第 1 章 基本语法与数据类型

Python 语言是一种简单而又功能强大的高级语言。Python 的官方介绍是：Python 是一种简单易学、功能强大的编程语言，它有高效率的高层数据结构，能简单而有效地实现面向对象编程。Python 简洁的语法和对动态输入的支持，加上解释性语言的本质，使得它在大多数平台上的许多领域都是一个理想的脚本语言，特别适用于快速应用程序开发。

Python 的发展经历了几个大的版本。当前还有少数的程序员在使用 Python 2.7，但是 Python 2.7 除了商用之外已经停止了更新。当前 Python 已经发展到 3.12 版本，并增加了许多新特性。

1.1 编辑器 Anaconda

Python 的编辑软件比较多，如 Anaconda、Pycharm 等。Windows 用户可以访问 https://Python.org/download 下载其原生的编辑器，本书使用的版本是 Python 3.10，大小约为 25MB，其安装过程与其他可执行软件类似。

打开 Python 的 IDLE，启动 Python 编辑器。

在提示符 >>> 下输入 print('Hello World')，然后按 Enter 键，就可以看到输出的句子 Hello World，如图 1-1 所示。

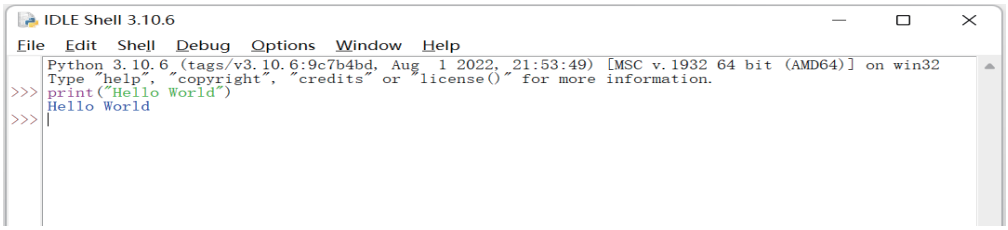


图 1-1 IDLE 界面图

注意：此处的 >>> 为编辑器自动显示的提示符，在 Python 代码中所涉及的括号 ()、引号 " 和冒号 : 等，都需要在英文半角状态下输入。

由于原生的编辑器在使用第三方库时，需要设置较多的运行环境，耗时费力，尤其对计算机系统设置不熟悉的新手来说，可能会无所适从。于是一些“自动化”的编辑平台应运而生，Anaconda 就是这样一款编辑软件。该软件预装了一些常用的库，真正体现了“注重解决实际问题，而非语言本身”，其大小约 700MB。本书将主要基于 Anaconda 下的 Spyder 和 Jupyter notebook(以下简称 Jupy) 运行代码，毕竟 Spyder 和 Jupy 已成为数据分析的标准环境，尤其 Jupy 更是数据分析使用较多的工具。

Anaconda 官方下载网址为：https://www.anaconda.com/products/individual。下载时

请按照计算机的配置情况，下载适配的版本。Anaconda 发展更新较快，若下载往期版本可直接到 https://repo.anaconda.com/archive 页面选择下载。本书使用的是 Anaconda 官方 Windows 系统 64 位的 Python 3.9 版本，下载界面如图 1-2 所示。

图 1-2　Anaconda 官网下载界面

下载后直接双击安装，可自选安装位置。安装完成后，在“开始”菜单里可以看到如图 1-3 所示的目录。注意在自选安装位置时，尽可能地避免中文路径，防止出现不可预见的错误。

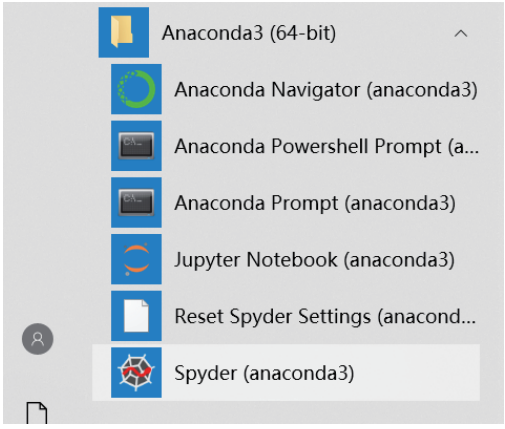


图 1-3　Anaconda 菜单

Anaconda 目录菜单中增加了 Prompt 项，双击打开输入 conda install 命令来安装第三方库，也可以使用 pip install 安装第三方包。例如，安装 jieba 包的命令为：conda install jieba 或 pip install jieba。

第一次打开 Spyder 比较慢。Spyder 的使用比较简单，其界面如图 1-4 所示。

在代码编辑区编辑代码，运行选定的代码，单击键盘上的 F9 键即可。在之前的一些版本中，可以使用 Ctrl+Enter 组合键运行代码。由于版本不同，界面上的按钮以及快捷

键可能略有不同。

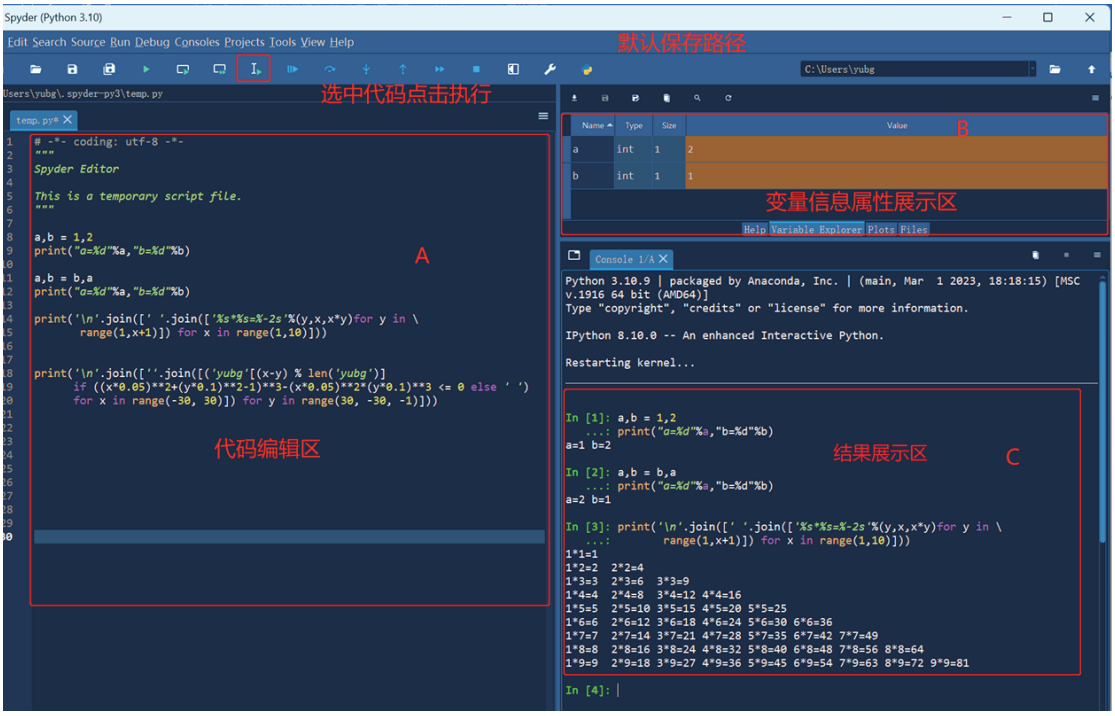


图 1-4　Spyder 界面

1.2　基本语法

下面，我们先了解 Python 的语法常识。

1. 代码注释方法

(1) 通常在一行中，“#”后的语句不再被执行，而表示注释，即注释不是写给机器执行的，而是写给程序员看的，如例 1-1 所示。

【例 1-1】三引号注释段落。

```
# -*- coding: utf-8 -*-
"""
Created on Sun Mar 13 21:20:06 2016
@author: yubg
"""
lis = [1,2,3]
for i in lis:    #半角状态冒号不能少，下一行注意缩进
     print(i)
```

本例无须上机操作，仅为展示用法。

(2) 如果有多行需要注释，可以使用三引号包括三个单引号 (' ') 或三个双引号 (" "),将注释内容包围。单引号和双引号在使用上没有本质的差别。

2. 用缩进表示分层

Python 不像其他语言用括号来表示语句块，而是使用代码缩进 4 个空格来表示分层，当然也可以使用 Tab 键，但不要混合使用 Tab 键和空格来进行缩进，这会使程序在跨平台时不能正常运行，官方推荐的做法是使用四个空格。

