个值开始数，并在到达指定的第二个值时停止

PS：可以理解为高中数学集合的左闭右开 😊

#要打印数1~5，需要使用range(1, 6)

```
for value in range(1, 6):  
    print(value)
```

2. 使用range()创建数值列表

```
numbers = list(range(1, 6))  
print(numbers)
```

#输出：[1, 2, 3, 4, 5]

PS：这里的list()函数相当于直接将range(1, 6)强制转换成列表对象 可以类比成C语言中的强制类型转换(int)'123'

```
!!! 补充!!!
r = range(1, 6)
print(r)
print(type(r))
```

```
#输出:
#range(1, 6)
#<class 'range'>
```

这里可以发现`range`也是一个数据类型 `range`对象采用惰性求值的策略，它只存储生成序列的起始值、终止值和步长，以及计算下一个值的方法。只有当你真正需要用到序列中的数字时（例如在`for`循环中迭代），它才会临时计算并提供给你。

[TOP](#) 所以最上面那个例子的`numbers`需要进行`list()`函数进行类型转换才能逐个显示1~5的数

```
#在使用range()函数时，还可指定步长 给这个函数指定第三个参数，Python将根据这个步长来生成数
even_numbers = list(range(2, 11, 2))
print(even_numbers)
```

```
#输出: [2, 4, 6, 8, 10]
```

在这个示例中，`range()`函数从2开始数，然后不断地加2，直到达到或超过终值（11）

```
#小案例
#生成前10个整数的平方
squares = []
for value in range(1, 11):
    square = value ** 2
    squares.append(square)
print(squares)
```

```
#输出: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

3.对数值列表执行简单的统计计算

```
digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
print(min(digits))
print(max(digits))
print(sum(digits))
```

```
#输出:
```

```
#0
```

```
#9
```

```
#45
```

4.列表推导式

```
squares = [value**2 for value in range(1, 11)]
print(squares)
```

```
#输出: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

#列表推导式（list comprehension）将`for`循环和创建新元素的代码合并成一行，并自动追加新元素。

要使用这种语法，首先指定一个描述性的列表名，如`squares`。然后指定一个左方括号，并定义一个表达式，用于生成要存储到列表中的值。在这个示例中，表达式为`value**2`，它计算平方值。接下来，编写一个`for`循环，用于给表达式提供值，再加上右方括号。在这个示例中，`for`循环为`for value in range(1, 11)`，它将值1~10提供给表达式`value**2`。注意，这里的`for`语句末尾没有冒号。

① Note

🐶 接下来考考你(重点看看Text6和Text7)

👉 答案在这: [0基础入门Python-小测试](#)

Test1

使用一个for循环打印数1~20 (含)

Test2

创建一个包含数1~1000000的列表，再使用一个for循环将这些数打印出来。（如果输出的时间太长，按 Ctrl+C停止输出，或关闭输出窗口。）

Text3

创建一个包含数1~1000000的列表，再使用min()和max()核实该列表确实是从1开始、到1000000结束的。另外，对这个列表调用函数 sum()，看看Python将100万个数相加需要多长时间。

Text4

通过给range()函数指定第三个参数来创建一个列表，其中包含1~20的奇数；再使用一个for循环将这些数打印出来。

Text5

创建一个列表，其中包含3~30内能被3整除的数，再使用一个for循环将这个列表中的数打印出来。

Text6

将同一个数乘三次称为立方。例如，在Python中，2的立方用`2**3`表示。创建一个列表，其中包含前10个整数（1~10）的立方，再使用一个for循环将这些立方数打印出来。

Text7

使用列表推导式生成一个列表，其中包含前10个整数的立方。

4.4 使用列表的一部分

切片

要创建切片，可指定要使用的第一个元素和最后一个元素的索引。与range()函数一样，Python在到达指定的第二个索引之前的元素时停止。（同样可以理解为左闭右开）

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[0:3])
#输出: ['charles', 'martina', 'michael']

players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[1:4])
```

```
#输出: ['martina', 'michael', 'florence']
```

如果没有指定第一个索引，自动从列表开头开始

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
```

```
print(players[:4])
```

```
#输出: ['charles', 'martina', 'michael', 'florence']
```

要让切片终止于列表末尾，也可使用类似的语法

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
```

```
print(players[2:])
```

```
#输出: ['michael', 'florence', 'eli']
```

负数索引返回与列表末尾有相应距离的元素，因此可以输出列表末尾的任意切片

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
```

```
print(players[-3:])
```

```
#输出: ['michael', 'florence', 'eli']
```

遍历切片

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
```

```
print("Here are the first three players on my team:")
```

```
for player in players[:3]:
```

```
    print(player.title())
```

```
#输出:
```

```
Here are the first three players on my team:
```

```
Charles
```

```
Martina
```

```
Michael
```

💡 !!!只需要记住列表的切片还是列表 只不过内容是列表的部分

复制列表

Tip

```
my_foods = ['pizza', 'falafel', 'carrot cake']  
friend_foods = my_foods[:]
```

```
print("My favorite foods are:")  
print(my_foods)
```

```
print("\nMy friend's favorite foods are:")  
print(friend_foods)
```

```
#输出:
```

```
My favorite foods are:
```

```
['pizza', 'falafel', 'carrot cake']
```

```
My friend's favorite foods are:
```

```
['pizza', 'falafel', 'carrot cake']
```

#为了核实确实有两个列表，下面在每个列表中都添加一种食品，并确认每个列表都记录了相应的人喜欢的食品

```
my_foods.append('cannoli')
```

```
friend_foods.append('ice cream')

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)

#输出:
My favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli']
My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake', 'ice cream']
```

!!!容易犯的误区

```
my_foods = ['pizza', 'falafel', 'carrot cake']

#这是行不通的:
friend_foods = my_foods

my_foods.append('cannoli')
friend_foods.append('ice cream')
print("My favorite foods are:")
print(my_foods)
print("\nMy friend's favorite foods are:")
print(friend_foods)

#输出:
My favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']
My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']
```

这里将`my_foods`赋给`friend_foods`, 而不是将`my_foods`的副本赋给`friend_foods`。这种语法实际上让 Python 将新变量`friend_foods`关联到已与`my_foods`相关联的列表, 因此这两个变量指向同一个列表。

因此创建副本需要`friend_foods = my_foods[:]`

① Note

⌚ 小小的练习一下

使用切片 来打印列表中间的三个元素

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']

#我一猜你会用players[1:4]
#想想如何不用指定的下标访问呢？如果列表很长的话难不成要一个一个数过去？

#bingo！想到用len()函数来获取列表长度
n = len(players)
print(players[int((n - 1) / 2) - 1:int((n - 1) / 2) + 2])
```

PS：这里用了int()强制类型转换 想想为什么？

第3章数的部分提到过将任意两个数相除，结果总是浮点数 那么我们切片的索引应该是整数吧=>所以需要进行类型转换

还有要注意我们切片的范围，左闭右开噢！

4.5 元组

Python将不能修改的值称为**不可变的**，而不可变的列表称为**元组** (tuple)

定义元组

元组看起来很像列表，但使用圆括号而不是方括号来标识。定义元组后，就可使用索引来访问其元素，就像访问列表元素一样。

```
#如果有一个大小不应改变的矩形，可将其长度和宽度存储在一个元组中，从而确保它们是不能修改的
dimensions = (200, 50)
print(dimensions[0])
print(dimensions[1])
#输出：
200
50
```

尝试修改元组dimensions的一个元素，看看结果如何：

```
dimensions = (200, 50)
dimensions[0] = 250
```

```
D:\Anaconda\envs\streamlit\python.exe "E:\Python编程 从入门到实践\example\4.操作列表.py"
Traceback (most recent call last):
  File "E:\Python编程 从入门到实践\example\4.操作列表.py", line 108, in <module>
    dimensions[0] = 250
~~~~~^~~^
TypeError: 'tuple' object does not support item assignment
```

在代码试图修改矩形的尺寸时，Python会报错。

⚠ Caution

严格地说，元组是由逗号标识的，圆括号只是让元组看起来更整洁、更清晰。如果你要定义只包含一个元素的元组，必须在这个元素后面加上逗号

```
my_t = (3)
for item in my_t:
    print(item)
```

```
D:\Anaconda\envs\streamlit\python.exe "E:\Python编程 从入门到实践\example\4.操作列表.py"
Traceback (most recent call last):
  File "E:\Python编程 从入门到实践\example\4.操作列表.py", line 112, in <module>
    for item in my_t:
TypeError: 'int' object is not iterable
```

解释器会把不加逗号的元组当成单个元素它本身的数据类型来看 而不是当元组 因此无法进行遍历

```
my_t = (3,)
for item in my_t:
    print(item)
print(type(my_t))

#输出:
#3
#<class 'tuple'>
```

加了逗号就会当作元组来看

遍历元组

```
dimensions = (200, 50)
for dimension in dimensions:
    print(dimension)

#输出:
200
50
```

修改元组变量

```
dimensions = (200, 50)
print("Original dimensions:")
for dimension in dimensions:
    print(dimension)

dimensions = (400, 100)
print("\nModified dimensions:")
for dimension in dimensions:
    print(dimension)

#输出:
original dimensions:
200
50
Modified dimensions:
400
100
```

相当于将一个新元组关联到变量**dimensions** 并没有改变原先元组中的元素 因此合法

总结：

相比于列表，元组是更简单的数据结构。如果需要存储一组在程序的整个生命周期内都**不变的值**，就可以使用元组=>**不可改变的列表+定义采用的是圆括号**

💡 Important

补充一个知识点方便写代码

如果用pycharm的话它自带自动缩进：Ctrl+Alt+L

列表的内容到此结束咯！✿

第5章 if语句

ⓘ Note

很多例子是书上的 我觉得有一丝繁琐 能get到点即可

5.1 一个简单的示例

```
# 一个简单的示例
cars = ['audi', 'bmw', 'subaru', 'toyota']
for car in cars:
    if car == 'bmw':
        print(car.upper())
    else:
        print(car.title())

#输出:
Audi
BMW
Subaru
Toyota
```

5.2 条件测试

检查是否相等

```
car = 'bmw'
print(car == 'bmw')

#输出: True
```

如何在检查是否相等时忽略大小写

```
#不管car里是什么 调用lower()全部小写进行比较 而且lower()方法并没有影响存储在变量car中的值
```

```
car = 'Audi'  
print(car.lower() == 'audi')  
  
#输出: True
```

检查是否不等

```
requested_topping = 'mushrooms'  
if requested_topping != 'anchovies':  
    print("Hold the anchovies!")  
  
#输出: Hold the anchovies!
```

数值比较

```
answer = 17  
if answer != 42:  
    print("That is not the correct answer. Please try again!")  
  
#输出: That is not the correct answer. Please try again!
```

检查多个条件

```
#使用and检查多个条件  
age_0 = 22  
age_1 = 18  
print(age_0 >= 21 and age_1 >= 21)  
  
#输出: False
```

💡 Tip

为了改善可读性，可将每个条件测试都分别放在一对括号内，但并非必须这样做。如果使用括号，条件测试将类似于下面这样：

```
(age_0 >= 21) and (age_1 >= 21)
```

```
#使用or检查多个条件  
age_0 = 22  
age_1 = 18  
print(age_0 >= 21 or age_1 >= 21)  
  
#输出: True
```

检查特定的值是否在列表中

```
requested_toppings = ['mushrooms', 'onions', 'pineapple']
print('mushrooms' in requested_toppings)
print('pepperoni' in requested_toppings)

#输出:
True
False
```

检查特定的值是否不在列表中

```
banned_users = ['andrew', 'carolina', 'david']
user = 'marie'
if user not in banned_users:
    print(f"{user.title()}, you can post a response if you wish.")

#输出: Marie, you can post a response if you wish.
```

布尔表达式

布尔表达式不过是条件测试的别名罢了。与条件表达式一样，布尔表达式的结果要么为**True**，要么为**False**。

布尔值通常用于**记录条件** 如游戏是否正在运行或用户是否可以编辑网站的特定内容

```
game_active = True
can_edit = False
```

5.3 if语句

简单的if语句

```
age = 19
if age >= 18:
    print("You are old enough to vote!")

#输出: You are old enough to vote!
```

if-else语句

```
age = 17
if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
else:
    print("Sorry, you are too young to vote.")
    print("Please register to vote as soon as you turn 18!")

#输出:
#Sorry, you are too young to vote.
#Please register to vote as soon as you turn 18!
```

if-elif-else语句

```
age = 12
if age < 4:
    print("Your admission cost is $0.")
elif age < 18:
    print("Your admission cost is $25.")
else:
    print("Your admission cost is $40.")

#输出: Your admission cost is $25.

#简洁版:
age = 12
if age < 4:
    price = 0
elif age < 18:
    price = 25
else:
    price = 40
print(f"Your admission cost is ${price}.")
```

使用多个elif代码块

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
elif age < 65:
    price = 40
else:
    price = 20

print(f"Your admission cost is ${price}.")

#输出: Your admission cost is $25.
```

省略else代码块

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
elif age < 65:
    price = 40
elif age >= 65:
    price = 20

print(f"Your admission cost is ${price}.")
```

测试多个条件

```

requested_toppings = ['mushrooms', 'extra cheese']
if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
if 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
if 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")
print("\nFinished making your pizza!")

#输出:
#Adding mushrooms.
#Adding extra cheese.

#Finished making your pizza!

```

如果像下面这样转而使用if-elif-else语句，代码将不能正确运行，因为只要有一个条件测试通过，就会跳过余下的条件测试

```

requested_toppings = ['mushrooms', 'extra cheese']

if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
elif 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
elif 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")

print("\nFinished making your pizza!")

#输出:
#Adding mushrooms.

#Finished making your pizza!

```

总之，如果只想运行一个代码块，就使用if-elif-else语句

如果要运行多个代码块，就使用一系列独立的if语句

5.4 使用if语句处理列表

检查特殊元素

```

#循环语句+条件判断

requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
    if requested_topping == 'green peppers':
        print("Sorry, we are out of green peppers right now.")
    else:
        print(f"Adding {requested_topping}.")

print("\nFinished making your pizza!")

```

```
#输出:  
#Adding mushrooms.  
#Sorry, we are out of green peppers right now.  
#Adding extra cheese.  
  
#Finished making your pizza!
```

确定列表非空

```
if requested_toppings:  
    for requested_topping in requested_toppings:  
        print(f"Adding {requested_topping}.")  
    print("\nFinished making your pizza!")  
else:  
    print("Are you sure you want a plain pizza?")
```

#输出: Are you sure you want a plain pizza?

使用多个列表

```
available_toppings = ['mushrooms', 'olives', 'green peppers',  
                      'pepperoni', 'pineapple', 'extra cheese']  
requested_toppings = ['mushrooms', 'french fries', 'extra cheese']  
for requested_topping in requested_toppings:  
    if requested_topping in available_toppings:  
        print(f"Adding {requested_topping}.")  
    else:  
        print(f"Sorry, we don't have {requested_topping}.")  
  
print("\nFinished making your pizza!")
```

#输出:
#Adding mushrooms.
#Sorry, we don't have french fries.
#Adding extra cheese.
#Finished making your pizza!

① Note

小补充 虽然可能用不上

```
#or和and的短路问题  
is_admin = True  
  
if(is_admin or print("显示管理员面板")==None):  
    print(1)  
  
#输出: 1  
这里的is_admin已经是True了 就不会再执行后面的条件 这就是经典的or的短路问题  
  
is_admin = False  
if(is_admin and print("显示管理员面板")==None):
```

```
print(1)
```

#无输出

这里的`is_admin = False` 整个`and`语句就不可能再会是`True` 就不会再执行后面的条件 直接跳过`if`语句块 这就是经典的`and`的短路问题

第5章到此结束！ 

第6章 字典

理解字典后，你就能够更准确地为各种真实物体建模。你可以创建一个表示人的字典，然后在其中存储你想存储的任何信息：姓名、年龄、地址，以及可以描述这个人的任何其他方面。

6.1 一个简单的字典

```
alien_0 = {'color': 'green', 'points': 5}
```

```
print(alien_0['color'])  
print(alien_0['points'])
```

#输出：

```
green  
5
```

6.2 使用字典

在Python中，字典（dictionary）是一系列键值对。每个键都与一个值关联，可以使用键来访问与之关联的值。

键值对包含两个相互关联的值。当你指定键时，Python将返回与之关联的值。键和值之间用冒号分隔，而键值对之间用逗号分隔。

访问字典中的值

```
alien_0 = {'color': 'green'}  
print(alien_0['color'])
```

#输出： green

```
alien_0 = {'color': 'green', 'points': 5}  
  
new_points = alien_0['points']  
print(f"You just earned {new_points} points!")  
  
#输出： You just earned 5 points!
```

添加键值对

字典是一种动态结构，可随时在其中添加键值对。要添加键值对，可依次指定字典名、用方括号括起来的键和与该键关联的值。

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

alien_0['x_position'] = 0 #添加键值对操作
alien_0['y_position'] = 25 #添加键值对操作
print(alien_0)

#输出:
{'color': 'green', 'points': 5}
{'color': 'green', 'points': 5, 'x_position': 0, 'y_position': 25}
```

从创建一个空字典开始

```
alien_0 = {} #可以类比于创建一个新列表 list=[] 后面再进行list.append()操作进行添加元素

alien_0['color'] = 'green'
alien_0['points'] = 5

print(alien_0)
#输出:
{'color': 'green', 'points': 5}
```

修改字典中的值

```
alien_0 = {'color': 'green'}
print(f"The alien is {alien_0['color']}.")

alien_0['color'] = 'yellow'
print(f"The alien is now {alien_0['color']}.")

#输出:
The alien is green.
The alien is now yellow.
```

删除键值对

对于字典中不再需要的信息，可使用del语句将相应的键值对彻底删除。在使用del语句时，必须指定字典名和要删除的键。

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

del alien_0['points']
print(alien_0)

#输出:
{'color': 'green', 'points': 5}
{'color': 'green'}
```

由类似对象组成的字典（这个能看懂代码就行 感觉书中这个小标题很多余）

```
favorite_languages = {  
    'jen': 'python',  
    'sarah': 'c',  
    'edward': 'rust',  
    'phil': 'python', #这里在最后一个键值对后面也加上逗号，为以后添加键值对做好准备  
}  
  
language = favorite_languages['sarah'].title()  
print(f"Sarah's favorite language is {language}.")  
  
#输出: Sarah's favorite language is C.
```

使用get()来访问值

```
alien_0 = {'color': 'green', 'speed': 'slow'}  
print(alien_0['points'])
```

这里可以发现alien_0并没有points这个键 那么肯定会有问题

```
Traceback (most recent call last):  
  File "E:\Python编程_从入门到实践\example\6.字典.py", line 60, in <module>  
    print(alien_0['points'])  
    ~~~~~~^~~~~~  
KeyError: 'points'
```

为了避免出现上述的问题 我们可以采用get()的方法来访问值

```
alien_0 = {'color': 'green', 'speed': 'slow'}  
  
point_value = alien_0.get('points', 'No point value assigned.')  
print(point_value)  
  
#输出: No point value assigned.
```

如果指定的键有可能不存在，应考虑使用get()方法，而不要使用方括号表示法

① Note

在调用get()时，如果没有指定第二个参数且指定的键不存在，Python将返回值None，这个特殊的值表示没有相应的值 因此用get()比用方括号好！

```
alien_0 = {'color': 'green', 'speed': 'slow'}  
point_value = alien_0.get('points')  
print(point_value)  
  
#输出: None
```

6.3 遍历字典

鉴于字典可能包含大量数据，Python支持对字典进行遍历。字典可用于以各种方式存储信息，因此有多种遍历方式：既可遍历字典的所有键值对，也可只遍历键或值。

遍历所有的键值对

```
user_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
}

for key, value in user_0.items():
    print(f"\nKey: {key}")
    print(f"Value: {value}")
```

#输出:

```
Key: username
Value: efermi
```

```
Key: first
Value: enrico
```

```
Key: last
Value: fermi
```

#上述的循环写法并不是唯一的

```
for a, b in user_0.items():
    print(f"\nKey: {a}")
    print(f"Value: {b}")
```

运行上述循环的结果也是一样的 key和value只不过是两个变量罢了！

再举一个例子

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'rust',
    'phil': 'python',
}

for name, language in favorite_languages.items():
    print(f"{name.title()}s favorite language is {language.title()}.")
```

#输出:

```
Jen's favorite language is Python.
Sarah's favorite language is C.
Edward's favorite language is Rust.
Phil's favorite language is Python.
```

#看完这个例子应该对遍历键值对有了一定的理解

遍历字典中的所有键

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'rust',
    'phil': 'python',
}
```

```
for name in favorite_languages.keys(): #这里的keys()就不能被替换成其他的了  
                                    #因为这是字典对象的一个属性!  
    print(name.title())
```

#输出:

```
Jen  
Sarah  
Edward  
Phil
```

当然了！在遍历字典时，会默认遍历所有的键 如果将上述代码中的

`for name in favorite_languages.keys():`替换成`for name in favorite_languages:`也是可以的！

输出将不变嗷！！！

#只不过显式地使用`keys()`方法能让代码的可读性更好一些！



再来个书上的例子

#能看懂这段代码在干什么你就掌握了！

```
favorite_languages = {  
    'jen': 'python',  
    'sarah': 'c',  
    'edward': 'rust',  
    'phil': 'python',  
}  
  
friends = ['phil', 'sarah']  
for name in favorite_languages.keys():  
    print(f"Hi {name.title()}")  
  
    if name in friends:  
        language = favorite_languages[name].title()  
        print(f"\t{name.title()}, I see you love {language}!")  
  
#输出:  
Hi Jen.  
Hi Sarah.  
Sarah, I see you love C!  
Hi Edward.  
Hi Phil.  
Phil, I see you love Python!
```

按特定的顺序遍历字典中的所有键



💡看到这里了该考考你了

`sort()`和`sorted()`在列表排序中的区别是什么？相信聪明的你是知道的呦！

```
favorite_languages = {  
    'jen': 'python',  
    'sarah': 'c',  
    'edward': 'rust',  
    'phil': 'python',  
}  
  
for name in sorted(favorite_languages.keys()):  
    print(f"\n{name.title()}, thank you for taking the poll.")
```

遍历字典中的所有值

如果你感兴趣的是字典包含的值，可使用**values()**方法。它会返回一个**值列表**，不包含任何键。

```
favorite_languages = {  
    'jen': 'python',  
    'sarah': 'c',  
    'edward': 'rust',  
    'phil': 'python',  
}  
  
print("The following languages have been mentioned:")  
for language in favorite_languages.values():  
    print(language.title())  
  
#输出:  
The following languages have been mentioned:  
Python  
C  
Rust  
Python
```

那么问题来了？如果字典的值包含很多重复项我该如何剔除重复项捏？办法就是采用**集合(set)**

```
favorite_languages = {  
    'jen': 'python',  
    'sarah': 'c',  
    'edward': 'rust',  
    'phil': 'python',  
}  
  
print("The following languages have been mentioned:")  
for language in set(favorite_languages.values()):  
    print(language.title())  
  
#输出:  
The following languages have been mentioned:  
Python  
C  
Rust
```

Note

💡 `favorite_languages.values()`返回的值列表跟列表还是有一些区别的

```
print(type(favorite_languages.values()))
#输出: <class 'dict_values'>
```

首先数据类型就不是列表list 其次它无法用索引来进行访问，只能允许遍历

```
values = favorite_languages.values()
print(values[0])
```

```
D:\Anaconda\envs\streamlit\python.exe "E:\Python编程 从入门到实践\example\6.字典.py"
Traceback (most recent call last):
  File "E:\Python编程 从入门到实践\example\6.字典.py", line 128, in <module>
    print(values[0])
    ~~~~~~^~~
TypeError: 'dict_values' object is not subscriptable
```

⚠ Caution

集合和字典很容易混淆，因为它们都是用一对花括号定义的。当花括号内**没有键值对**时，定义的很可能是**集合**。不同于列表和字典，集合不会以特定的顺序存储元素。

```
languages = {'python', 'rust', 'python', 'c'}
print(languages)

#输出: {'rust', 'python', 'c'}
```

6.4 嵌套

有时候，需要将多个字典存储在列表中或将列表作为值存储在字典中，这称为**嵌套**。

字典列表

字典alien_0包含一个外星人的各种信息，但无法存储第二个外星人的信息，更别说屏幕上全部外星人的信息了。如何管理成群结队的外星人呢？一种办法是创建一个外星人列表，其中每个外星人都是一个字典，包含有关该外星人的各种信息。

```
alien_0 = {'color': 'green', 'points': 5}
alien_1 = {'color': 'yellow', 'points': 10}
alien_2 = {'color': 'red', 'points': 15}

aliens = [alien_0, alien_1, alien_2]

for alien in aliens:
    print(alien)

#输出:
{'color': 'green', 'points': 5}
{'color': 'yellow', 'points': 10}
{'color': 'red', 'points': 15}
```

更符合现实的情形是，外星人不止三个，而且每个外星人都是用代码自动生成的。在下面的示例中，使用range()生成了30个外星人

```
#创建一个用于存储外星人的空列表

aliens = []

#创建30个绿色的外星人

for alien_number in range(30): #! ! ! #循环遍历0-29 共30次
    new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
    aliens.append(new_alien)

for alien in aliens[:5]: #! ! ! #遍历列表 不指明开始索引的话默认是从0开始的
    #因此是0 1 2 3 4 左闭右开
    print(alien)
print("...")

#显示创建了多少个外星人

print(f"Total number of aliens: {len(aliens)}")

#输出:
{'color': 'green', 'points': 5, 'speed': 'slow'}
...

```

Total number of aliens: 30

① Note

再来一个书上的例子

将前三个外星人修改为黄色、速度中等且值10分

```
#创建一个用于存储外星人的空列表

aliens = []

#创建30个绿色的外星人

for alien_number in range(30):
    new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
    aliens.append(new_alien)

for alien in aliens[:3]:
    if alien['color'] == 'green':
        alien['color'] = 'yellow'
        alien['speed'] = 'medium'
        alien['points'] = 10

#显示前5个外星人
for alien in aliens[:5]:
    print(alien)
print("...")
```

```
#输出:  
{'color': 'yellow', 'points': 10, 'speed': 'medium'}  
{'color': 'yellow', 'points': 10, 'speed': 'medium'}  
{'color': 'yellow', 'points': 10, 'speed': 'medium'}  
{'color': 'green', 'points': 5, 'speed': 'slow'}  
{'color': 'green', 'points': 5, 'speed': 'slow'}  
...
```

在字典中存储列表

```
#存储顾客所点比萨的信息  
  
pizza = {  
    'crust': 'thick',  
    'toppings': ['mushrooms', 'extra cheese'],  
}  
  
# 概述顾客点的比萨  
print(f"You ordered a {pizza['crust']}-crust pizza "  
      "with the following toppings:")  
  
for topping in pizza['toppings']:  
    print(f"\t{topping}")  
  
#输出:  
You ordered a thick-crust pizza with the following toppings:  
    mushrooms  
    extra cheese
```

💡 Tip

再来一个书上的例子(能看懂代码即可)

```
favorite_languages = {  
    'jen': ['python', 'rust'],  
    'sarah': ['c'],  
    'edward': ['rust', 'go'],  
    'phil': ['python', 'haskell'],  
}  
  
for name, languages in favorite_languages.items():  
    print(f"\n{name.title()}'s favorite languages are:")  
    for language in languages:  
        print(f"\t{language.title()}")  
  
#输出:  
Jen's favorite languages are:  
    Python  
    Rust  
  
Sarah's favorite languages are:  
    C
```

Edward's favorite languages are:

Rust

Go

Phil's favorite languages are:

Python

Haskell

在字典中存储字典