一般来说，行尾遇到“:”就表示下一行缩进的开始，如例 1-1 中“for i in lis”行尾有冒号，下一行的“print(i)”就需要缩进四个空格。

3. 语句断行

一般来说，Python 一条语句占一行，在每条语句的结尾处不需要使用“;”。但在 Python 中也可以使用“;”，表示将两条简单的语句写在一行。分号还有一个作用，使用在一行语句的末尾，表示对本行语句的结果不打印输出。但如果一条语句较长要分几行来写，可以用“\”来进行续行，即续行符。

```
a = "昨夜雨疏风骤，浓睡不消残酒。试问卷帘人，却道海棠依旧。知否？知否？应是绿肥红瘦。"
b = "昨夜雨疏风骤，浓睡不消残酒。试问卷帘人，\
却道海棠依旧。知否？知否？应是绿肥红瘦。"
```

上面的 a 和 b 用 print() 输出时，效果是一样的。

再看下面在括号中断行，也即续行。代码如下：

```
df =DataFrame({'age':Series([26,85]),'name':Series(['Ben','Joh'])})
```

一般来说，系统能够自动识别断行，在一对括号中间或三引号之间均可断行。例如，上面代码行较长，若要对其断行，则必须在括号内进行 (包括圆括号、方括号和花括号)，断行后的第二行一般空四个空格，但为了层次清晰，一般采用“逻辑”对齐，在 Spyder 下会自动对齐。这在括号内断行的情况不需要加续行符。

```
df = DataFrame({'age':Series([26,85]),          #此语句断行后分成了两行
                'name':Series(['Ben','Joh'])})
```

4.print() 函数的作用

print() 函数会在输出窗口中显示一些文本或结果，便于验证和显示数据。

print() 是一个常用函数，其功能就是输出括号中的字符串。print() 可以有多个输出，以逗号分隔。代码如下：

```
In [1]: a=10

In [2]: print(a, type(a)) # 输出变量a的值和a的数据类型
   10 <class 'int'>
```

当在循环输出要将多个结果打印在一行并以逗号分隔时，可以在 print() 中添加 end=',' ，如例 1-1 中 print 行可以改为 print(i, end=',')。

5. input() 函数的作用

input() 函数主要用于接收来自键盘的输入，如输入取款数额、手机号码、密码、E-mail 等，将用户输入的内容作为字符串形式返回。需要特别注意，若输入的是数字，

则返回的“数字”类型将会是加了引号的字符型。

【例 1-2】input() 函数输入。

```
In [3]: a = input("请输入: ") #接收来自键盘的输入

请输入: 12

In [4]: a
Out[4]: '12'

In [5]: type(a) #检测a变量的数据类型
Out[5]: str

In [6]: b=input('请您输入:')

请您输入:abc

In [7]: b
Out[7]: 'abc'

In [8]: type(b)
Out[8]: str

In [9]: c = int(input("请您输入数字: ")) #将接收到的内容转化数据类型

请您输入数字: 231

In [10]: type(c)
Out[10]: int

In [11]: d=int(input('请输入字符: '))

请输入字符: acd
Traceback (most recent call last):

  File "C:\Users\yubg\AppData\Local\Temp/ipykernel_7512/2054611132.py",
line 1, in <module>
    d=int(input('请输入字符: '))

ValueError: invalid literal for int() with base 10: 'acd'
```

如果要想获取数字，可以使用 int 函数将接收进来的“数字”字符转化为数字。如：

```
x = int(input("输入数字: "))
```

需要注意，这里若输入字母或者其他符号则会报错，如例 1-2 最后输入的 acd。

type() 函数用来检测变量的数据类型。

6. 特殊符号的输出

有时在字符串中需要输入如续行符“\”或者引号“'”等特殊符号 (无须在代码中进行转义的符号)，此时，可以在特殊符号前加反斜杠 (\)，即 \\ 或 \'。

```
In [12]: s1='I\'am a boy. ' #第二个单引号前增加了\
```

```
In [13]: print(s1)
   I'am a boy.

In [14]: s2="I'am a boy. " #此处外层双引号是为了区分里面的单引号

In [15]: print(s2)
   I'am a boy.
```

转义符详见 1.4.1 字符串。

7. 变量

变量类似于数学中的变量，可以给它赋值，如：a = 3。

变量的名称只能由数字、字母和下划线构成，数字不能用在开头，字母要区分大小写；以下划线开头的变量有特殊含义，变量名不能含有空格和标点符号，如：括号、引号、逗号、斜线、反斜线、冒号、句号、问号等；在 Python 3.x 中，变量名也可以是中文，但不建议使用。

后续还会学习函数、类等概念，它们也有名称，像这些需要命名的对象称之为标识符。标识符的命名跟变量的命名一致，但又有一些约定俗成的规则，如全局变量名称中的字母一般全大写，小写的字母或单词一般表示变量或函数，类的名称一般首字母要大写。

1.3 数值运算符

Python 的数据类型有数值型、字符串 (str)、列表 (list)、元组 (tuple)、字典 (dict)、集合 (set) 等。数值型中常用的有整型 (int) 和浮点型 (float)。

1.3.1 算术运算符

算术运算符如表 1-1 所示。具体运算示例见例 1-3。

表 1-1 算术运算符

运 算 符	描 述	示 例 (a=10,b=20)
+	加法，两个对象相加	a + b = 30
−	减法，一个数减去另一个数	a − b = −10
*	乘法，两个数相乘	a * b = 200
/	除法，b 除以 a	b / a = 2
%	取模，返回除法的余数	b % a = 0
**	指数，返回 a 的 b 次幂	a**b=10000000000000000000， 10 的 20 次幂
//	取整除，返回商的整数部分	9//2 = 4 ，而 9.0//2.0 = 4.0

1.3.2 比较运算符

比较运算符如表 1-2 所示。具体运算示例见例 1-3。

表 1-2 比较运算符

运算符	描 述	示 例 (a=10,b=20)
==	判断两个操作数的值是否相等，如果相等则为真	(a == b) 为 false
!=	判断两个操作数的值是否相等，如果不相等则为真	(a != b) 为 true
>	检查左操作数的值是否大于右操作数，如果是则条件成立	(a > b) 为 false
<	检查左操作数的值是否小于右操作数，如果是则条件成立	(a < b) 为 true
>=	检查左操作数的值是否大于等于右操作数，如果是则条件成立	(a >= b) 为 false
<=	检查左操作数的值是否小于等于右操作数，如果是则条件成立	(a <= b) 为 true

1.3.3 赋值运算符

赋值运算符如表 1-3 所示。具体运算示例见例 1-3。

表 1-3 赋值运算符

运算符	描 述	示 例
=	简单的赋值运算，赋值从右操作数到左操作数	c = a + b，将 a + b 赋值给 c
+=	加法赋值操作，左操作数和右操作数和的结果赋给左操作数	c += a 相当于 c = c + a
-=	减法赋值操作，左操作数减右操作数的差赋给左操作数	c -= a 相当于 c = c - a
*=	乘法赋值操作，左操作数与右操作数的乘积赋给左操作数	c *= a 相当于 c = c * a
/=	除法赋值操作，左操作数除以右操作数的结果赋给左操作数	c /= a 相当于 c= c / a
%=	取模赋值操作符，左操作数与右操作数模的结果赋给左操作数	c %= a 相当于 c = c % a
**=	指数赋值运算符，左操作数的右操作数指数之值赋给左操作数	c **= a 相当于 c = c ** a
//=	地板除，左操作数地板除以右操作数，将结果赋给左操作数	c //= a 相当于 c = c // a

【例 1-3】各类运算示例。

```
In [1]: print(1+9) # 加法
        10

In [2]: print(1.3-4) # 减法
        -2.7

In [3]: print(3*5) # 乘法
```

```
          15

In [4]: print(4.5/1.5) # 除法
          3.0

In [5]: print(3**2) # 乘方
          9

In [6]: print(10%3) # 求余数
          1

In [7]: print(5==6) # 判断是否相等
          False

In [8]: print(8.0!=8.0) #判断是否不相等
          False

In [9]: print(3<3, 3<=3) # <小于; <=小于等于
          False True

In [10]: print(4>5, 4>=0) # > 大于; >= 大于等于
           False True

In [11]: print(5 in [1,3,5]) # 判断5是否属于 [1,3,5]
           True

In [12]: print(True and True, True and False) # and,两者都为真才是真
           True False

In [13]: print(True or False) # or, "或"运算，其中之一为真即为真
           True

In [14]: print(not True) # not, “非”运算，取反
           False

In [15]: divmod(5,2) #表示5除以2，返回商和余数
Out[15]: (2, 1)

In [16]: 5/2
Out[16]: 2.5

In [17]: a = 1.5

In [18]: b = 1

In [19]: a += b

In [20]: print(a)
           2.5
```

说明：在 Python 3.5 之前的版本中，当两个整数相除时，其结果取商的整数部分 (不是四舍五入)，当除数和被除数之一或者都是浮点数时，其结果才是浮点数。如计算 5/2 时，得到的结果不是 2.5，而是 2。但在 Python 3.5 版本中已经做了优化，其结果显示为 2.5。对浮点数的四舍五入如下所示。

```
In [21]: round(1.234567,2)     #保留小数点后两位有效数字
Out[21]: 1.23
```

但是也有例外：

```
In [22]: round(2.235,2)
Out[22]: 2.23
```

结果为什么不是 2.24？因为这是由浮点数引起的精确度问题，此类问题可由 decimal 模块来处理 (from decimal import Decimal)。

```
In [23]: from decimal import Decimal
    ...: from decimal import localcontext
    ...: a = Decimal('1.3')
    ...: b = Decimal('1.7')
    ...: print(a / b)
         0.76470588235294117647058823 53

In [24]: with localcontext() as ctx:
    ...:       ctx.prec = 3
    ...:       print(a / b)
Out[24]: 0.765
```

思考：为什么 >>>print('I love www.pylab.club ' * 5) 可以正常执行？而 >>>print('I love www.pylab.club ' + 5) 却报错？

这是因为运算类型不一致。在 Python 中不能把两个完全不同的东西加在一起，比如，数字和文本，正是这个原因，print('I love www.pylab.club ' + 5) 才会报错。这就像在说“我的体重 67 公斤加上我的身高 170cm 是多少”一样没有意义！不过文本乘以一个整数来翻倍就具有一定的意义，print('I love www.pylab.club ' * 5) 意思就是将 "I love www.pylab.club " 这个字符串连续输出 5 次。

1.4 字符串

一个单词、一句话、一首诗或一篇文章等，都可以是字符串。

1.4.1 字符串的表示

字符串是字符的序列。字符串需用单引号 (') 或双引号 ("") 括起来。

单引号 (')：可以用单引号指示字符串，就如同 'Quote me on this' 这样。单引号内的所有的空格和制表符都按照原样保留。

双引号 (")：双引号与单引号在字符串使用上完全相同，但同时使用时又有所区别，例如 "What's your name?"，但不能表示为 'What's your name? '。

三引号 (''' 或 """)：利用三引号可以指示一个多行的字符串。可以在三引号中自由地使用单引号和双引号。

转义符：假设想要在一个字符串中包含一个单引号 (')，例如，字符串 What's your name? 则不能用 'What's your name?' 来指示，因为这里有三个单引号，Python 不知引

号该从何处开始，何处结束。这时，可以通过转义符来完成这个任务，用 \' 来指示单引号——注意这里是反斜杠 (和续行符同号)。现在可以把字符串表示为 'What\'s your name?'。还可以用另一个表示方法，即用双引号与字符串内的单引号区分开，"What's your name?"。

类似地，要在双引号字符串中使用双引号时，可以借助于转义符。同样地，也可以用转义符“\\”来指示反斜杠“\”。

```
In [1]: s = 'Yes,he doesn\'t'
   ...: print(s, type(s), len(s)) #len()函数用于查看字符串s的长度
        Yes,he doesn't <class 'str'> 14
```

值得注意的是，在一个字符串中，行末单独一个反斜杠表示字符串在下一行继续——续行符。例如：

```
"This is the first sentence.\
This is the second sentence."
```

等价于：

```
"This is the first sentence. This is the second sentence."
```

转义符的种类如表 1-4 所示。

表 1-4　转义符的种类

转 义 符	描　述
\	续行符 (在行尾时)
\\	反斜杠符号
\'	单引号
\"	双引号
\b	退格 (Backspace)
\e	转义
\000	空
\n	换行
\v	纵向制表符
\t	横向制表符
\r	回车
\f	换页

原始字符串：如果想要输出字符串“\n”，而不是转义的换行，即直接按照原样输出 \n，则需要在字符串 \n 前加上前缀 r 或 R 来指定。例如，a=r"Newlines are indicated by \n"，则 print(a) 的结果输出为 "Newlines are indicated by \n"，这里最后的 \n 照原样输出显示。

如果不想让反斜杠发生转义，可以在字符串前添加 r，表示原始字符串。

```
In [2]: print('C:\some\name') #\n 表示转义成了换行符
        C:\some
        ame
```

```
In [3]: print(r'C:\some\name') #加了r表示不转义，按照原始输出
        C:\some\name
```

字符串是不可变的：这意味着一旦创造了一个字符串，就不能再改变它。虽然这看起来像是缺点，但实际上并不是。我们将会在后面的程序中讨论为什么说它不是一个缺点。

字符串可以通过运算符“+”连接在一起，或者用运算符“*”实现重复输出。

```
In [4]: print('str'+'ing', 'yubg'*3)
        string yubgyubgyubg
```

1.4.2 索引与切片

字符串一旦被确定，就不可被更改，其中的每一个字符都有自己的位置。这个位置顺序号就叫索引 (index)。有了这个索引，就可以读取字符串中的某个或某一部分字符。

1. 字符串的索引

Python 中的字符串有两种索引方式：第一种是从左往右，从 0 开始依次增加，称为顺序索引；第二种是从右往左，从 -1 开始依次减少，称为逆序索引。例如，python 有 6 个字母，取第一个字母 p 可以是按顺序索引取索引号为 0，或者按逆序索引取索引号为 -6，索引号必须用中括号 [] 括起来，示例如下。

```
In [1]: word = 'Python'

In [2]: print(word[0])
        P

In [3]: print(word[-6] ,word[-1])
        P n
```

2. 字符串的切片

切片就是从给定的字符串中截取一段。Python 中用冒号分隔起止两个索引，即截取的这一段子串开始位置的索引和结束位置的索引，形式为 var[开始索引：结束索引]，注意截取的范围是左闭右开，即不包含结束索引的字符，所以截取包含结束位置就需要后移一位。若从开始位置截取到最右边，则结束索引可以省略，同样，若从最左边开始截取索引号为 0，则开始索引也可以省略。

```
In [12]: str_0 = "Hello My friend"

In [4]: print(str_0[1:4]) #截取索引号1到3三个字符，注意要后移一位
        ell

In [5]: print(str_0[:-7]) #从开头位置截取
        Hello My

In [6]: print(str_0[5:]) #截取到最后
        My friend
```

```
In [7]: print(str_0[:]) #表示从开始到结束都截取
        Hello My friend
```

切片的扩展形式如下。

str[I:J:K] 表示从 str 字符串中截取从 I 到 J−1(包含 J−1)，且每隔 K−1 个元素依次序取一次，K 默认为 1，可以省略不写。如果 K 为负数，表示按从右往左开始取，但起止索引必须是逆序。

```
In [8]: print(str_0[:]) #表示从头到尾都截取
        Hello My friend

In [9]: print(str_0[2:7:2])
        loM

In [10]: print(str_0[-8:-13:-1])
         yM ol
```

字符串可以使用包含判断操作符：in，not in。

```
In [11]: str_0 = "Hello My friend"
    ...: "He" in str_0
Out[11]: True

In [12]: "she" not in str_0
Out[12]: True

In [13]: str_0.find('o') #字符串模块提供的查找方法
Out[13]: 4
```

ord 函数是将字符转化为对应的 ASCII 码值，而 chr 函数是将数字转化为字符。

```
In [14]: print(ord('a'))
          97

In [15]: print(chr(97))
          a
```

处理字符串的内置函数有 len()、max()、min() 等，这些函数对后续学习的 list(列表)、dict(字典)、set(集合) 等类型也适用。

```
In [16]: len(str_0)      #测变量str_0的长度
Out[16]: 15

In [17]: max('abcxyz') #寻找字符串中最大的字符
Out[17]: 'z'

In [18]: min('abcxyz') #寻找字符串中最小的字符
Out[18]: 'a'
```

数值型字符串的转换。

```
In [19]: s = "12"
    ...: ss = int(s) #将s转化为整型
    ...: type(ss)
Out[19]: int
```

```
In [20]: ff = "4.9"
    ...: type(float(ff))   # float(ff)将ff转化为浮点型
Out[20]: float
```