```
users = {  
    'aeinstein': {  
        'first': 'albert',  
        'last': 'einstein',  
        'location': 'princeton',  
    },  
  
    'mcurie': {  
        'first': 'marie',  
        'last': 'curie',  
        'location': 'paris',  
    },  
  
}  
  
for username, user_info in users.items():  
    print(f"\nUsername: {username}")  
    full_name = f"{user_info['first']} {user_info['last']}"  
    location = user_info['location']  
  
    print(f"\tFull name: {full_name.title()}")  
    print(f"\tLocation: {location.title()}")
```

第6章到此结束啦！ 🍓

第7章 用户输入和while循环

7.1 input()函数的工作原理

```
message = input("Tell me something, and I will repeat it back to you: ")  
print(message)
```

这里函数接受一个参数，即要向用户显示的提示

当运行Python的第一行代码时，用户将看到提示“Tell me something, and I will repeat it back to you:”

这时候程序等待用户输入，并在用户按回车键后继续运行。用户的输入被赋给变量message，接下来的print将输入呈现给用户

```
Tell me something, and I will repeat it back to you: Hello everyone!
Hello everyone!
```

用int()来获取数值的输入

```
age = input("How old are you?")
#模拟终端输入 <<<How old are you?21
print(age>=18)
```

```
Traceback (most recent call last):
  File "E:\Python编程 从入门到实践\example\7.用户输入和while循环.py", line 3, in <module>
    print(age>=18)
    ^^^^^^^^
TypeError: '>=' not supported between instances of 'str' and 'int'
```

① Note

这里可以发现我们input的输入是字符串类型

```
age = input("How old are you?")
#模拟终端输入 <<<How old are you?21
age = int(age)
print(age>=18)

#输出:True
```

求模运算符

求模运算符（%）是个很有用的工具，它将两个数相除并返回余数

```
print(4%3)
#输出:1
```

7.2 while循环简介

使用while循环

举一个小例子

```
#使用while循环来数数
current_number = 1
while current_number <= 5:
    print(current_number)
    current_number += 1

#输出:
1
2
3
4
5
```

让用户选择何时退出

```
message = ""  
while message != 'quit':  
    message = input(prompt)  
    print(message)  
  
#输出:  
Hello everyone!  
Hello everyone!  
  
Hello again.  
Hello again.  
  
quit  
quit
```

程序执行流程:

```
1.message = "" message != 'quit'  
2.message = input(prompt) message="Hello everyone!" print(message)  
3.message="Hello everyone!" message != 'quit'  
4.message = input(prompt) message="Hello again." print(message)  
5.message="Hello again." message != 'quit'  
6.message = input(prompt) message="quit" print(message)  
7.message="quit" message == 'quit'=>退出
```

#这个程序很好，唯一美中不足的是，它将单词'quit'也作为一条消息打印了出来。为了修复这种问题，只需要使用一个简单的if测试

```
message = ""  
while message != 'quit':  
    message = input(prompt)  
  
    if message != 'quit':  
        print(message)
```

使用标志

这个应该学过C语言会有印象。在要求满足很多条件才继续运行的程序中，可定义一个变量，用于判断整个程序是否处于活动状态。这个变量称为标志（flag），充当程序的交通信号灯。

```
#通过使用标志可以让上述程序更为简洁易懂  
active = True  
while active:  
    message = input()  
    if message == 'quit':  
        active = False  
    else:  
        print(message)
```

使用break退出循环

如果不管条件测试的结果如何，想立即退出while循环，不再运行循环中余下的代码，可使用break语句。

再来看一个例子

```
prompt = "\nPlease enter the name of a city you have visited:"
```

```
prompt += "\n(Enter 'quit' when you are finished.) "
while True:
    city = input(prompt)
    if city == 'quit':
        break
    else:
        print(f"I'd love to go to {city.title()}!")

#输出:
Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) New York
I'd love to go to New York!
Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) San Francisco
I'd love to go to San Francisco!
Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) quit
```

在循环中使用continue

要返回循环开头，并根据条件测试的结果决定是否继续执行循环，可使用continue语句，它不像break语句那样不再执行余下的代码并退出整个循环。

可以举一个1-10奇数循环的例子

```
current_number = 0
while current_number < 10:
    current_number += 1
    if current_number % 2 == 0:
        continue
    print(current_number)

#输出:
1
3
5
7
9
```

避免无限循环

这个是我个人认为写程序最重要的地方 一个死循环程序在实际应用中可能自己都检查不出来 就是因为自己在**写循环条件和条件状态改变时**没有注意到这个点

每个while循环都必须有结束运行的途径，这样才不会没完没了地执行下去。

```
x = 1
while x <= 5:
    print(x)
    x += 1

#加入忘记了写x += 1
#这个循环将没完没了地运行！

x = 1
while x <= 5:
    print(x)
```

每个程序员都会偶尔不小心地编写出无限循环，在循环的退出条件比较微妙时尤其如此。如果程序陷入无限循环，既可按Ctrl+C，也可关闭显示程序输出的终端窗口。

这段话我觉得书上说的很好

💡 Tip

要避免编写无限循环，务必对每个while循环进行测试，确保它们按预期那样结束。如果希望程序在用户输入特定值时结束，可运行程序并输入该值。如果程序在这种情况下没有结束，请检查程序处理这个值的方式，确认程序至少有一个地方导致循环条件为False或导致break语句得以执行。

7.3 使用while循环处理列表和字典

以下的例子能看懂代码就行 实际案例到时遇到个人觉得是能够写出来的

在列表之间移动元素

假设有一个列表包含新注册但还未验证的网站用户。验证这些用户后，如何将他们移到已验证用户列表中呢？一种办法是使用一个while循环，在验证用户的同时将其从未验证用户列表中提取出来，再将其加入已验证用户列表。

```
unconfirmed_users = ['alice', 'brian', 'candace']
confirmed_users = []

#将每个经过验证的户都移到已验证用户列表中

while unconfirmed_users:
    current_user = unconfirmed_users.pop()
    print(f"Verifying user: {current_user.title()}")
    confirmed_users.append(current_user)

print("\nThe following users have been confirmed:")
for confirmed_user in confirmed_users:
    print(confirmed_user.title())
```

删除为特定值的所有列表元素

```

pets = ['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
print(pets)
while 'cat' in pets:
    pets.remove('cat')
print(pets)

#输出:
['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
['dog', 'dog', 'goldfish', 'rabbit']

```

使用用户输入填充字典

```

responses = {}

polling_active = True

while polling_active:
    name = input("\nwhat is your name? ")
    response = input("which mountain would you like to climb someday?")

    #将回答存储在字典中
    responses[name] = response

    #看看是否还有人要参与调查

    repeat = input("Would you like to let another person respond? (yes/no) ")
    if repeat == 'no':
        polling_active = False

#调查结束，显示结果
print("\n--- Poll Results ---")
for name, response in responses.items():
    print(f"{name} would like to climb {response}.")

#输出:
what is your name? Eric
which mountain would you like to climb someday? Denali
Would you like to let another person respond? (yes/no) yes
what is your name? Lynn
which mountain would you like to climb someday? Devil's Thumb
Would you like to let another person respond? (yes/no) no --- Poll Results ---
Eric would like to climb Denali.
Lynn would like to climb Devil's Thumb.

```

Note

来个小测试

做个猜数字小游戏

提示：使用`random.randint(a,b)`实现模拟随机数 在a,b之间包括a,b取一个随机数(通过时间戳)

猜数字小游戏

第8章 函数

8.1 定义函数

第一行代码使用关键字def来告诉Python，你要定义一个函数。这是函数定义，向Python指出了函数名，还可以在括号内指出函数为完成任务需要什么样的信息(也就是C语言的传参，只不过python不需要定义数据类型只需要变量名)。

```
def greet_user():
    """显示简单的问候语"""
    print("Hello!")

greet_user()
```

向函数传递信息

```
def greet_user(username):
    """显示简单的问候语"""
    print(f"Hello, {username.title()}!")

greet_user('jesse')
```

实参和形参

ⓘ Note

在greet_user()函数的定义中，变量username是一个(parameter)，即函数完成工作所需的信息。在代码greet_user('jesse')中，值'jesse'是一个形参实参(argument)，即在调用函数时传递给函数的信息。在调用函数时，我们将要让函数使用的信息放在括号内。在greet_user('jesse')这个示例中，我们将实参'jesse'传递给函数greet_user()，这个值被赋给了形参username。

8.2 传递实参

函数定义中可能包含多个形参，因此函数调用中也可能包含多个实参。向函数传递实参的方式很多：既可以使用位置实参，这要求实参的顺序与形参的顺序相同；也可以使用关键字实参，其中每个实参都由变量名和值组成；还可以使用列表和字典。

位置实参

在调用函数时，Python必须将函数调用中的每个实参关联到函数定义中的一个形参。最简单的方式是基于实参的顺序进行关联。以这种方式关联的实参称为**位置实参**。

```
def describe_pet(animal_type, pet_name):
    """显示宠物的信息"""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")
describe_pet('hamster', 'harry')

#输出:
I have a hamster.
My hamster's name is Harry.
```

💡 Tip

调用函数多次

```
def describe_pet(animal_type, pet_name):
    """显示宠物的信息"""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")
describe_pet('hamster', 'harry')
describe_pet('dog', 'willie')

#输出:
I have a hamster.
My hamster's name is Harry.
I have a dog.
My dog's name is willie.
```

多次调用同一个函数是一种效率极高的工作方式。(个人觉得是函数的一个非常重要的作用-简化重复的代码)

位置实参的顺序很重要

```
#当使用位置实参来调用函数时, 如果实参的顺序不正确, 结果可能会出乎意料
def describe_pet(animal_type, pet_name):
    """显示宠物的信息"""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")
describe_pet('harry', 'hamster') #这边把实参的位置反了, 因此会得到不一样的结果

#输出:
I have a harry.
My harry's name is Hamster.
```

关键字实参

关键字实参是传递给函数的名值对。这样会直接在实参中将名称和值关联起来，因此向函数传递实参时就不会混淆了。

```
def describe_pet(animal_type, pet_name):
    """显示宠物的信息"""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet(pet_name='harry', animal_type='hamster')

#现在即使将参数顺序反了 结果依然是正确的逻辑 可以认为是一一对应
```

默认值

在编写函数时，可以给每个形参指定默认值。如果在调用函数中给形参提供了实参，Python将使用指定的实参值；否则，将使用形参的默认值。

```
def describe_pet(pet_name, animal_type='dog'):
    """显示宠物的信息"""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.)
```

```
describe_pet(pet_name='willie')

#输出:
I have a dog.
My dog's name is willie.

describe_pet(pet_name='harry', animal_type='hamster')
#输出:
I have a harry.
My harry's name is Hamster.
```

等效的函数调用

鉴于可混合使用位置实参、关键字实参和默认值，通常有多种等效的函数调用方式。请看`describe_pet()`函数的如下定义，其中给一个形参提供了默认值：

```
def describe_pet(pet_name, animal_type='dog'):
```

```
#一条名为willie的小狗
describe_pet('willie')
describe_pet(pet_name='willie')

#一只名为Harry的仓鼠
describe_pet('harry', 'hamster')
describe_pet(pet_name='harry', animal_type='hamster')
describe_pet(animal_type='hamster', pet_name='harry')
```

PS：可以认为是不同的调用函数方式但输出的结果是一样的=>等效的函数调用

避免实参错误

等你开始使用函数后，也许会遇到实参不匹配错误。当你提供的实参多于或少于函数完成工作所需的实参数量时，将出现实参不匹配错误。

```
def describe_pet(animal_type, pet_name):
    """显示宠物的信息"""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")\n\n

describe_pet()
```

Python发现该函数调用缺少必要的信息，并用**traceback**指出了这一点：

```
D:\Anaconda\envs\streamlit\python.exe "E:\Python编程 从入门到实践\example\8.函数.py"
Traceback (most recent call last):
  File "E:\Python编程 从入门到实践\example\8.函数.py", line 59, in <module>
    describe_pet()
TypeError: describe_pet() missing 2 required positional arguments: 'animal_type' and 'pet_name'
```

进程已结束，退出代码为 1

Note

traceback首先指出问题出在什么地方(line 59)，让我们能够回过头去找出函数调用中的错误。然后，指出导致问题的函数调用 (describe_pet()函数有问题)。最后，traceback指出该函数调用缺少两个实参，并指出了相应形参的名称(animal_type和pet_name)。

8.3 返回值

函数并非总是直接显示输出，它还可以处理一些数据，并返回一个或一组值。函数返回的值称为返回值。

上述的描述后续在学numpy和pandas会理解的非常清楚

返回简单的值

```
def get_formatted_name(first_name, last_name):
    full_name = f"{first_name} {last_name}"
    return full_name.title()

musician = get_formatted_name('jimi', 'hendrix')
print(musician)

#输出:
Jimi Hendrix
```

① Note

```
#原本只需编写下面的代码就可以输出这个标准格式的姓名
print("Jimi Hendrix")
```

似乎是不是觉得很复杂

但其实并不是

如果需要存储大量名和姓的大型程序中。像get_formatted_name()这样的函数非常有用。你可以分别存储名和姓，每当需要显示姓名时就调用这个函数。

让实参变成可选的

假设要扩展get_formatted_name() 函数，使其除了名和姓之外还可以处理中间名。为此，可将其修改成类似这样：

```
def get_formatted_name(first_name, middle_name, last_name):
    full_name = f"{first_name} {middle_name} {last_name}"
    return full_name.title()

musician = get_formatted_name('john', 'lee', 'hooker')
print(musician)
```

然而，并非所有人都有中间名。如果调用这个函数时只提供了名和姓，它将不能正确地运行。为让中间名变成可选的，可给形参middle_name指定默认值（空字符串），在用户不提供中间名时不使用这个形参。

```
def get_formatted_name(first_name, last_name, middle_name=''):
    if middle_name:
```