为字符串还提供了其他的查找方法，如：

```
str.find(substring,[start[,end]])    #在指定范围内查找子串，返回找到的第一个索引值
str.rfind(substring,[start [,end]])   #反向查找
str.index(substring,[start [,end]])   #同find，但是找不到时产生Value Error异常
str.rindex(substring,[start [,end]]) #同上反向查找
str.count(substring,[start [,end]])   #返回找到子串的个数
str.capitalize()                     #首字母大写
str.lower()                           #转小写
str.upper()                           #转大写
str.swapcase()                       #大小写互换
str.split()          #将str默认以空格切分(也可以指定字符)，返回切分后的列表
```

注意：find(x) 返回的是找到的第一个 x 的索引，否则返回 −1。例如，"abcad".find("a")，返回的是第一个 a 的索引 0。

1.5 列表、元组、字典、集合

1.5.1 列表

列表 (list) 用方括号 [ ] 标识，其元素写在方括号之中并用逗号分隔开。列表中元素的类型可以不相同，甚至也可以是列表、元组、字典等。列表形式如：["I","you","he",5] 。

列表的索引与字符串相同，如图 1-5 所示。

对于长度为 7 的列表 lis = [1, 2, 5, 3, 6, 8, 4]，lis[0] 和 lis[−7] 都表示取的是 lis 中的第一个元素 1。

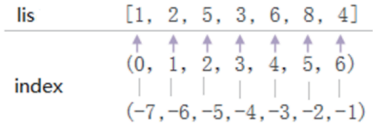


图 1-5 list 索引

```
In [1]: lis = [1, 2, 5, 3, 6, 8, 4]

In [2]: lis[0]
Out[2]: 1

In [3]: lis[-7]
Out[3]: 1

In [4]: len(lis)
Out[4]: 7

In [5]: max(lis)
Out[5]: 8
```

```
In [6]: min(lis)
Out[6]: 1
```

创建连续元素的列表，如 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]，当然可以手工输入，但如果需要输出一个从 1 到 1000 的列表呢？手工输入太耗时费力了，这时需要用到一个函数 range()。

Python 中的内置函数 range(m,n)，表示一个从 m(m 若从 0 开始可忽略不写) 到 n−1 的一个序列 (有序排列)，即同字符串切片一样，左闭右开，最后一位取不到。如要表示前文所说的 1 到 10 包含十个元素的序列，则 range(1,11)，这样则产生包含从 1 到 10 十个元素的序列。

但要注意，range(1,11) 输出的并不是列表。

```
In [7]: range(3)
Out[7]: range(0, 3)

In [8]: range(1,11)
Out[8]: range(1, 11)
```

从上面语句的输出可以看出，输出的并不是列表。其实这里的 range() 函数是一个容器，生成的这个容器里包含所需要的各个元素，当需要它是列表时，用 list 去调用即可，即 list(range(1,11))，当需要它是元组时就用 tuple() 去调用它。

```
In [9]: list(range(1,11))
Out[9]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

range() 函数还可以定义步长。下面定义一个从 1 开始到 30 结束，步长为 3(每间隔两个元素取一次) 的列表 [1, 4, 7, 10, 13, 16, 19, 22, 25, 28]。可以使用 range(start,stop, step) 形式实现，类似于切片 (切片用冒号，这里用逗号)，start 表示开始值，stop 表示结束值 (stop 数取不到)，step 表示步长，step 默认为 1，可省略不写，产生的是依次递增的自然数序列。range(n) 就是 range(0, n, 1) 的省略形式。

```
In [10]: list(range(1, 30, 3))  #步长为3，即每间隔两个数取一个值
Out[10]: [1, 4, 7, 10, 13, 16, 19, 22, 25, 28]
```

列表和字符串一样，列表同样可以被索引和切片，列表被切片后返回一个包含所取元素的新列表。

```
In [11]: a = ["I","you","he",5]

In [12]: a[1:3]
Out[12]: ['you', 'he']

In [13]: a[1]
Out[13]: 'you'
```

列表的元素是可以修改的，直接对其索引赋值即可。

```
In [14]: a[1]='she'

In [15]: a
Out[15]: ['I', 'she', 'he', 5]
```

从上面语句可以看出，a[1] 元素由原来的 you 变成了 she。

列表还支持合并操作，使用“+”操作符。

```
In [16]: a = [1, 2, 3, 4, 5]

In [17]: a + [6, 7, 8]
Out[17]: [1, 2, 3, 4, 5, 6, 7, 8]
```

也可在列表指定的位置插入一个元素，如列表 L 中 n 位置插入一个值“q”，可执行 L[n:n]=['q'] 语句。

```
In [18]: a = [1, 2, 4, 5]

In [19]: a[2:2]=[3]          #在列表中2的位置插入3

In [20]: a
Out[20]: [1, 2, 3, 4, 5]
```

list() 可将字符串转化为列表：list(str)

```
In [21]: word='hello'

In [22]: list(word)          #将字符串转化为列表
Out[22]: ['h', 'e', 'l', 'l', 'o']
```

列表 (list) 的其他常用方法如下。

1) 列表 (list) 的元素追加

L.append(var) 表示追加元素，追加的元素可以是一个数值、字符串甚至 list 等。

```
In [23]: w=[1,2,'L']

In [24]: w.append("y") #不能写成w= w.append("y")

In [25]: w
Out[25]: [1, 2, 'L', 'y']

In [26]: q=[8,9]

In [27]: w.append(q)

In [28]: w
Out[28]: [1, 2, 'L', 'y', [8, 9]] #q列表的整体转化为w的一个元素
```

注意：append 方法不能返回值，所以 w= w.append() 这样的写法是错误的。

L.extend(list) 合并两个列表，即把 list 中的元素合并到 L 列表中。

```
In [29]: w=[1,2,'L']
    ...: q=[8,9]
    ...: w.extend(q)

In [30]: w
Out[30]: [1, 2, 'L', 8, 9]
```

注意：append 和 extend 的区别是：append 是对列表添加元素，extend 是合并两个列表。

L.insert(index,elm) 在 index 位置插入 elm 元素。

```
In [31]: a = ["I","you","he",5]
    ...: a.insert(1,'love')
    ...: a
Out[31]: ['I', 'love', 'you', 'he', 5]
```

2) 从列表 (list) 中删除元素

L.pop(index) 表示从列表 (list) 中删除 index 位置的元素，返回被删除的元素。默认删除最后一个元素。

```
In [32]: a = ["I","you","he",5]
    ...: a.pop(2)       #返回被删除的元素
Out[32]: 'he'

In [33]: a
Out[33]: ['I', 'you', 5]

In [34]: a.pop()       #默认删除最后一个元素
Out[34]: 5

In [35]: a
Out[35]: ['I', 'you']
```

删除指定索引位置的元素：del L[index]

```
In [36]: a = ["I","you","he",5]
    ...: del a[3]

In [37]: a
Out[37]: ['I', 'you', 'he']
```

删除指定索引范围的元素：del L[1:3]

```
In [38]: a = [1, 2, 3, 4, 5]
    ...: del a[1:3]       #删除list中索引为1、2的元素

In [39]: a
Out[39]: [1, 4, 5]
```

删除第一次出现的 elm 元素：L.remove(elm)

```
In [40]: li = [1,2,3,4,5,4,6]
    ...: li.remove(4)       #删除从顺序索引开始的第一个值为4的元素

In [41]: li
Out[41]: [1, 2, 3, 5, 4, 6]
```

另外，还可以使用切片的方法来删除，即重新创建一个列表。

```
In [42]: li = [1,2,3,4,5,6]
    ...: li = li[:-1]

In [43]: li
Out[43]: [1, 2, 3, 4, 5]
```

3) 列表 (list) 的操作符 :，+，*，

```
In [44]: a = [1, 4, 5]
    ...: a[1:]                #切片操作符，用于提取一个新的列表
Out[44]: [4, 5]

In [45]: [1,2] + [3,4]      #同[1,2].extend([3,4])
Out[45]: [1, 2, 3, 4]

In [46]: [2]*4                #复制元素4次
Out[46]: [2, 2, 2, 2]
```

4) 列表 (list) 的索引冒号用法

冒号前后表示索引切片的起止位置。

```
In [47]: array = [1, 2, 5, 3, 6, 8, 4]

In [48]: array[0:] #列出索引0以后的元素
Out[48]: [1, 2, 5, 3, 6, 8, 4]

In [49]: array[1:] #列出索引1以后的元素
Out[49]: [2, 5, 3, 6, 8, 4]

In [50]: array[:-1] #列出索引-1之前的元素
Out[50]: [1, 2, 5, 3, 6, 8]

In [51]: array[3:-3] #列出索引3到索引-3之间的元素
Out[51]: [3]
```

两个冒号 [::]表示取全部索引。

```
In [52]: array[::2] #表示按步长为2取元素，即隔一个元素取一个元素
Out[52]: [1, 5, 6, 4]

In [53]: array[2::]
Out[53]: [5, 3, 6, 8, 4]

In [54]: array[::3]
Out[54]: [1, 3, 4]

In [55]: array[::4]
Out[55]: [1, 6]
```

双冒号 [::]还可以形成 reverse 函数的效果。

```
In [56]: array[::-1]              #倒序
Out[56]: [4, 8, 6, 3, 5, 2, 1]

In [57]: array[::-2]
Out[57]: [4, 6, 5, 1]
```

5) 对列表 (list) 排序

对列表进行排序可以使用 sort() 方法和 sorted() 函数。

(1) sort() 方法。使用 sort() 方法对列表进行排序时，会改变列表本身，从而让其中的元素按一定的顺序排列。

```
In [58]: a = [3,2,5,4,9,8,1]
    ...: a.sort()

In [59]: a
Out[59]: [1, 2, 3, 4, 5, 8, 9]

In [60]: sort(a)
Traceback (most recent call last):
File "C:\Users\yubg\AppData\Local\Temp/ipykernel_17828/2106328678.py",
   line 1, in <module>
sort(a)
        NameError: name 'sort' is not defined

In [61]: help(a)
… … …
sort(...)
   L.sort(key=None, reverse=False) -> None -- stable sort *IN PLACE*
```

从 help 可知，sort() 默认是从小到大排序，可以添加参数 reverse=True，改为从大到小排列。

```
In [62]: e=[1, 3, 2, 4, 5]
    ...: e.sort(reverse=True)    #按逆序排序

In [63]: e                              #e被改变
Out[63]: [5, 4, 3, 2, 1]
```

注意：sort() 使用的是“.”方法，即 a.sort() ，使用 sort(a) 就会报错。sort() 方法会改变原来的列表，且返回值为 None。因此，如果需要一个排好序的列表，同时又要保留原有列表，就不能直接简单地使用 sort() 方法。为了实现上述功能，可以使用 sorted(a) 函数。

(2) sorted() 函数。sorted() 函数对列表进行排序时，直接获取列表排序的一个副本。sorted() 函数可以用于任何可迭代的对象。

```
In [64]: a = [3,2,5,4,9,8,1]
    ...: b = sorted(a)                  #对a排序后产生一个新的列表
    ...: b
Out[64]: [1, 2, 3, 4, 5, 8, 9]

In [65]: a                                  #a不被改变
Out[65]: [3, 2, 5, 4, 9, 8, 1]
```

注意：

① a.sort() 和 sorted(a) 的区别：sorted(a) 产生的是一个新列表，不改变原列表 a；而 a.sort() 是对列表 a 直接排序，破坏了原列表。

② sorted() 是产生新的排序列表不改变原列表，这个功能也可以通过拷贝副本的方法来实现：先获取列表 a 的一个副本 b，然后再对 b 进行 b.sort() 排序。为了更深刻地理解，我们将对 a 和 b 的存储地址进行查验，代码如下。

```
In [66]: a = [3,2,5,4,9,8,1]
    ...: id(a) #查验a的存储地址
Out[66]: 1715894984384
```

```
In [67]: b=a[:] #拷贝一个副本b
    ...: b
Out[67]: [3, 2, 5, 4, 9, 8, 1]

In [68]: id(b) #查验b的存储地址
Out[68]: 1715895012224    #发现b和a地址不一致，说明复制了一份

In [69]: c=a #再复制一个副本c，比较c和a的存储地址
    ...: c
Out[69]: [3, 2, 5, 4, 9, 8, 1]

In [70]: id(c) #查验c的存储地址
Out[70]: 1715894984384

In [71]: b.sort() #对b进行排序

In [72]: b
Out[72]: [1, 2, 3, 4, 5, 8, 9]

In [73]: a #b排序后对a没有影响
Out[73]: [3, 2, 5, 4, 9, 8, 1]

In [74]: c.sort() #对c进行排序

In [75]: c
Out[75]: [1, 2, 3, 4, 5, 8, 9]

In [76]: a #c排序后对a有影响
Out[76]: [1, 2, 3, 4, 5, 8, 9]
```

从上面代码显示的内存存储地址可知，c 仅仅是 a 的一个标签，并不是真正意义上的复制，不论是 a 改变，还是 c 改变，其实改变的都是同一个地址里的内容，所以互相有影响。只有 b 才是真正意义上的复制，后面我们还会遇到“深复制”。

说明：调用 x[:] 得到的是包含了 x 所有元素的切片，这是一种复制整个列表很有效率的方法。通过 y=x 简单地将 x赋值给 y，仅仅是给 y“贴”了一个指向 x 的标签，最终 x 和 y 都指向了同一个列表。

(3) reverse() 函数。

```
L.reverse()      #倒序排列
In [77]: e.reverse()

In [78]: e
Out[78]: [1, 2, 3, 4, 5]
```

或者：

```
In [79]: reversed([1,2,'L']) #返回一个迭代器，可以用list转化为列表
Out[79]: <list_reverseiterator at 0x18f821fe670>

In [80]: list(reversed([1,2,'L']))
Out[80]: ['L', 2, 1]
```

或者：

```
In [81]: w=[1,2,'L']
    ...: w[::-1]
Out[81]: ['L', 2, 1]
```

对字符串也可以同样反转 (即逆序排序)。

6) 其他方法

```
L.count(elm)        #统计该元素在列表中出现的次数
L.index(elm)        #返回第一个elm元素的位置,如果无则抛出异常
```

list 对象常用的方法如表 1-5 所示。

表 1-5　list 对象常用的方法

方　法	描　述
list.append(x)	把一个元素添加到列表的结尾，相当于 a[len(a):] = [x]
list.extend(L)	将一个给定列表中的所有元素都添加到另一个列表中，相当于 a[len(a):] = L
list.insert(i, x)	在指定的索引位置 i 插入一个元素 x。如 a.insert(0, x) 会插入到整个列表之前，而 a.insert(len(a), x) 相当于 a.append(x)
list.remove(x)	删除列表中第一次出现 x 的元素。如果没有这样的元素，则返回一个错误
list.pop(i)	从指定索引位置删除元素，并将其返回。如果没有指定索引，a.pop() 默认删除最后一个元素并返回被删除的元素
list.index(x)	返回列表中第一个值为 x 元素的索引。如果没有匹配的元素就会返回一个错误
list.count(x)	返回 x 在列表中出现的次数 (可以用于列表中的查重)
list.sort( )	对列表中的元素就地进行排序 (改变了原列表)
list.reverse( )	逆序排列列表中的元素 (改变了原列表)

注意：L.index(x) 返回的是第一个 x 元素的索引，并非所有 x 元素的索引。如 [1,2,1,3].index(1)，返回的是第一个 1 的索引 0。

1.5.2　元组

元组 (tuple) 是列表的特殊形式——不能被修改，所以其表现形式是仅将列表的中括号改为小括号即可。

```
In [1]: a = (1991, 2014, 'physics', 'math')
   ...: print(a, type(a), len(a))
        (1991, 2014, 'physics', 'math') <class 'tuple'> 4
```

tuple 可以被索引，也可以切片。其实，也可以把字符串看作一种特殊的元组。

```
In [2]: tup = (1, 2, 3, 4, 5, 6)
   ...: print(tup[0], tup[1:5])
        1 (2, 3, 4, 5)

In [3]: tup[0] = 11              #元组一旦生成是不可以被修改的
Traceback (most recent call last):
File "C:\Users\yubg\AppData\Local\Temp/ipykernel_17828/2907376088.py",
   line 1, in <module>
tup[0] = 11
TypeError: 'tuple' object does not support item assignment
```

虽然 tuple 的元素不可改变，但它可以包含可变的对象，比如 list(列表)。对于构造包含 0 个或 1 个元素的 tuple 是个特殊的问题，所以有一些额外的语法规则。

```
In [4]: tup1 = () #空元组
   ...: tup1
Out[4]: ()

In [5]: tup2 = (20,) #创建只有一个元素的元组，该元素后面的逗号不可忽略
   ...: tup2
Out[5]: (20,)

In [6]: tup3 = (20)
   ...: tup3
Out[6]: 20

In [7]: type(tup3)     #不加逗号结果为数值型
Out[7]: int
```

注意：tuple(2) 其实就是数字 2，仍然是整型，但是 tuple (2,) 是 tuple。元组是不能被修改的，所以没有添加、删除操作。另外，元组之间支持使用“+”操作符。