```
        full_name = f"{first_name} {middle_name} {last_name}"
    else:
        full_name = f"{first_name} {last_name}"

    return full_name.title()

musician = get_formatted_name('jimi', 'hendrix')
print(musician)
musician = get_formatted_name('john', 'hooker', 'lee')
print(musician)

#输出:
Jimi Hendrix
John Lee Hooker

#这个修改后的版本不仅适用于只有名和姓的人，也适用于还有中间名的人
```

返回字典

函数可返回任何类型的值，包括列表和字典等较为复杂的数据结构。

```
def build_person(first_name, last_name):
    person = {'first': first_name, 'last': last_name}
    return person

musician = build_person('jimi', 'hendrix')
print(musician)

#输出: {'first': 'jimi', 'last': 'hendrix'}
```

这个函数接受简单的文本信息，并将其放在一个更合适的数据结构中，让你不仅能打印这些信息，还能以其他方式处理它们。当前，字符串'jimi'和'hendrix'分别被标记为名和姓。你可以轻松地扩展这个函数，使其接受可选值，如中间名、年龄、职业或其他任何要存储的信息。例如，下面修改能让你存储年龄：

```
def build_person(first_name, last_name, age=None):
    person = {'first': first_name, 'last': last_name}
    if age:
        person['age'] = age
    return person

musician = build_person('jimi', 'hendrix', age=27)
print(musician)
```

在函数定义中，新增了一个可选形参age，其默认值被设置为特殊值None（表示变量没有值）。可将None视为占位值。在条件测试中，None相当于False。如果函数调用中包含形参age的值，这个值将被存储到字典中。

结合使用函数和while循环

```
def get_formatted_name(first_name, last_name):
    full_name = f"{first_name} {last_name}"
    return full_name.title()

while True:
    print("\nPlease tell me your name:")
    f_name = input("First name: ")
    l_name = input("Last name: ")

    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nHello, {formatted_name}!")
```

⚠ Warning

这个就是非常经典的死循环

原因在于没有退出条件

我们要让用户能够尽可能容易地退出，因此在每次提示用户输入时，都应提供退出途径。使用break语句可以在每次提示用户输入时提供退出循环的简单途径：

```
def get_formatted_name(first_name, last_name):
    full_name = f"{first_name} {last_name}"
    return full_name.title()

while True:
    print("\nPlease tell me your name:")
    print("(enter 'q' at any time to quit)")

    f_name = input("First name: ")
    if f_name == 'q':
        break

    l_name = input("Last name: ")
    if l_name == 'q':
        break

    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nHello, {formatted_name}!")
```

#输出：

```
Please tell me your name:
(enter 'q' at any time to quit)
First name: eric #输入
Last name: matthes #输入
```

Hello, Eric Matthes! #输出

```
Please tell me your name:
(enter 'q' at any time to quit)
First name: q #输入
```

8.4 传递列表

你经常会发现，向函数传递列表很有用，可能是名字列表、数值列表或更复杂的对象列表（如字典）。将列表传递给函数后，函数就能直接访问其内容。

```
def greet_users(names):
    for name in names:
        msg = f"Hello, {name.title()}!"
        print(msg)

usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)

#输出:
Hello, Hannah!
Hello, Ty!
Hello, Margot!
```

在函数中修改列表

将列表传递给函数后，函数就可以对其进行修改了。在函数中对这个列表所做的任何修改都是永久的，这让你能够高效地处理大量数据。（在做数据分析的时候非常重要）

来看一家为用户提交的设计制作3D打印模型的公司。需要打印的设计事先存储在一个列表中，打印后将被移到另一个列表中。下面是在不使用函数的情况下模拟这个过程的代码：

```
#首先创建一个列表，其中包含一些要打印的设计
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

#模拟打印每个设计，直到没有未打印的设计为止
#打印每个设计后，都将其移到列表completed_models中

while unprinted_designs:
    current_design = unprinted_designs.pop()
    print(f"Printing model: {current_design}")
    completed_models.append(current_design)

#显示打印好的所有模型
print("\nThe following models have been printed:")
for completed_model in completed_models:
    print(completed_model)

#输出:
Printing model: dodecahedron
Printing model: robot pendant
Printing model: phone case

The following models have been printed:
dodecahedron
robot pendant
phone case
```

接下来就是使用函数来实现

```

def print_models(unprinted_designs, completed_models):
    """
    模拟打印每个设计，直到没有未打印的设计为止。打印每个设计后，都将其移到列表
    completed_models中
    """

    while unprinted_designs:
        current_design = unprinted_designs.pop()
        print(f"Printing model: {current_design}")
        completed_models.append(current_design)

def show_completed_models(completed_models):
    """
    显示打印好的所有模型
    """
    print("\nThe following models have been printed:")
    for completed_model in completed_models:
        print(completed_model)

unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)

```

由于已经定义了两个函数，因此只需要调用它们并传入正确的实参即可。我们调用print_models()并向它传递两个列表。像预期的一样，print_models()模拟了打印设计的过程。接下来，调用show_completed_models()，并将打印好的模型列表传递给它，让它能够指出打印了哪些模型。

这就是我认为**函数**比较重要的第二个作用-----能让阅读这些代码的人也能一目了然的看出这段代码在干嘛，因为有函数名、传参等信息能够推断出来

禁止函数修改列表

```
print_models(unprinted_designs[:], completed_models)
```

这个语句非常的熟悉 之前在列表那章里面有涉及unprinted_designs[:]代表着列表unprinted_designs的副本，因此将其当作实参传入函数，不会对原列表进行改动，只会对副本进行改动。

Tip

虽然向函数传递列表的副本可保留原始列表的内容，但除非**有充分的理由**，否则还是应该将原始列表传递给函数。这是因为，让函数使用现成的列表可**避免花时间和内存创建副本**，从而提高效率，在处理大型列表时尤其如此。

8.5 传递任意数量的实参

有时候，你预先不知道函数需要接受多少个实参，好在Python允许函数从调用语句中收集任意数量的实参。

下面的函数只有一个形参 *toppings，不管调用语句提供了多少实参，这个形参都会将其收入囊中

```

def make_pizza(*toppings):
    """打印顾客点的所有配料"""
    print(toppings)

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')

#输出:
('pepperoni',)
('mushrooms', 'green peppers', 'extra cheese')

```

形参名*toppings中的星号让Python创建一个名为toppings的元组，该元组包含函数收到的所有值。函数体内的函数调用print()生成的输出证明，Python既能处理使用一个值调用函数的情形，也能处理使用三个值调用函数的情形。它以类似的方式处理不同的调用。

```

#遍历配料列表
def make_pizza(*toppings):
    """概述要制作的比萨"""
    print("\nMaking a pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')

#输出:
Making a pizza with the following toppings:
- pepperoni

Making a pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese

```

结合使用位置实参和任意数量的实参

如果要让函数接受不同类型的实参，必须在函数定义中将接纳任意数量实参的形参放在最后。Python先匹配位置实参和关键字实参，再将余下的实参都收集到最后一个形参中。

```

def make_pizza(size, *toppings):
    """概述要制作的比萨"""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')

#输出:
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers

```

- extra cheese

基于上述函数定义，Python将收到的第一个值赋给形参size，将其他所有的值都存储在元组toppings中。在函数调用中，首先指定表示比萨尺寸的实参，再根据需要指定任意数量的配料。

使用任意数量的关键字实参

有时候，你需要接受任意数量的实参，但预先不知道传递给函数的会是什么样的信息。在这种情况下，可将函数编写成能够接受任意数量的键值对——调用语句提供了多少就接受多少。

```
def build_profile(first, last, **user_info):  
  
    """创建一个字典，其中包含我们知道的有关用户的一切"""  
    user_info['first_name'] = first  
    user_info['last_name'] = last  
    return user_info  
  
user_profile = build_profile('albert', 'einstein',  
                             location='princeton',  
                             field='physics')  
print(user_profile)  
  
#输出：  
{'location': 'princeton', 'field': 'physics',  
'first_name': 'albert', 'last_name': 'einstein'}
```

build_profile()函数的定义要求提供名和姓，同时允许根据需要提供任意数量的名值对。形参**user_info中的两个星号让Python创建一个名为user_info的字典，该字典包含函数收到的其他所有名值对。

① Note

总结

*toppings以元组形式接收任意数量的实参 而**user_info以字典形式接收

8.6 将函数存储在模块中

使用函数的优点之一是可将代码块与主程序分离。通过给函数指定描述性名称，能让程序容易理解得多。你还可以更进一步，将函数存储在称为模块的独立文件中，再将模块导入（import）主程序。import语句可让你在当前运行的程序文件中使用模块中的代码。

导入整个模块

要让函数是可导入的，得先创建模块。模块是扩展名为.py的文件，包含要导入程序的代码。下面来创建一个包含make_pizza()函数的模块。

```
#pizza.py  
def make_pizza(size, *toppings):  
    print(f"\nMaking a {size}-inch pizza with the following toppings:")  
    for topping in toppings:  
        print(f"- {topping}")
```

接下来，在pizza.py所在的目录中创建一个名为making_pizzas.py的文件。这个文件先导入刚创建的模块，再调用make_pizza()两次：

```
import pizza

pizza.make_pizza(16, 'pepperoni')
pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')

#输出:
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

这是一种导入方法：只需编写一条import语句并在其中指定模块名，就可在程序中使用该模块中的所有函数。如果使用这种import语句导入了名为module_name.py的整个模块，就可使用下面的语法来使用其中的任意一个函数：

```
module_name.function_name()
```

导入特定的函数

还可以只导入模块中的特定函数，语法如下：

```
from module_name import function_name
```

用逗号分隔函数名，可根据需要从模块中导入任意数量的函数：

```
from module_name import function_0, function_1, function_2
```

对于前面的making_pizzas.py示例，如果只想导入要使用的函数，代码将类似于下面这样：

```
from pizza import make_pizza

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

如果使用这种语法，在调用函数时则无须使用句点。由于在import语句中显式地导入了make_pizza()函数，因此在调用时只需指定其名称即可。

使用as给函数指定别名

如果要导入的函数的名称太长或者可能与程序中既有的名称冲突，可指定简短而独一无二的别名（alias）：函数的另一个名称，类似于外号。

```
#下面给make_pizza()函数指定了别名mp()

from pizza import make_pizza as mp

mp(16, 'pepperoni')
mp(12, 'mushrooms', 'green peppers', 'extra cheese')
```

上面的import语句将函数make_pizza()重命名为mp()。在这个程序中，每当需要调用make_pizza()时，都可将其简写成mp()。Python将运行make_pizza()中的代码，同时避免与程序可能包含的make_pizza()函数混淆。

指定别名的通用语法如下：

```
from module_name import function_name as fn
```

使用as给模块指定别名

还可以给模块指定别名。通过给模块指定简短的别名（如给pizza模块指定别名p），你能够更轻松地调用模块中的函数。

```
import pizza as p

p.make_pizza(16, 'pepperoni')
p.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

ⓘ Note

之后学numpy会用到

import numpy as np一般都是这么写的

给模块指定别名的通用语法如下：

```
import module_name as mn
```

导入模块中的所有函数

使用星号(*)运算符可让Python导入模块中的所有函数：

```
from pizza import *

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

import语句中的星号让Python将模块pizza中的每个函数都复制到这个程序文件中。由于导入了每个函数，可通过名称来调用每个函数，无须使用点号(dot notation)。然而，在使用并非自己编写的大型模块时，最好不要使用这种导入方法，因为如果模块中有函数的名称与当前项目中既有的名称相同，可能导致意想不到的结果：Python可能会因为遇到多个名称相同的函数或变量而覆盖函数，而不是分别导入所有的函数。

❗ Caution

不要用这种办法 这里只是介绍

8.7 函数编写指南

在编写函数时，需要牢记几个细节。应给函数指定描述性名称，且只使用小写字母和下划线。描述性名称可帮助你和别人明白代码想要做什么。

每个函数都应包含简要阐述其功能的注释。该注释应紧跟在函数定义后面，并采用文档字符串的格式。这样，其他程序员只需阅读文档字符串中的描述就能够使用它：他们完全可以相信代码会如描述的那样运行，并且只要知道函数名、需要的实参以及返回值的类型，就能在自己的程序中使用它。

💡 Tip

1.在给形参指定默认值时，等号两边不要有空格：

```
def function_name(parameter_0, parameter_1='default value')
```

2.函数调用中的关键字实参也应遵循这种约定：

```
function_name(value_0, parameter_1='value')
```

3.PEP8建议代码行的长度不要超过79个字符。这样，只要编辑器窗口适中，就能看到整行代码。如果形参很多，导致函数定义的长度超过了79个字符，可在函数定义中输入左括号后按回车键，并在下一行连接两次制表符键，从而将形参列表和只缩进一层的函数体区分开来。

大多数编辑器会自动对齐后续参数列表行，使其缩进程度与你给第一个参数列表行指定的缩进程度相同

4.所有的import语句都应放在文件开头

5.如果程序或模块包含多个函数，可使用两个空行将相邻的函数分开。这样将更容易知道前一个函数到什么地方结束，下一个函数从什么地方开始

第9章 类

面向对象编程 (object-oriented programming, OOP) 是最有效的软件编写方法之一。在面向对象编程中，你编写表示现实世界中的事物和情景的类 (class)，并基于这些类来创建对象 (object)。

根据类来创建对象称为实例化，这让你能够使用类的实例 (instance)。在本章中，你将编写一些类并创建其实例。你将指定可在实例中存储什么信息，定义可对这些实例执行哪些操作。你还将编写一些类来扩展既有类的功能，让相似的类能够共享功能，从而使用更少的代码做更多的事情。你将把自己编写的类存储在模块中，并在自己的程序文件中导入其他程序员编写的类。

ⓘ Note

在深度学习中一般就会定义一个模型类然后创建并调用定义的相关函数

9.1 创建和使用类

创建Dog类

根据Dog类创建的每个实例都将存储名字和年龄，而且我们会赋予每条小狗坐下 (sit()) 和打滚 (roll_over()) 的能力：