```
In [8]: tup1, tup2 = (1, 2, 3), (4, 5, 6) #注意python支持这种同时赋值
   ...: print(tup1+tup2)
        (1, 2, 3, 4, 5, 6)
```

元组的每个元素可以存储不同类型的数据，而元组中的元素则代表不同的数据项。元组和字符串一样，一旦创建，都是不可修改的。

元组的访问同列表。

```
In [9]: user = ('01','02','03','04')
   ...: user[0]
Out[9]: '01'

In [10]: user[2]
Out[10]: '03'
```

元组包含了以下内置函数。

len(tuple)：计算元组中元素的个数。

max(tuple)：返回元组中元素的最大值。

min(tuple)：返回元组中元素的最小值。

tuple(list)：将列表转换为元组。

list(tuple)：将元组转换为列表。

二元元组 (二维) 的访问：

```
In [11]: user1 = (1,2,3)
    ...: user2 = (4,5,6)
    ...: user = (user1,user2)
    ...: print(user[1][2]) #提取user中的数值6，位于user中索引为1的元素中索引为2
的位置上是数值6
```

元组的解包：

```
In [12]: user = (1,2,3)
```

```
    ...: a,b,c = user          #变量个数要等于元组的长度

In [13]: a
Out[13]: 1

In [14]: b
Out[14]: 2

In [15]: c
Out[15]: 3
```

1.5.3 字典

先看看列表存储通讯录。

【例 1-4】通讯录的存储。

```
In [1]: name=["Ben","Jone","Jhon","Jerry","Anny","Ivy","Jan","Wong"]
   ...: tel=[6601,6602,6603,6604,6605,6606,6607,6608]
```

这里通讯录存储在两个列表中 (一个列表存储姓名，一个列表存储对应的手机号码)，但是要查阅某个人的电话号码，显得很不方便。为了便于查阅，在 Python 中有另外一种存储方式：字典。

字典是一种映射类型 (mapping type)，它是一个无序的“键：值”对 (pair) 的集合。每一个元素是一对，包含关键字 (key) 和值 (value) 两部分。key 是不可变类型，value 是任意类型。即：

```
{Key: Value}
```

关键字 (key) 必须使用不可变类型，在同一个字典中，关键字必须互不相同。

```
In [2]: dic = {} #创建一个空字典

In [3]: dic_tel = {'Jack':"Man", 'Tom':1320, 'Rose':"Good"}   #创建一个字典

In [4]: print(dic_tel) #打印字典
        {'Jack':"Man", 'Tom':1320, 'Rose':"Good"}
```

例 1-4 可改用字典来存储通讯录。下面用遍历的方法来操作 (遍历的内容详见第 2 章)，其原理是：每次从 name 中取一个姓名记为 n1，再从 tel 中取对应的号码记为 t1，再把 n1 和 t1 组成键值对 n1:t1，成为字典 Tellbook 中的一个元素，对列表 name 和列表 tel 中的所有元素如此循环，就全部构成了字典的元素。

【例 1-5】列表转化成字典。

```
In [5]: name=["Ben","Jone","Jhon","Jerry","Anny","Ivy","Jan","Wong"]
   ...: tel=[6601,6602,6603,6604,6605,6606,6607,6608]
   ...: Tellbook={}                          #创建一个空字典
   ...: for i in range(len(name)):
   ...: d1="{}".format(name[i])      #从name中取一个姓名
   ...: d2="{}".format(tel[i])       #从tel中取一个电话
   ...: Tellbook[d1]=d2              #再把d2赋值给字典Tellbook的d1键
   ...: print(Tellbook)
```

```
   {'Ben': '6601', 'Jone': '6602', 'Jhon': '6603', 'Jerry': '6604', 'Anny':
'6605', 'Ivy': '6606', 'Jan': '6607', 'Wong': '6608'}
```

1) 字典的增删改查

字典的一些常用操作方法如下。

```
Tellbook['Wang'] =3          #修改键值，若键不存在则直接创建此键(创建一个新元素)
del Tellbook['Wong']         #删除一个键值对
Tellbook['Ben']              #通过key查询对应的值
list(Tellbook.keys())        #返回所有key组成的list
list(Tellbook.values())      #返回所有value组成的list
sorted(Tellbook.keys())      #按key对字典排序
'Ben' in Tellbook            #成员测试
'Mary' not in Tellbook       #成员测试
```

构造函数 dict()，直接从键值对构建字典，语法格式如下。

```
In [6]: dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
Out[6]: {'sape': 4139, 'guido': 4127, 'jack': 4098}

In [7]: dict(sape=4139, guido=4127, jack=4098)
Out[7]: {'sape': 4139, 'guido': 4127, 'jack': 4098}
```

字典有 .items 方法：将字典里的每一个元素 (键值对) 转化为二元元组作为列表的一个元素。

```
In [8]: d= {'a':1, 'b':2, 'c':3}
   ...: t= d.items()
   ...: print(t)
dict_items([('a', 1), ('b', 2), ('c', 3)])

In [9]: list(t)       #用list调用，即转化为list
Out[9]: [('a', 1), ('b', 2), ('c', 3)]
```

当然，上面的过程是可逆的，即二元元组列表可以初始化成字典：

```
In [10]: t=[('a',1),( 'b',2),( 'c',3)]
    ...: d=dict(t)
    ...: print(d)
         {'a': 1, 'b': 2, 'c': 3}
```

用 update 可以合并两个字典：

```
In [11]: dict1 = {'Name': 'Zara', 'Age': 7}
    ...: dict2 = {'Sex': 'female' }
    ...: dict1.update(dict2)

In [12]: dict1
Out[12]: {'Name': 'Zara', 'Age': 7, 'Sex': 'female'}
```

用 tuple 作为键可以创建字典：

```
In [13]: seq = ('name', 'age', 'sex')
    ...: dict1 = dict1.fromkeys(seq) #字典的键(key)来自元组seq的元素
    ...: dict1       #因为仅有key，没有value，所以显示键值为空None
Out[13]: {'name': None, 'age': None, 'sex': None}
```

```
In [14]: dict1 = dict1.fromkeys(seq, 10) #给字典键seq赋值，都赋值为10
    ...: dict1
Out[14]: {'name': 10, 'age': 10, 'sex': 10}
```

字典的“.”方法：

```
D.get(key, 0)       #同dict[key]，此处参数值为0(也可以是其他值)，表示若没有key键则返回0
D.keys()            #返回字典键的列表
D.values()          #返回字典值的列表
D.items()           #将字典转化为元组作为元素的一个列表
D.update(dict2)     #合并字典，将dict2增加到当前的字典中
D.pop(key)          #从字典中删除指定的键值对，键名这个参数必须有
D.popitem()         #没有参数则从字典中随机删除一个键值对。已空则抛出异常
D.clear()           #清空字典，不同于命令del dict，del是删除字典
D.copy()            #复制字典
```

示例如下：

```
In [1]: dict0={'sex': 10, 'age': 10, 'name': 10}
   ...: dict0.get('sex', 'None')     #若没有sex键，则返回指定的None
Out[1]: 10

In [2]: dict0.keys()      #要想获取键名列表，直接使用list(dict0.keys())即可
Out[2]: dict_keys(['sex', 'age', 'name'])

In [3]: dict0.items() #要想获取键值对列表，直接使用list(dict0.keys())即可
Out[3]: dict_items([('sex', 10), ('age', 10), ('name', 10)])

In [4]: dict0.pop('sex') #删除sex键值对
Out[4]: 10

In [5]: dict0
Out[5]: {'age': 10, 'name': 10}

In [6]: dict0['sex']=10 #增加键值对sex: 10

In [7]: dict0
Out[7]: {'age': 10, 'name': 10, 'sex': 10}

In [8]: dict0.popitem() #随机删除一个键值对
Out[8]: ('sex', 10)

In [9]: dict0
Out[9]: {'age': 10, 'name': 10}

In [10]: dict1=dict0.copy() #复制(浅拷贝)一个字典，浅拷贝只对简单类型拷贝
    ...: dict1
Out[10]: {'age': 10, 'name': 10}

In [11]: dict0.clear() #清空字典，即得到一个空字典

In [12]: dict0 #dict已经变成了一个空字典
Out[12]: {}

In [13]: dict1
Out[13]: {'age': 10, 'name': 10}
```

【例 1-6】字典内置 get 方法的调用。

假设用户在终端输入字符串 "1"、"2" 或 "3"，则返回对应的内容，如果输入其他内容，则返回 "error"。

```
In [14]: info = {'1':'first','2':'second','3':'third'}
In [15]: info.get(input('input type you number:'),'error')

input type you number:2
Out[15]: 'second'
```

D.get(key, None) 的用法：表示字典 D 中若有 key 键，则返回其键值；若没有 key 键，则返回指定的值 None。当然，这里的 None 也可以改写成别的，如 D.get(key, '哈哈，别逗，你输错了！')，当没有 key 键时，则返回：'哈哈，别逗，你输错了！'。代码如下：

```
In [16]: info = {'1':'first','2':'second','3':'third'}
    ...: info.get(input('input type you number:'),'哈哈，别逗，你输错了！')

input type you number:5
Out[16]: '哈哈，别逗，你输错了！'
```

2) 字典的排序

在程序中使用字典进行数据信息统计时，字典是无序的，因此，打印输出的字典内容也是无序的。因此，为了方便结果查看，需要对字典进行排序。Python 中字典的排序分为按 Value 值排序和按 Key 键排序。

(1) 按 Value 值排序。按“值”排序就是根据字典的 Value 值进行排序，可以使用内置的 sorted() 函数。

```
In [17]: dict3={'班级': 1, 'age': 10, 'score': 10}
    ...: sorted(dict3.items(), key=lambda e:e[1], reverse=True)
Out[17]: [('age', 10), ('score', 10), ('班级', 1)]
```

其中 e 表示 dict3.items() 中的元素，e[1] 则表示按值排序。Reverse= False 可以省略，默认为升序排列。

说明：匿名函数 lambda, 后文会专门介绍。

(2) 按 Key 键排序。对字典进行按键排序也可以使用 sorted() 函数，只要修改为 sorted(dict.items(), key=lambda e:e[0], reverse=True) 即可。

1.5.4 集合

集合 (set) 是一个无序不重复元素的集，set 的基本功能是去重。使用大括号 {} 或者 set() 函数创建 set 集合。

注意：创建一个空集合必须用 set()，不能使用 { }，因为 { } 是用来创建一个空字典。

```
In [1]: student = {'Tom', 'Jim', 'Mary', 'Tom', 'Jack', 'Rose'} #有重复元素
   ...: print(student)                                  #重复的元素被自动去掉
Out[1]: {'Tom', 'Mary', 'Jack', 'Rose', 'Jim'}

In [2]: 'Rose' in student      #membership testing(成员测试)
Out[2]: True
```

```
In [3]: student.add('Ben')      #增加一个元素

In [4]: print(student)
{'Tom', 'Mary', 'Jack', 'Ben', 'Rose', 'Jim'}
```

集合的运算。

```
In [1]: a = set('abracadabra') #将字符串拆成集合
   ...: a
Out[1]: {'a', 'b', 'c', 'd', 'r'}

In [2]: b = set('alacazam') #将字符串拆成集合
   ...: b
Out[2]: {'a', 'c', 'l', 'm', 'z'}

In [3]: a-b #从a中去除b的元素
Out[3]: {'b', 'd', 'r'}

In [4]: a|b #a和b的并集
Out[4]: {'a', 'b', 'c', 'd', 'l', 'm', 'r', 'z'}

In [5]: a&b #提取a和b的公共元素——交集
Out[5]: {'a', 'c'}

In [6]: a^b #提取a和b中不同时存在的元素(交集的补集，也叫对称差)
Out[6]: {'b', 'd', 'l', 'm', 'r', 'z'}
```

集合有过滤重复元素的功能，自动将重复元素删除。

```
In [7]: set((2,2,2,4,4))
Out[7]: {2, 4}
```

1.6 格式化输出

1.6.1 % 输出

Python 用 print() 进行格式化输出，有以下几种模式，代码如下。

1) 输出字符串

```
In [1]: print("His name is %s"%("Aviad"))
        His name is Aviad
```

输出的内容里 ("His name is %s"%("Aviad")) 有两个 %，其中第一个 % 表示先在这个字符串中占一个位置，而第二个 % 表示的是其后的 "Aviad" 是赋值给第一个 % 位置上要显示的内容，也就是 %s 位置上要显示的内容在第二个 % 后面的括号 () 内。%s 表示占位输出字符串。

2) 输出整数

```
In [2]: print("He is %d years old"%(25))
        He is 25 years old
```

%d 表示占位输出整数。

3) 输出浮点数

```
In [3]: print("His height is %f m"%(1.83))
        His height is 1.830000 m
```

%f 表示占位输出浮点数。

4) 输出指定保留小数点位数的浮点数

```
In [4]: print("His height is %.2f m"%(1.836))
        His height is 1.84 m
```

这里的 %.2f 表示显示到小数点后两位数字，也就是指定了保留小数点位数。

5) 输出指定占位符的宽度 (右对齐)

```
In [5]: print("Name:%10s Age:%8d Height:%8.2f"%("Aviad",25,1.83))
        Name:     Aviad Age:      25 Height:   1.83

In [6]: print('i love %.3s'%'python')
        i love pyt
```

%10s 表示占位 10 个字符右对齐，不够的左边用空格替补；%8.2f 表示占位 8 个位置且保留两位小数的浮点数，不够的用空格替补；%8d 表示占位 8 个位置的整数，不够的用空格替补；%.3s 表示截取字符串的前 3 个字符，不够 3 个的则全部截取。

6) 输出指定占位符的宽度 (左对齐)

```
In [7]: print("Name:%-10s Age:%-8d Height:%-8.2f"%("Aviad",25,1.83))
        Name:Aviad      Age:25       Height:1.83
```

%-10s 表示占位 10 个字符左对齐，不够的右边用空格补齐。%-8d 和 %-8.2f 同理。

7) 指定占位符 (0 或者空格)

```
In [8]: print("Name:%-10s Age:%08d Height:%08.2f"%("Aviad",25,1.83))
        Name:Aviad      Age:00000025 Height:00001.83
```

%08.2f 表示占位 8 个位置且保留两位小数的浮点数，不够的左边用 0 补齐。

1.6.2 format 输出

格式化字符串函数 format()，功能强大。format() 函数跟 % 格式化字符串相比，其优越性是通过 {} 来占位，示例如下：

```
In [1]: '{0} is {1}'.format('yubg',39) #这里的0和1表示的是占位位置的索引
Out[1]: 'yubg is 39'

In [2]: '{} is {}'.format('yubg',39) #位置索引也可以为空，占位位置与format()
          给定的参数一一对应
Out[2]: 'yubg is 39'

In [3]: '{1},{0},{1}'.format('a',5) #可以接受多个参数，位置可以无序
Out[3]: '5,a,5'
```

format() 的关键字参数：

```
In [4]: '{name} is {age}'.format(age=39,name='yubg')
Out[4]: 'yubg is 39'
```

有多个输出需要多个占位符时，可以通过设定下标以区分：

```
In [5]: p=['yubg',39]
   ...: '{0[0]},{0[1]}'.format(p)
Out[5]: 'yubg,39'
```

格式限定符：它有着丰富的格式，比如填充和对齐，语法为 {} 中带：号。填充和对齐常一起使用。

^：居中，后面带宽度。

<：左对齐，后面带宽度。

>：右对齐，后面带宽度。

“:”：后面带填充的字符，只能是一个字符，如果不指定默认是用空格填充。

```
In [6]: '{:>8}'.format('189') #默认是空格来占位，要显示的内容靠右对齐
Out[6]: '     189'

In [7]: '{:0>8}'.format('189') #用0来占位
Out[7]: '00000189'

In [8]: '{:a<8}'.format('189') #用字母a来占位，要显示的内容靠左对齐
Out[8]: '189aaaaa'

In [9]: '{:*^7}'.format('189') #用*来占位，共显示7位，要显示的内容居中
Out[9]: '**189**'
```

精度与类型 f：精度常跟浮点数类型一起使用，表示保留小数点后的有效数字位数。

```
In [10]: '{:.2f}'.format(321.33345) #保留小数点后两位有效数字
Out[10]: '321.33'
```

1.6.3 f 输出

F 或 f 是 Python 高版本才有的输出方式。

```
In [1]: name="yubg"
   ...: f"my name is %s."%name
Out[1]: 'my name is yubg.'
```

1.7 其他常用的函数

1.7.1 strip() 函数

strip、lstrip、rstrip 函数的使用方法如下：

strip：去掉字符串两边的空格。

lstrip：去掉左边的空格。

rstrip：去掉右边的空格 。

Python 中的 strip() 函数用于去除字符串的首尾字符，同理，lstrip() 函数用于去除左边的字符，rstrip() 函数用于去除右边的字符。这三个函数都可以传入一个参数，指定要去除的首尾字符。需要注意的是，传入的是一个字符数组时，编译器去除两端所有相应的字符，直到没有匹配的字符为止，示例如下。