```
class Dog:  
    def __init__(self, name, age):  
        """初始化属性name和age"""  
        self.name = name  
        self.age = age  
  
    def sit(self):  
        """模拟小狗收到命令时坐下"""  
        print(f"{self.name} is now sitting.")  
  
    def roll_over(self):  
        """模拟小狗收到命令时打滚"""  
        print(f"{self.name} rolled over!")
```

Tip

首先，定义一个名为Dog的类。根据约定，在Python中，首字母大写的名称指的是类。

#__init__()方法

类中的函数称为方法。你在前面学到的有关函数的一切都适用于方法，就目前而言，唯一重要的差别是调用方法的方式。这个函数是一个特殊的方法，每当你根据Dog类创建新实例时，Python都会自动运行它。在这个方法的名称中，开头和末尾各有两个下划线，这是一种约定，旨在避免Python默认方法与普通方法发生名称冲突。务必确保__init()__的两边都有两个下划线，否则当你使用类来创建实例时，将不会自动调用这个方法，进而引发难以发现的错误。

我们将__init__()方法定义成包含三个形参：self、name和age。在这个方法的定义中，形参self必不可少，而且必须位于其他形参的前面。为何必须在方法定义中包含形参self呢？因为当Python调用这个方法来创建Dog实例时，将自动传入实参self。每个与实例相关联的方法调用都会自动传递实参self，该实参是一个指向实例本身的引用，让实例能够访问类中的属性和方法。当我们创建Dog实例时，Python将调用Dog类的__init__()方法。我们将通过实参向Dog()传递名字和年龄；self则会自动传递，因此不需要我们来传递。每当我们根据Dog类创建实例时，都只需给最后两个形参(name和age)提供值。

在__init__()方法内定义的两个变量都有前缀self。以self为前缀的变量可供类中的所有方法使用，可以通过类的任意实例来访问。self.name=name获取与形参name相关联的值，并将其赋给变量name，然后该变量被关联到当前创建的实例。self.age=age的作用与此类似。像这样可通过实例访问的变量称为属性。

Dog类还定义了另外两个方法：sit()和roll_over()。由于这些方法执行时不需要额外的信息，因此只有一个形参self。稍后将创建的实例能够访问这些方法，换句话说，它们都会坐下和打滚。当前，sit()和roll_over()所做的有限，只是打印一条消息，指出小狗正在坐下或打滚。但是可以扩展这些方法以模拟实际情况：如果这个类属于一个计算机游戏，那么这些方法将包含创建小狗坐下和打滚动画效果的代码；如果这个类是用于控制机器狗的，那么这些方法将让机器狗做出坐下和打滚的动作。

根据类创建实例

下面创建一个表示特定小狗的实例：

```
class Dog:  
    def __init__(self, name, age):  
        """初始化属性name和age"""  
        self.name = name  
        self.age = age  
  
    def sit(self):  
        """模拟小狗收到命令时坐下"""  
        print(f"{self.name} is now sitting.")  
  
    def roll_over(self):  
        """模拟小狗收到命令时打滚"""  
        print(f"{self.name} rolled over!")  
  
my_dog = Dog('willie', 6)  
print(f"My dog's name is {my_dog.name}.")  
print(f"My dog is {my_dog.age} years old.")
```

```
#输出:  
My dog's name is willie.  
My dog is 6 years old.
```

这里使用的是上一个示例中编写的**Dog类**。我们让Python创建一条名字为'Willie'、年龄为6的小狗。在处理这行代码时，Python调用Dog类的`__init__()`方法，并传入实参'Willie'和6。`__init__()`方法创建一个表示特定小狗的实例，并且使用提供的值设置属性name和age。接下来，Python**返回一个表示这条小狗的实例**，而我们将这个实例赋给变量my_dog。在这里，命名约定很有用：通常可以认为首字母大写的名称（如Dog）指的是类，而全小写的名称（如my_dog）指的是根据类创建的实例。

Important

访问属性

要访问实例的属性，可使用点号可以用如下代码来访问my_dog的属性name的值

```
my_dog.name
```

点号在Python中很常用，这种语法演示了Python如何**获取属性的值**。

调用方法

```
class Dog:  
    def __init__(self, name, age):  
        """初始化属性name和age"""  
        self.name = name  
        self.age = age  
  
    def sit(self):  
        """模拟小狗收到命令时坐下"""  
        print(f"{self.name} is now sitting.")  
  
    def roll_over(self):  
        """模拟小狗收到命令时打滚"""  
        print(f"{self.name} rolled over!")  
  
my_dog = Dog('willie', 6)  
my_dog.sit()  
my_dog.roll_over()  
  
#输出:  
willie is now sitting.  
willie rolled over!
```

要调用方法，需指定实例名（这里是my_dog）和想调用的方法，并用句点分隔。在遇到代码`my_dog.sit()`时，Python在类Dog中查找方法sit()并运行其代码。Python用同样的方式解读代码`my_dog.roll_over()`。

这种语法很有用。如果给属性和方法指定了合适的描述性名称，如name、age、sit()和roll_over()，即便对于从未见过的代码块，我们也能够轻松地推断出它是做什么的。

创建多个实例

```
class Dog:  
    def __init__(self, name, age):  
        """初始化属性name和age"""
```

```

        self.name = name
        self.age = age

    def sit(self):
        """模拟小狗收到命令时坐下"""
        print(f"{self.name} is now sitting.")

    def roll_over(self):
        """模拟小狗收到命令时打滚"""
        print(f"{self.name} rolled over!")

my_dog = Dog('willie', 6)
your_dog = Dog('Lucy', 3)

print(f"My dog's name is {my_dog.name}.")
print(f"My dog is {my_dog.age} years old.")
my_dog.sit()

print(f"\nYour dog's name is {your_dog.name}.")
print(f"Your dog is {your_dog.age} years old.")
your_dog.sit()

#输出:
My dog's name is willie.
My dog is 6 years old.
willie is now sitting.

Your dog's name is Lucy.
Your dog is 3 years old.
Lucy is now sitting.

```

9.2 使用类和实例

Car类

下面编写一个表示汽车的类，它存储了有关汽车的信息，并提供了一个汇总这些信息的方法：

```

class Car:
    """一次模拟汽车的简单尝试"""

    def __init__(self, make, model, year):
        """初始化描述汽车的属性"""
        self.make = make
        self.model = model
        self.year = year

    def get_descriptive_name(self):
        """返回格式规范的描述性信息"""
        long_name = f'{self.year} {self.make} {self.model}'
        return long_name.title()

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())

#输出:
2024 Audi A4

```

定义`__init__()`方法。与前面的Dog类中一样，这个方法的第一个形参为`self`。此外，这个方法还包含三个形参：`make`、`model`和`year`。`__init__()`方法接受这些形参的值，并将它们赋给根据这个类创建的实例的属性。在创建新的Car实例时，需要指定其制造商、型号和生产年份。

定义一个名为`get_descriptive_name()`的方法，它使用属性`year`、`make`和`model`创建一个对汽车进行描述的字符串，让我们无须分别打印每个属性的值。为了在这个方法中访问属性的值，使用了`self.make`、`self.model`和`self.year`。

根据Car类创建一个实例，并将其赋给变量`my_new_car`。接下来，调用`get_descriptive_name()`方法，指出我们拥有一辆什么样的汽车

给属性指定默认值

有些属性无须通过形参来定义，可以在`__init__()`方法中为其指定默认值。

下面来添加一个名为`odometer_reading`的属性，其初始值总是为0。我们还添加了一个名为`read_odometer()`的方法，用于读取汽车的里程表：

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()

    def read_odometer(self):
        print(f"This car has {self.odometer_reading} miles on it.")

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

现在，当Python调用`__init__()`方法创建新实例时，将像上一个示例一样以属性的方式存储制造商、型号和生产年份。接下来，Python创建一个名为`odometer_reading`的属性，并将其初始值设置为0。定义一个名为`read_odometer()`的方法，让你能够轻松地知道汽车的行驶里程。

修改属性的值

可以用三种不同的方式修改属性的值：直接通过实例修改，通过方法设置，以及通过方法递增。

Note

直接修改属性的值

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()
```

```

def read_odometer(self):
    print(f"This car has {self.odometer_reading} miles on it.")

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())


my_new_car.odometer_reading = 23 #这里使用点号直接访问并设置汽车的属性
odometer_reading
my_new_car.read_odometer()

#输出:
2024 Audi A4
This car has 23 miles on it.

```

通过方法修改属性的值

有一个替你更新属性的方法大有裨益。这样就无须直接访问属性了，而是可将值传递给方法，由它在内部进行更新。

```

class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f'{self.year} {self.make} {self.model}'
        return long_name.title()

    def read_odometer(self):
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        self.odometer_reading = mileage

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())


my_new_car.update_odometer(23)
my_new_car.read_odometer()

#输出:
2024 Audi A4
This car has 23 miles on it.

```

通过实例my_new_car调用update_odometer()，并向它提供了实参23（该实参对应于方法定义中的形参 mileage）。这将里程表读数设置为23。

还可以对update_odometer()方法进行扩展，使其在修改里程表读数时做些额外的工作。

```

class Car:
    def __init__(self, make, model, year):

```

```

    self.make = make
    self.model = model
    self.year = year
    self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f'{self.year} {self.make} {self.model}'
        return long_name.title()

    def read_odometer(self):
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

```

现在，update_odometer()会在修改属性前检查指定的读数是否合理。如果给mileage指定的值大于或等于原来的行驶里程（self.odometer_reading），就将里程表读数改为新指定的行驶程；否则发出警告，指出不能将里程表往回调。

通过方法让属性的值递增

有时候需要将属性值递增特定的量，而不是将其设置为全新的值。假设我们购买了一辆二手车，从购买到登记期间增加了100英里的里程。下面的方法让我们能够传递这个增量，并相应地增大里程表读数：

```

class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f'{self.year} {self.make} {self.model}'
        return long_name.title()

    def read_odometer(self):
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        self.odometer_reading += miles

my_used_car = Car('subaru', 'outback', 2019)
print(my_used_car.get_descriptive_name())

my_used_car.update_odometer(23_500) #这里的_是用来划分大数据的格式的 读入的时候会自动去掉

```

```
my_used_car.read_odometer()

my_used_car.increment_odometer(100)
my_used_car.read_odometer()

#输出:
2019 subaru outback
This car has 23500 miles on it.
This car has 23600 miles on it.
```

新增的方法increment_odometer()接受一个单位为英里的数，并将其加到self.odometer_reading上。首先，创建一辆二手车my_used_car。然后，调用update_odometer()方法并传入23_500，将这辆二手车的里程表读数设置为23500。最后，调用increment_odometer()并传入100，以增加从购买到登记期间行驶的100英里。

9.3 继承

在编写类时，并非总是要从头开始。如果要编写的类是一个既有的类的特殊版本，可使用继承。当一个类继承另一个类时，将自动获得后者的所有属性和方法。原有的类称为父类，而新类称为子类。子类不仅继承了父类的所有属性和方法，还可定义自己的属性和方法。

子类的__init__()方法

例如，下面来模拟电动汽车。电动汽车是一种特殊的汽车，因此可在之前Car类的基础上创建新类ElectricCar。这样，只需为电动汽车特有的属性和行为编写代码即可。

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()

    def read_odometer(self):
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        self.odometer_reading += miles

class ElectricCar(Car):
    def __init__(self, make, model, year):
        super().__init__(make, model, year)

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
```

```
#输出:  
2024 Nissan Leaf
```

首先是Car类的代码。在创建子类时，父类必须包含在当前文件中，且位于子类前面。接下来，定义子类ElectricCar。在定义子类时，必须在括号内指定父类的名称。`__init__()`方法接受创建Car实例所需的信息。

`super()`是一个特殊的函数，让你能够调用父类的方法。这行代码让Python调用Car类的`__init__()`方法，从而让ElectricCar实例包含这个方法定义的所有属性。父类也称为**超类**，函数名`super`由此得名。

给子类定义属性和方法

下面添加一个电动汽车特有的属性（电池），以及一个描述该属性的方法。我们将存储电池容量，并编写一个方法打印对电池的描述：

```
class Car:  
    def __init__(self, make, model, year):  
        self.make = make  
        self.model = model  
        self.year = year  
        self.odometer_reading = 0  
  
    def get_descriptive_name(self):  
        long_name = f'{self.year} {self.make} {self.model}'  
        return long_name.title()  
  
    def read_odometer(self):  
        print(f"This car has {self.odometer_reading} miles on it.")  
  
    def update_odometer(self, mileage):  
        if mileage >= self.odometer_reading:  
            self.odometer_reading = mileage  
        else:  
            print("You can't roll back an odometer!")  
  
    def increment_odometer(self, miles):  
        self.odometer_reading += miles  
  
class ElectricCar(Car):  
    def __init__(self, make, model, year):  
        super().__init__(make, model, year)  
        self.battery_size = 40  
  
    def describe_battery(self):  
        print(f"This car has a {self.battery_size}-kwh battery.")  
  
my_leaf = ElectricCar('nissan', 'leaf', 2024)  
print(my_leaf.get_descriptive_name())  
my_leaf.describe_battery()  
  
#输出:  
2024 Nissan Leaf  
This car has a 40-kwh battery.
```

添加新属性self.battery_size，并设置其初始值（40）。根据ElectricCar类创建的所有实例都将包含这个属性，但所有的Car实例都不包含它。还添加了一个名为describe_battery()的方法，用来打印有关电池的信息。

重写父类中的方法

在使用类模拟的实物的行为时，如果父类中的一些方法不能满足子类的需求，就可以用下面的办法重写：在子类中定义一个与要重写的父类方法同名的方法。这样，Python将忽略这个父类方法，只关注你在子类中定义的相应方法。

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()

    def read_odometer(self):
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage

        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        self.odometer_reading += miles

    def fill_gas_tank(self):
        print("This car has a gas tank!")

class ElectricCar(Car):
    def __init__(self, make, model, year):
        super().__init__(make, model, year)
        self.battery_size = 40

    def describe_battery(self):
        print(f"This car has a {self.battery_size}-kwh battery.")

    def fill_gas_tank(self):
        print("This car doesn't have a gas tank!")

my_used_car = Car('subaru', 'outback', 2019)
my_used_car.fill_gas_tank()
```

```
my_leaf = Electriccar('nissan', 'leaf', 2024)
my_leaf.fill_gas_tank()
```

#输出：

```
This car has a gas tank!
This car doesn't have a gas tank!
```

现在，如果有人对电动汽车调用fill_gas_tank()方法， Python将忽略Car类中的fill_gas_tank()方法， 转而运行上述代码。

将实例用作属性

在使用代码模拟实物时，你可能会发现自己给类添加了太多细节：属性和方法越来越多，文件越来越长。在这种情况下，可能需要将类的一部分提取出来，作为一个独立的类。将大型类拆分成多个协同工作的小类，这种 方法称为组合。

例如，在不断给ElectricCar类添加细节时，我们可能会发现其中包含很多专门针对汽车电池的属性和方法。在这种情况下，可将这些属性和方法提取出来，放到一个名为Battery的类中，并将一个Battery实例作为ElectricCar类的属性：

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()

    def read_odometer(self):
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage

        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        self.odometer_reading += miles

class Battery:
    def __init__(self, battery_size=40):
        self.battery_size = battery_size

    def describe_battery(self):
        print(f"This car has a {self.battery_size}-kwh battery.")
```

```

class ElectricCar(Car):
    def __init__(self, make, model, year):
        super().__init__(make, model, year)
        self.battery = Battery()

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
my_leaf.battery.describe_battery()

#输出:
2024 Nissan Leaf
This car has a 40-kwh battery.

```

我们定义了一个名为Battery的新类，它没有继承任何类。`__init__()`方法在`self`之外还有一个形参`battery_size`。这个形参是可选的：如果没有给它提供值，电池容量将被设置为40。`describe_battery()`方法也被移到了这个类中。

在`ElectricCar`类中，添加一个名为`self.battery`的属性。这行代码让Python创建一个新的`Battery`实例（因为没有指定容量，所以为默认值40，并将该实例赋给属性`self.battery`。每当`__init__()`方法被调用时，都将执行该操作，因此现在每个`ElectricCar`实例都包含一个自动创建的`Battery`实例。

我们创建一辆电动汽车，并将其赋给变量`my_leaf`。在描述电池时，需要使用电动汽车的属性`battery`：

```
my_leaf.battery.describe_battery()
```

这行代码让Python在实例`my_leaf`中查找属性`battery`，并对存储在该属性中的`Battery`实例调用`describe_battery()`方法。

9.4 导入类

下面创建一个只包含`Car`类的模块。有一个微妙的命名问题：在本章中，已经有一个名为`car.py`的文件，但这个模块也应命名为`car.py`，因为它包含表示汽车的代码。我们将这样解决这个命名问题：将`Car`类存储在一个名为`car.py`的模块中，该模块将覆盖前面的文件`car.py`。从现在开始，使用该模块的程序都必须使用更具体的文件名，如`my_car.py`。下面是模块`car.py`，其中只包含`Car`类的代码：

```

#car.py
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f'{self.year} {self.make} {self.model}'
        return long_name.title()

    def read_odometer(self):
        print(f'This car has {self.odometer_reading} miles on it.')

    def update_odometer(self, mileage):

```

```
if mileage >= self.odometer_reading:
    self.odometer_reading = mileage

else:
    print("You can't roll back an odometer!")

def increment_odometer(self, miles):
    self.odometer_reading += miles
```

下面来创建另一个文件——my_car.py，在其中导入Car类并创建其实例：

```
from car import Car

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

import语句让Python打开模块car并导入其中的Car类。这样，我们就可以使用Car类，就像它是在当前文件中定义的一样。输出与你在前面看到的一样：

```
2024 Audi A4
This car has 23 miles on it.
```

导入类是一种高效的编程方式。如果这个程序包含整个Class类，它该有多长啊！通过将这个类移到一个模块中并导入该模块，依然可使用其所有功能，但主程序文件变得整洁易读了。这还让你能够将大部分逻辑存储在独立的文件中。在确定类能像你希望的那样工作后，就可以不管这些文件，专注于主程序的高级逻辑了。

在一个模块中存储多个类

尽管同一个模块中的类之间应该存在某种相关性，但其实可以根据需要在一个模块中存储任意数量的类。Battery类和ElectricCar类都可帮助模拟汽车，下面将它们都加入模块car.py：

```
#car.py
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f'{self.year} {self.make} {self.model}'
        return long_name.title()

    def read_odometer(self):
        print(f'This car has {self.odometer_reading} miles on it.)
```

```

def update_odometer(self, mileage):
    if mileage >= self.odometer_reading:
        self.odometer_reading = mileage

    else:
        print("You can't roll back an odometer!")

def increment_odometer(self, miles):
    self.odometer_reading += miles

class Battery:
    def __init__(self, battery_size=40):
        self.battery_size = battery_size

    def describe_battery(self):
        print(f"This car has a {self.battery_size}-kwh battery.")

class ElectricCar(Car):
    def __init__(self, make, model, year):
        super().__init__(make, model, year)
        self.battery = Battery()

```

现在，可以新建一个名为my_electric_car.py的文件，导入ElectricCar类，并创建一辆电动汽车了：

```

from car import ElectricCar

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
my_leaf.battery.describe_battery()
my_leaf.battery.get_range()

#输出:
2024 Nissan Leaf
This car has a 40-kwh battery.
This car can go about 150 miles on a full charge.

```

从一个模块中导入多个类

可以根据需要在程序文件中导入任意数量的类。如果要在同一个程序中创建燃油汽车和电动汽车，就需要将Car类和ElectricCar类都导入：

```

from car import Car, ElectricCar

my_mustang = Car('ford', 'mustang', 2024)
print(my_mustang.get_descriptive_name())
my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())

#输出:
2024 Ford Mustang
2024 Nissan Leaf

```

导入整个模块

还可以先导入整个模块，再使用点号访问需要的类。这种导入方法很简单，代码也易读。由于创建类实例的代码都包含模块名，因此不会与当前文件使用的任何名称发生冲突。

下面的代码导入整个car模块，并创建一辆燃油汽车和一辆电动汽车：

```
import car

my_mustang = car.Car('ford', 'mustang', 2024)
print(my_mustang.get_descriptive_name())

my_leaf = car.ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
```

首先，导入整个car模块。接下来，使用语法module_name.classname访问需要的类。像前面一样，我们创建了一辆福特野马燃油汽车和一辆日产聆风电动汽车。

导入模块中的所有类

要导入模块中的每个类，可使用下面的语法：

```
from module_name import *
```

不推荐这种导入方式，原因有二。第一，最好只需要看一下文件开头的import语句，就能清楚地知道程序使用了哪些类。第二，这种导入方式还可能引发名称方面的迷惑。如果不小心导入了一个与程序文件中的其他东西同名的类，将引发难以诊断的错误。这里之所以介绍这种导入方式，是因为虽然不推荐，但你可能在别人编写的代码中见到它。

当需要从一个模块中导入很多类时，还是最好在导入整个模块之后使用module_name.classname语法来访问这些类。这样，虽然文件开头并没有列出用到的所有类，但是你清楚地知道在程序的哪些地方使用了导入的模块。此外，这还避免了导入模块中的每个类可能引发的名称冲突。

在一个模块中导入另一个模块

有时候，需要将类分散到多个模块中，以免模块太大或者在同一个模块中存储不相关的类。在将类存储在多个模块中时，你可能会发现一个模块中的类依赖于另一个模块中的类。在这种情况下，可在前一个模块中导入必要的类。

```
#electric_car.py
from car import Car

class Battery:
    def __init__(self, battery_size=40):
        self.battery_size = battery_size

    def describe_battery(self):
        print(f"This car has a {self.battery_size}-kWh battery.")

class ElectricCar(Car):
    def __init__(self, make, model, year):
        super().__init__(make, model, year)
        self.battery = Battery()
```

ElectricCar类需要访问其父类Car，因此直接将Car类导入该模块。如果忘记了这行代码，Python将在我们试图创建ElectricCar实例时报错。

```
D:\Anaconda\envs\streamlit\python.exe "E:\Python编程 从入门到实践\example\9.类\9.类.py"
Traceback (most recent call last):
  File "E:\Python编程 从入门到实践\example\9.类\9.类.py", line 256, in <module>
    from electric_car import ElectricCar
  File "E:\Python编程 从入门到实践\example\9.类\electric_car.py", line 12, in <module>
    class ElectricCar(Car):
          ^
NameError: name 'Car' is not defined. Did you mean: 'chr'?
```

现在可分别从每个模块中导入类，以根据需要创建任意类型的汽车了：

```
from car import Car
from electric_car import ElectricCar

my_mustang = Car('ford', 'mustang', 2024)
print(my_mustang.get_descriptive_name())

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
```

我们从car模块中导入了Car类，并从electric_car模块中导入了ElectricCar类。接下来，创建一辆燃油汽车和一辆电动汽车。这两种汽车都被正确地创建了：

```
#输出:
2024 Ford Mustang
2024 Nissan Leaf
```

使用别名

假设要在程序中创建大量电动汽车实例，需要反复输入ElectricCar，非常烦琐。为了避免这种烦恼，可在import语句中给ElectricCar指定一个别名：

```
from electric_car import ElectricCar as EC
```

现在每当需要创建电动汽车实例时，都可使用这个别名：

```
my_leaf = EC('nissan', 'leaf', 2024)
```

还可以给模块指定别名。下面导入模块electric_car并给它指定了别名：

```
import electric_car as ec
```

现在可以结合使用模块别名和完整的类名了：

```
my_leaf = ec.ElectricCar('nissan', 'leaf', 2024)
```

9.5 Python标准库

Python标准库是一组模块，在安装Python时已经包含在内。你现在已经对函数和类的工作原理有了大致的了解，可以开始使用其他程序员编写好的模块了。

9.6 类的编程风格

类名应采用驼峰命名法，即将类名中的每个单词的首字母都大写，并且不使用下划线。实例名和模块名都采用全小写格式，并在单词之间加上下划线。

漫长的第9章到此结束！ 

第10章 文件和异常

10.1 读取文件

文本文件可存储的数据多得令人难以置信：天气数据、交通数据、社会经济数据、文学作品，等等。每当需要分析或修改存储在文件中的信息时，读取文件都很有用，对数据分析应用程序来说尤其如此。

读取文件的全部内容

要读取文件，需要一个包含若干行文本的文件。下面来创建一个文件，它包含精确到小数点后30位的圆周率值，且在小数点后每10位处换行：

```
#for_you
3.1415926535
8979323846
2643383279
```

下面的程序打开并读取这个文件，再将其内容显示到屏幕上：

```
from pathlib import Path

path = Path('for_you.txt')
contents = path.read_text()
print(contents)
```

要使用文件的内容，需要将其路径告知Python。**路径**指的是文件或文件夹在系统中的准确位置。Python提供了pathlib模块，让你能够更轻松地在各种操作系统中处理文件和目录。提供特定功能的模块通常称为库。这就是这个模块被命名为pathlib的原因所在。

这里首先从pathlib模块导入Path类。Path对象指向一个文件，可用来做很多事情。例如，让你在使用文件前核实它是否存在，读取文件的内容，以及将新数据写入文件。这里创建了一个表示文件for_you.txt的Path对象，并将其赋给了变量path。由于这个文件与当前编写的.py文件位于同一个目录中，因此Path只需要知道其文件名就能访问它。

创建表示文件for_you.txt的Path对象后，使用read_text()方法来读取这个文件的全部内容。read_text()将该文件的全部内容作为一个字符串返回，而我们将这个字符串赋给了变量contents。

相比于原始文件，该输出唯一不同的地方是末尾多了一个空行。为何会多出这个空行呢？因为read_text()在到达文件末尾时会返回一个空字符串，而这个空字符串会被显示为一个空行。

第2章介绍过，Python方法rstrip()能删除字符串末尾的空白。

要在读取文件内容时删除末尾的换行符，可在调用read_text()后直接调用方法rstrip()：

```
contents = path.read_text().rstrip()
```

这行代码先让Python对当前处理的文件调用read_text()方法，再对read_text()返回的字符串调用rstrip()方法，然后将整理好的字符串赋给变量contents。这种做法称为**方法链式调用**，在编程时很常用。

相对文件路径和绝对文件路径

当将类似于for_you.txt这样的简单文件名传递给Path时，Python将在当前执行的文件所在的目录中查找。

根据你组织文件的方式，有时可能要打开不在程序文件所属目录中的文件。例如，你可能将程序文件存储在了文件夹python_work中，并且在文件夹python_work中创建了一个名为text_files的文件夹，用于存储程序文件要操作的文本文件。虽然文件夹text_files在文件夹python_work中，但仅向Path传递文件夹text_files中的文件的名称也是不可行的，因为Python只在文件夹python_work中查找，而不会在其子文件夹text_files中查找。要让Python打开不与程序文件位于同一个目录中的文件，需要提供正确的路径。

在编程中，指定路径的方式有两种。首先，**相对文件路径**让Python到相对于当前运行的程序所在目录的指定位置去查找。由于文件夹text_files位于文件夹python_work中，因此需要创建一个以text_files打头并以文件名结尾的路径，如下所示：

```
path = Path('text_files/filename.txt')
```

其次，可以将文件在计算机中的准确位置告诉Python，这样就不用管当前运行的程序存储在什么地方了。这称为**绝对文件路径**。在相对路径行不通时，可使用绝对路径。假如text_files并不在文件夹python_work中，则仅向Path传递路径'text_files/filename.txt'是行不通的，因为Python只在文件夹python_work中查找该位置。为了明确地指出希望Python到哪里去查找，需要提供绝对路径。

绝对路径通常比相对路径长，因为它们以系统的根文件夹为起点：

```
path = Path('/home/eric/data_files/text_files/filename.txt')

#上述for_you在我电脑中的绝对路径为E:\Python编程 从入门到实践\example\10. 文件和异常
\for_you.txt
#可在文件的属性-安全中查看到
```

⚠ Warning

注意：在显示文件路径时，Windows系统使用反斜杠（\）而不是斜杠（/）。但是你在代码中应该始终使用斜杠，即便在Windows系统中也是如此。

采用斜杠

The screenshot shows the PyCharm IDE interface. The top window displays the code for `10.文件和异常.py`, which reads the content of the file `for_you.txt`. The bottom window shows the run output, indicating a successful execution with exit code 0.

```
1 #读取文件内容
2 from pathlib import Path
3
4 path = Path('E:/Python编程 从入门到实践/example/10.文件和异常(for_you.txt')
5 contents = path.read_text()
6 print(contents)
```

```
D:\Anaconda\envs\streamlit\python.exe "E:/Python编程 从入门到实践\example\10.文件和异常\10.文件和异常.py"
3.1415926535
8797323846
2643383279
进程已结束，退出代码为 0
```

采用反斜杠 发现会报错

The screenshot shows the PyCharm IDE interface. The top window displays the same code as before, but the bottom window shows a detailed traceback of the error. The error occurs at line 5, where the code tries to read the file, because the path is enclosed in double quotes.

```
1 #读取文件内容
2 from pathlib import Path
3
4 path = Path('E:/Python编程 从入门到实践/example/10.文件和异常(for_you.txt')
5 contents = path.read_text()
6 print(contents)
```

```
D:\Anaconda\envs\streamlit\python.exe "E:/Python编程 从入门到实践\example\10.文件和异常\10.文件和异常.py"
Traceback (most recent call last):
  File "E:/Python编程 从入门到实践\example\10.文件和异常\10.文件和异常.py", line 5, in <module>
    contents = path.read_text()
               ^
File "D:\Anaconda\envs\streamlit\lib\pathlib.py", line 1058, in read_text
    with self.open(mode='r', encoding=encoding, errors=errors) as f:
               ^
File "D:\Anaconda\envs\streamlit\lib\pathlib.py", line 1944, in open
    return io.open(self._fl_name, mode, buffering, encoding, errors, newline)
               ^
```

访问文件中的各行

你可以使用`splitlines()`方法将冗长的字符串转换为一系列行，再使用`for`循环以每次一行的方式检查文件中的各行：

```
from pathlib import Path

path = Path('for_you.txt')
contents = path.read_text()

lines = contents.splitlines()
for line in lines:
    print(line)
```

与前面一样，首先读取文件的全部内容。如果要处理文件中的各行，就无须在读取文件时删除任何空白。`splitlines()`方法返回一个列表，其中包含文件中所有的行，而我们将这个列表赋给了变量`lines`。

使用文件的内容

将文件的内容读取到内存中后，就能以任意方式使用这些数据了。下面以简单的方式使用圆周率的值。

```
from pathlib import Path

path = Path('for_you.txt')
contents = path.read_text()

lines = contents.splitlines()
pi_string = ''
for line in lines:
    pi_string += line

print(pi_string)
print(len(pi_string))

#输出:
3.1415926535 8979323846 2643383279
36
```

像上一个示例一样，首先读取文件，并将其中的所有行都存储在一个列表中。然后，创建变量`pi_string`，用于存储圆周率的值。接下来，使用循环将各行加入`pi_string`。

变量`pi_string`存储的字符串包含原来位于每行左端的空格。要删除这些空格，可对每行调用`lstrip()`：

```
from pathlib import Path

path = Path('for_you.txt')
contents = path.read_text()

lines = contents.splitlines()
pi_string = ''
for line in lines:
    pi_string += line.lstrip()

print(pi_string)
print(len(pi_string))

#输出:
3.141592653589793238462643383279
32
```

Note

回想一下`lstrip()`、`rstrip()`、`strip()`的作用

注意：在读取文本文件时，Python将其中的所有文本都解释为字符串。如果读取的是数，并且要将其作为数值使用，就必须使用`int()`函数将其转换为整数，或者使用`float()`函数将其转换为浮点数。

包含100万位的大型文件

尽管前面分析的都是一个只有三行的文本文件，但是这些代码示例也可以处理比它大得多的文件。如果一个文本文件包含精确到小数点后1000000位而不是30位的圆周率值，也可以创建一个包含所有这些数字的字符串。无须对前面的程序做任何修改，只需将这个文件传递给它即可。

```
from pathlib import Path

path = Path('pi_million_digits.txt')
contents = path.read_text()

lines = contents.splitlines()
pi_string = ''
for line in lines:
    pi_string += line.lstrip()

print(f"{pi_string[:52]}...")
print(len(pi_string))

#输出:
3.14159265358979323846264338327950288419716939937510...
1000002
```

圆周率值中包含你的生日吗

```
from pathlib import Path

path = Path('pi_million_digits.txt')
contents = path.read_text()

lines = contents.splitlines()
pi_string = ''
for line in lines:
    pi_string += line.lstrip()

birthday = input("Enter your birthday, in the form mmddyy: ")
if birthday in pi_string:
    print("Your birthday appears in the first million digits of pi!")
else:
    print("Your birthday does not appear in the first million digits of pi.")
```

首先提示用户输入其生日，再检查这个字符串是否在pi_string中。运行这个程序：

```
D:\Anaconda\envs\streamlit\python.exe "E:\Python编程 从入门到实践\example\10.文件和异常\10.文件和异常.py"
Enter your birthday, in the form mmddyy: 0202
Your birthday appears in the first million digits of pi!
```

10.2 写入文件

保存数据的最简单的方式之一是将其写入文件。通过将输出写入文件，即便关闭包含程序输出的终端窗口，这些输出也依然存在：既可以在程序结束运行后查看这些输出，也可以与他人共享输出文件，还可以编写程序来将这些输出读取到内存中并进行处理。

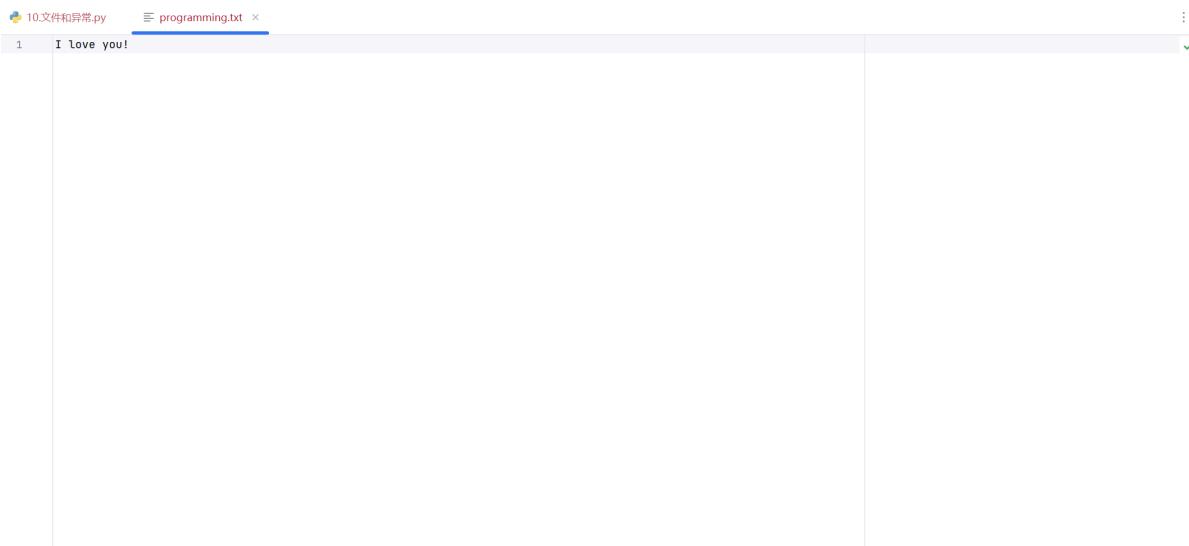
写入一行

定义一个文件的路径后，就可使用write_text()将数据写入该文件了。

```
from pathlib import Path

path = Path('programming.txt')
path.write_text("I love you!")
```

`write_text()`方法接受单个实参，即要写入文件的字符串。这个程序没有终端输出，但你如果打开文件 `programming.txt`，将看到如下一行内容：



Note

注意：Python只能将字符串写入文本文件。如果要将数值数据存储到文本文件中，必须先使用函数 `str()`将其转换为字符串格式。

写入多行

`write_text()`方法会在幕后完成几项工作。首先，如果`path`变量对应的路径指向的文件不存在，就创建它。其次，将字符串写入文件后，它会确保文件得以妥善地关闭。如果没有妥善地关闭文件，可能会导致数据丢失或受损。

要将多行写入文件，需要先创建一个字符串（其中包含要写入文件的全部内容），再调用`write_text()`并将这个字符串传递给它。

```
from pathlib import Path

contents = "I love you!\n"
contents += "You are like diamonds in the sky!\n"
contents += "Your beauty leaves me speechless.\n"

path = Path('programming.txt')
path.write_text(contents)
```

首先定义变量contents，用于存储要写入文件的所有内容。接下来，使用运算符+=在该变量中追加这个字符串。可根据需要执行这种操作任意多次，以创建任意长度的字符串。这里在每行末尾都添加了换行符，让每个句子都占一行。

```
I love you!
You are like diamonds in the sky!
Your beauty leaves me speechless.
```

10.3 异常(感兴趣可以看 个人觉得非科班可以不看)

Python使用称为异常的特殊对象来管理程序执行期间发生的错误。每当发生让Python不知所措的错误时，它都会创建一个异常对象。如果你编写了处理该异常的代码，程序将继续运行；如果你未对异常进行处理，程序将停止，并显示一个traceback，其中包含有关异常的报告。

异常是使用try-except代码块处理的。try-except代码块让Python执行指定的操作，同时告诉Python在发生异常时应该怎么办。在使用try-except代码块时，即便出现异常，程序也将继续运行：显示你编写的友好的错误消息，而不是令用户迷惑的traceback。

处理ZeroDivisionError异常

下面来看一种导致Python引发异常的简单错误。你可能知道不能将数除以0，但还是让Python试试看吧：

```
print(5/0)
```

Python无法这样做，因此你将看到一个traceback：

```
D:\Anaconda\envs\streamlit\python.exe "E:\Python编程 从入门到实践\example\10.文件和异常\10.文件和异常.py"
Traceback (most recent call last):
  File "E:\Python编程 从入门到实践\example\10.文件和异常\10.文件和异常.py", line 94, in <module>
    print(5/0)
           ^
ZeroDivisionError: division by zero
```

在上述traceback中，错误ZeroDivisionError是个异常对象。Python在无法按你的要求做时，就会创建这种对象。在这种情况下，Python将停止运行程序，并指出引发了哪种异常，而我们可根据这些信息对程序进行修改。下面将告诉Python，在发生这种错误时该怎么办。这样，如果再次发生这样的错误，我们就有所准备了。

使用try-except代码块

当你认为可能发生错误时，可编写一个try-except代码块来处理可能引发的异常。你让Python尝试运行特定的代码，并告诉它如果这些代码引发了指定的异常，该怎么办。

处理ZeroDivisionError异常的try-except代码块类似于下面这样：

```
try:  
    print(5/0)  
except ZeroDivisionError:  
    print("You can't divide by zero!")
```

这里将导致错误的代码行print(5/0)放在一个try代码块中。如果try代码块中的代码运行起来没有问题，Python 将跳过except代码块；如果try代码块中的代码导致错误，Python将查找与之匹配的except代码块并运行其中的代码。

在这个示例中，try代码块中的代码引发了ZeroDivisionError异常，因此Python查找指出了该怎么办的except 代码块，并运行其中的代码。这样，用户看到的是一条友好的错误消息，而不是traceback：

```
You can't divide by zero!
```

如果try-except代码块后面还有其他代码，程序将继续运行，因为Python已经知道了如何处理错误。下面来看一个在捕获错误后让程序继续运行的示例。

使用异常避免崩溃

如果在错误发生时，程序还有工作没有完成，妥善地处理错误就显得尤其重要。这种情况经常出现在要求用户提供输入的程序中。如果程序能够妥善地处理无效输入，就能提示用户提供有效输入，而不至于崩溃。

下面来创建一个只执行除法运算的简单运算器：

```
print("Give me two numbers, and I'll divide them.")  
print("Enter 'q' to quit.")  
while True:  
    first_number = input("\nFirst number: ")  
    if first_number == 'q':  
        break  
    second_number = input("Second number: ")  
    if second_number == 'q':  
        break  
  
    answer = int(first_number) / int(second_number)  
    print(answer)
```

程序提示用户输入一个数，并将其赋给变量first_number。如果用户输入的不是表示退出的q，就再提示用户输入一个数，并将其赋给变量second_number。接下来，计算这两个数的商。这个程序没有采取任何处理错误的措施，因此在执行除数为0的除法运算时，它将崩溃：

```
D:\Anaconda\envs\streamlit\python.exe "E:\Python编程 从入门到实践\example\10.文件和异常\10.文件和异常.py"  
Give me two numbers, and I'll divide them.  
Enter 'q' to quit.  
  
First number: 5  
Second number: 0  
Traceback (most recent call last):  
  File "E:\Python编程 从入门到实践\example\10.文件和异常\10.文件和异常.py", line 114, in <module>  
    answer = int(first_number) / int(second_number)  
    ~~~~~^~~~~~  
ZeroDivisionError: division by zero
```

程序崩溃可不好，让用户看到traceback也不是个好主意。不懂技术的用户会感到糊涂，怀有恶意的用户还能通过traceback获悉你不想让他们知道的信息。

else代码块

通过将可能引发错误的代码放在try-except代码块中，可提高程序抵御错误的能力。因为错误是执行除法运算的代码行导致的，所以需要将它放到try-except代码块中。这个示例还包含一个else代码块，只有try代码块成功执行才需要继续执行的代码，都应放到else代码块中：

```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")

while True:
    first_number = input("\nEnter first number: ")
    if first_number == 'q':
        break
    second_number = input("Second number: ")

    if second_number == 'q':
        break

    try:
        answer = int(first_number) / int(second_number)
    except ZeroDivisionError:
        print("You can't divide by 0!")
    else:
        print(answer)
```

我们让Python尝试执行try代码块中的除法运算，这个代码块只包含可能导致错误的代码。依赖try代码块成功执行的代码都被放在else代码块中。在这个示例中，如果除法运算成功，就使用else代码块来打印结果。

except代码块告诉Python，在出现ZeroDivisionError异常时该怎么办。如果try代码块因零除错误而失败，就打印一条友好的消息，告诉用户如何避免这种错误。程序会继续运行，而用户根本看不到 traceback：

```
D:\Anaconda\envs\streamlit\python.exe "E:\Python编程 从入门到实践\example\10.文件和异常\10.文件和异常.py"
Give me two numbers, and I'll divide them.
Enter 'q' to quit.

First number: 5
Second number: 0
You can't divide by 0!

First number: 6
Second number: 2
3.0

First number: q
```

进程已结束，退出代码为 0

只可能引发异常的代码才需要放在try语句中。有时候，有一些仅在try代码块成功执行时才需要运行的代码，这些代码应放在else代码块中。except代码块告诉Python，如果在尝试运行try代码块中的代码时引发了指定的异常该怎么办。

通过预测可能发生错误的代码，可编写稳健的程序。它们即便面临无效数据或缺少资源，也能继续运行，不受无意的用户错误和恶意攻击的影响。

处理FileNotFoundException异常

在使用文件时，一种常见的问题是找不到文件：要查找的文件可能在其他地方，文件名可能不正确，或者这个文件根本就不存在。对于所有这些情况，都可使用try-except代码块来处理。

我们来尝试读取一个不存在的文件。下面的程序尝试读取文件alice.txt的内容，但这个文件并没有被存储在alice.py所在的目录中：

```
from pathlib import Path

path = Path('alice.txt')
contents = path.read_text(encoding='utf-8')
```

请注意，这里使用read_text()的方式与前面稍有不同。如果系统的默认编码与要读取的文件的编码不一致，参数encoding必不可少。如果要读取的文件不是在你的系统中创建的，这种情况更容易发生。

Python无法读取不存在的文件，因此引发了一个异常：

```
D:\Anaconda\envs\streamlit\python.exe "E:\Python编程 从入门到实践\example\10.文件和异常\10.文件和异常.py"
Traceback (most recent call last):
  File "E:\Python编程 从入门到实践\example\10.文件和异常\10.文件和异常.py", line 140, in <module>
    contents = path.read_text(encoding='utf-8')
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "D:\Anaconda\envs\streamlit\Lib\pathlib.py", line 1058, in read_text
    with self.open(mode='r', encoding=encoding, errors=errors) as f:
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "D:\Anaconda\envs\streamlit\Lib\pathlib.py", line 1044, in open
    return io.open(self, mode, buffering, encoding, errors, newline)
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
FileNotFoundException: [Errno 2] No such file or directory: 'alice.txt'
```

这里的traceback比前面的那些都长，因此下面介绍如何看懂复杂的traceback。通常最好从traceback的末尾着手。从最后一行可知，引发了异常FileNotFoundException。这一点很重要，它让我们知道应该在要编写的except代码块中使用哪种异常。

回头看看traceback开头附近，从这里可知，错误发生在文件alice.py的第四行。接下来的一行列出了导致错误的代码行。traceback的其余部分列出了一些代码，它们来自打开和读取文件涉及的库。通常，不需要详细阅读和理解traceback中的这些内容。

为了处理这个异常，应将traceback指出的存在问题的代码行放到try代码块中。这里，存在问题的是包含read_text()的代码行：

```
from pathlib import Path
path = Path('alice.txt')
try:
    contents = path.read_text(encoding='utf-8')
except FileNotFoundError:
    print(f"Sorry, the file {path} does not exist.")
```

在这个示例中，try代码块中的代码引发了FileNotFoundException异常，因此要编写一个与该异常匹配的except代码块。这样，当找不到文件时，Python将运行except代码块中的代码，从而显示一条友好的错误消息，而不是traceback：

```
Sorry, the file alice.txt does not exist.
```

如果文件不存在，这个程序就什么也做不了，因此上面就是这个程序的全部输出。下面来扩展这个示例，看看当你使用多个文件时，异常处理可提供什么样的帮助。

静默失败

要让程序静默失败，可像通常那样编写try代码块，但在except代码块中明确地告诉Python什么都不要做。Python有一个pass语句，可在代码块中使用它来让Python什么都不做：

```
def count_words(path):
    try:
        --snip--
    except FileNotFoundError:
        pass
    else:
        --snip--
```

相比于上一个程序，这个程序唯一的不同之处是，except代码块包含一条pass语句。现在，当出现FileNotFoundError异常时，虽然仍将执行except代码块中的代码，但什么都不会发生。当这种错误发生时，既不会出现traceback，也没有任何输出。

决定报告哪些错误

该在什么情况下向用户报告错误？又该在什么情况下静默失败呢？如果用户知道要分析哪些文件，他们可能希望在有文件未被分析时出现一条消息来告知原因。如果用户只想看到结果，并不知道要分析哪些文件，可能就无须在有些文件不存在时告知他们。向用户显示他们不想看到的信息可能会降低程序的可用性。Python的错误处理结构让你能够细致地控制与用户共享错误信息的程度，要共享多少信息由你决定。

编写得很好且经过恰当测试的代码不容易出现内部错误，如语法错误和逻辑错误，但只要程序依赖于外部因素，如用户输入、是否存在指定的文件、是否有网络连接，就有可能出现异常。凭借经验可判断该在程序的什么地方包含异常处理块，以及出现错误时该向用户提供多少相关的信息。

10.4 存储数据

很多程序要求用户输入某种信息，比如让用户存储游戏首选项或提供要可视化的数据。不管专注点是什么，程序都会把用户提供的信息存储在列表和字典等数据结构中。当用户关闭程序时，几乎总是要保存他们提供的信息。一种简单的方式是使模块json来存储数据。

模块json让你能够将简单的Python数据结构转换为JSON格式的字符串，并在程序再次运行时从文件中加载数据。你还可以使用json在Python程序之间共享数据。更重要的是，JSON数据格式并不是Python专用的，这让你能够将以JSON格式存储的数据与使用其他编程语言的人共享。这是一种轻量级数据格式，不仅很有用，也易于学习。

Note

注意：JSON格式最初是为JavaScript开发的，但随后成了一种通用的格式，被包括Python在内的众多语言采用。

使用`json.dumps()`和`json.loads()`

下面先编写一个存储一组数的简短程序，再编写一个将这些数读取到内存中的程序。第一个程序将使用`json.dumps()`来存储这组数，而第二个程序将使用`json.loads()`来读取它们。

`json.dumps()`函数接受一个实参，即要转换为JSON格式的数据。这个函数返回一个字符串，这样你就可将其写入数据文件了：

```
#number_writer.py
from pathlib import Path
import json

numbers = [2, 3, 5, 7, 11, 13]
path = Path('numbers.json')
contents = json.dumps(numbers)
path.write_text(contents)
```

首先导入模块json，并创建一个数值列表。然后选择一个文件名，指定要将该数值列表存储到哪个文件中。通常使用文件扩展名.json来指出文件存储的数据为JSON格式。接下来，使用json.dumps()函数生成一个字符串，它包含我们要存储的数据的JSON表示形式。生成这个字符串后，像本章前面一样，使用write_text()方法将其写入文件。

这个程序没有输出，我们打开文件numbers.json一探究竟。该文件中数据的存储格式看起来与Python中一样：

```
[2, 3, 5, 7, 11, 13]
```

下面再编写一个程序，使用json.loads()将这个列表读取到内存中：

```
from pathlib import Path
import json
path = Path('numbers.json')
contents = path.read_text()
numbers = json.loads(contents)
print(numbers)
```

确保读取的是前面写入的文件。这个数据文件是使用特殊格式的文本文件，因此可使用read_text()方法来读取它。然后将这个文件的内容传递给json.loads()。这个函数将一个JSON格式的字符串作为参数，并返回一个Python对象（这里是一个列表），而我们将这个对象赋给了变量numbers。最后，打印恢复的数值列表，看看是否与number_writer.py中创建的数值列表相同：

```
[2, 3, 5, 7, 11, 13]
```

这是一种在程序之间共享数据的简单方式。

保存和读取用户生成的数据

使用json保存用户生成的数据很有必要，因为如果不以某种方式进行存储，用户的信息就会在程序停止运行时丢失。下面来看一个这样的例子：提示用户在首次运行程序时输入自己的名字，并且在他再次运行程序时仍然记得他。

先来存储用户名：

```
from pathlib import Path
import json
username = input("what is your name? ")
path = Path('username.json')
contents = json.dumps(username)
path.write_text(contents)
print(f"We'll remember you when you come back, {username}!")
```

提示用户输入名字。接下来，将收集到的数据写入文件username.json。然后，打印一条消息，指出存储了用户输入的信息：

```
what is your name? linyingtian
we'll remember you when you come back, linyingtian!
```

现在再编写一个程序，向名字已被存储的用户发出问候：

```
from pathlib import Path
import json

path = Path('username.json')
contents = path.read_text()
username = json.loads(contents)
print(f"welcome back, {username}!")
```

我们读取数据文件的内容，并使用json.loads()将恢复的数据赋给变量username。有了已恢复的用户名，就可以使用个性化的问候语欢迎用户回来了：

```
welcome back, linyingtian!
```

需要将这两个程序合并到一个程序中。在这个程序运行时，将尝试从内存中获取用户的用户名。如果没有找到，就提示用户输入用户名，并将其存储到文件username.json中，以供下次使用。这里原本可以编写一个try-except代码块，以便在文件username.json不存在时采取合适的措施，但我们没有这样做，而是使用了pathlib模块提供的一个便利方法：

```
from pathlib import Path
import json

path = Path('username.json')
if path.exists():
    contents = path.read_text()
    username = json.loads(contents)
    print(f"welcome back, {username}!")
else:
    username = input("what is your name? ")
    contents = json.dumps(username)
    path.write_text(contents)
    print(f"we'll remember you when you come back, {username}!")
```

Path类提供了很多很有用的方法。如果指定的文件或文件夹存在，exists()方法返回True，否则返回False。这里使用path.exists()来确定是否存储了用户名。如果文件username.json存在，就加载其中的用户名，并向用户发出个性化问候。

如果文件username.json不存在，就提示用户输入用户名，并存储用户输入的值。此外，还会打印一条消息，指出当用户再回来时我们还会记得他。

无论执行的是哪个代码块，都将显示用户名和合适的问候语。如果这是程序首次运行，输出将如下所示：

```
what is your name? linyingtian
we'll remember you when you come back, Eric!
```

否则，输出将如下所示：

```
welcome back, linyingtian!
```

这是程序之前至少运行了一次时的输出。虽然这里存储的数据只是单个字符串，但这个程序可处理所有可转换为JSON格式字符串的数据。

重构

你经常会遇到这样的情况：虽然代码能够正确地运行，但还可以将其划分为一系列完成具体工作的函数来进行改进。这样的过程称为代码更清晰、更易于理解、更容易扩展。

要重构remember_me.py，可将其大部分逻辑放到一个或多个函数中。remember_me.py的重点是问候用户，因此将其所有代码都放到一个名为greet_user()的函数中：

```
from pathlib import Path
import json


def greet_user():
    path = Path('username.json')
    if path.exists():
        contents = path.read_text()
        username = json.loads(contents)
        print(f"welcome back, {username}!")
    else:
        username = input("what is your name? ")
        contents = json.dumps(username)
        path.write_text(contents)
        print(f"We'll remember you when you come back, {username}!")


greet_user()
```

考虑到现在使用了一个函数，我们删除注释，转而使用一个文档字符串来指出程序的作用。这个程序更加清晰，但greet_user()函数所做的不仅是问候用户，还在存储了用户名时获取它，在没有存储用户名时提示用户输入。

下面重构greet_user()，不让它执行这么多任务。首先将获取已存储用户名的代码移到另一个函数中：

```
from pathlib import Path
import json


def get_stored_username(path):
    if path.exists():
        contents = path.read_text()
        username = json.loads(contents)
        return username
    else:
        return None


def greet_user():
    path = Path('username.json')
```

```
username = get_stored_username(path)

if username:
    print(f"welcome back, {username}!")
else:
    username = input("what is your name? ")
    contents = json.dumps(username)
    path.write_text(contents)
    print(f"We'll remember you when you come back, {username}!")

greet_user()
```

新增的get_stored_username()函数目标明确，文档字符串指出了这一点。如果存储了用户名，就获取并返回它；如果传递给get_stored_username()的路径不存在，就返回None。这是一种不错的做法：函数要么返回预期的值，要么返回None。这让我们能够使用函数的返回值做简单的测试。如果成功地获取了用户名，就打印一条欢迎用户回来的消息，否则提示用户输入用户名。

还需要将greet_user()中的另一个代码块提取出来，将在没有存储用户名时提示用户输入的代码放在一个独立的函数中：

```
from pathlib import Path
import json


def get_stored_username(path):
    if path.exists():
        contents = path.read_text()
        username = json.loads(contents)
        return username
    else:
        return None


def get_new_username(path):
    username = input("what is your name? ")
    contents = json.dumps(username)
    path.write_text(contents)
    return username


def greet_user():
    path = Path('username.json')
    username = get_stored_username(path)
    if username:
        print(f"welcome back, {username}!")
    else:
        username = get_new_username(path)
        print(f"We'll remember you when you come back, {username}!")

greet_user()
```

在remember_me.py的这个最终版本中，每个函数都执行单一而清晰的任务。我们调用greet_user()，它打印一条合适的消息：要么欢迎老用户回来，要么问候新用户。为此，它首先调用get_stored_username()，这个函数只负责获取已存储的用户名，再在必要时调用get_new_username()，这个函数只负责获取并存储新用户的用户名。要编写出清晰且易于维护和扩展的代码，这种划分必不可少。