```
In [1]: theString = 'saaaay yes or no yaaaass'
   ...: print(theString.strip('say'))#yes的左边有一个空格，no的右边有一个空格
        yes or no

In [2]: theString
Out[2]: 'saaaay yes or no yaaaass'
```

theString 依次被去除首尾在 ['s'，'a'，'y'] 列表内的字符，直到字符不在数组内，所以左边的 saaaay 都由 s、a、y 构成，即 s、a、y 都属于 ['s'，'a'，'y']，直到 yes 左边的空格；同理右边。所以，输出的结果为：“ yes or no ”。当然，这里生成的是一个“副本”，不会改变原来的字符串 theString。

lstrip 和 rstrip 原理一样。当没有传入参数时，是默认去除首尾的空格。

```
In [3]: print(theString.strip('say') )
Out[3]: yes or no

In [4]: theString.strip('say')
Out[4]: ' yes or no '

In [5]: theString.strip('say ')
Out[5]: 'es or no'

In [6]: theString.lstrip('say')
Out[6]: ' yes or no yaaaass'

In [7]: theString.rstrip('say')
Out[7]: 'saaaay yes or no '
```

1.7.2 split() 函数

split() 函数的作用是对字符串的分割。

(1) 按某个字符分割，如‘.’。

```
In [1]: str = ('www.pylab.club')
   ...: print(str)
        www.pylab.club

In [2]: str_split = str.split('.')
   ...: print(str_split)
        ['www', 'pylab', 'club']
```

(2) 按某个字符分割，且分割 n 次。如按‘.’分割 1 次。

```
In [3]: str = ('www.pylab.club ')
```

```
   ...: str_split = str.split('.',1)
   ...: print(str_split)
        ['www', 'pylab.club']
```

(3) 按某一字符 (字符串) 分割，且分割 n 次，并将分割完成的字符串 (字符) 赋给新的 (n+1 个) 变量。例如，按‘.’分割字符，且分割 1 次，并将分割后的字符串赋给 2 个变量 str1 和 str2。

```
In [4]: url = ('www.pylab.club')
   ...: str1, str2 = url.split('.', 1)
   ...: print(str1)
        www

In [5]: print(str2)
        pylab.club
```

【例 1-7】提取下面字符串中的 50，0，51。

```
In [6]: str="xxxxxxxxxxxx5 [50,0,51]>,xxxxxxxxxx"
   ...: lst = str.split("[")[1].split("]")[0].split(",")
   ...: print(lst)
        ['50', '0', '51']
```

分解代码如下。

```
In [7]: list =str.split("[") #按照左边分割

In [8]: print(list)
        ['xxxxxxxxxxxx5 ', '50,0,51]>,xxxxxxxxxx']

In [9]: str.split("[")[1].split("]") #再对list的index=1的元素按"]"分割
Out[9]: ['50,0,51', '>,xxxxxxxxxx']

In [10]: str.split("[")[1].split("]")[0]#提取分割后的第一个元素，即index=0
Out[10]: '50,0,51'

In [11]: str.split("[")[1].split("]")[0].split(",")#再对提取后的按“,”分割
Out[11]: ['50', '0', '51']
```

1.7.3 divmod() 函数

divmod( ) 函数把除数和余数运算结合起来，返回一个包含商和余数的元组。

格式：divmod(a，b)　　#a 和 b 都是数字类型

返回值：(a//b，a%b)

```
In [1]: t = divmod(7,3)
   ...: t
Out[1]: (2, 1)
```

或者：

```
In [2]: quot,rem = divmod(7,3)
    ...: print(quot,rem)
        2 1
```

1.7.4 join() 函数

把一个序列 (list 或 tuple 等) 中所有的元素按照给定的分隔符 (sep) 连接起来，但限定连接的元素是字符型。

语法：'sep'.join(seq)

```
In [1]: a = ['a','b','c']
   ...: sep = '|'
   ...: "|".join(a) #用 “1” 符号将a中的元素连接起来
Out[1]: 'a|b|c'

In [2]: s=('1','2','3')
   ...: "|".join(s)
Out[2]: '1|2|3'

In [3]: c=[1,2,3]
   ...: '|'.join(c)
Traceback (most recent call last):
File "C:\Users\yubg\AppData\Local\Temp/ipykernel_16540/1020655982.py",
   line 2, in <module>
'|'.join(c)
        TypeError: sequence item 0: expected str instance, int found
```

由于列表 c 中的元素不是字符型，而是整型，所以运行结果出错，修改如下。

```
In [4]: c=['1','2','3']
   ...: '|'.join(c)
Out[4]: '1|2|3'
```

本章小结

本章知识点较多，重点是数据类型 list(列表)、tuple(元组)、dict(字典)、set(集合)。

1. 测试变量类型：type(object)。

2. 转换变量类型：str(object) # 将变量转化为字符串类型。

```
int(object)    #将变量转化为整型
```

3. 查询已安装的模块：help('modules')。

对于初学者而言，也许 dir() 和 help() 这两个函数是最有用的，使用 dir() 可以查看指定模块中所包含的所有成员或者指定对象类型所支持的操作，而 help() 则返回指定模块或函数的说明文档。例如，list 和 tuple 是否都有 pop() 和 sort() 方法呢？那就用 help() 查一下就很清楚了，下面列出了具体的用法，代码如下。

```
>>> help(list)
Help on class list in module builtins:
 |  ...
 |  append(...)
 |   L.append(object)-> None -- append object to end
 |  pop(...)
 |   L.pop([index])->item--remove and return item at index (default last).
```

```
 |      Raises IndexError if list is empty or index is out of range.
 |  sort(...)
 |      L.sort(key=None, reverse=False) -> None -- stable sort *IN PLACE*
...
>>> help(tuple)
Help on class tuple in module builtins:
...
 | count(...)
 |   T.count(value) -> integer -- return number of occurrences of value
 |
 | index(...)
 |   T.index(value, [start, [stop]])->integer--return first index of value.
 |      Raises ValueError if the value is not present.
>>>
```

4. 使用 dir( ) 查询相关参数的属性和方法，代码如下。

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__
init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__
subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
```

从上面列表中能看出 list 删除列表元素有两个方法 pop() 和 removedir()(pop 默认删除最后一个元素，remove() 删除首次出现的指定元素)。

5. 查询两个变量存储地址是否一致，使用 id() 函数即可。

6. 查询字符的 ASCII 码 (十进制的)，使用 ord() 函数，代码如下。

```
>>> ord('a')
97
>>>
```

反过来有了字符的十进制的 ASCII 码，用 chr() 函数找出对应的字符，代码如下。

```
>>> chr(97)
'a'
>>>
```

7. 查找字符串的长度：len()。

8. 在字符串 str 中通过索引能找出对应的元素，反过来，能否通过元素找出索引？代码如下。

```
>>>s='python good, you need it.'
>>>s[1]
'y'
>>>s.index('y')
1
>>>
```

s.index( ) 只能返回某个值第一个匹配项的索引。

9. tuple、list、string 的相同点。

元组，列表，字符串中的每一个元素都可以通过索引来读取，它们也都可以用 len( ) 方法测长度，都可以使用加法“+”和数乘“*”操作符。数乘表示将 tuple、list、string 重复数倍。

list 的 append()、insert()、pop()、del() 和 list[n] 赋值等方法属性均不能用于 tuple 和 string。

10. split 的逆运算：join

```
'sep'.join(list)
```

11. 列表和元组之间是可以相互转化的：list(tuple)、tuple(list)。

元组操作速度比列表快；列表可改变，元组不可变，可以将列表转化为元组，以保障列表中数据的安全。字典的 key 也要求不可变，所以元组可以作为字典的 key，但元素不能重复出现。

12. 字符串检测开头和结尾：string.endswith('str')、string.startswith('str')。示例如下。

```
>>> file = 'F:\\ data\\catering_dish_profit.xls'
>>> file.endswith('xls')       #判断file是否以xls结尾
True
>>>
>>> url = 'http://www.i-nuc.com'
>>> url.startswith('https') #判断url是否以https开头
False
>>>
14.S.replace(被查找词,替换词)  #查找与替换
>>> S='I love python, do you love python?'
>>> S.replace('python','R')
'I love R, do you love R?'
>>>
```

13. Python 的命名规范。

(1) 包名、模块名、局部变量名、函数名：全小写 + 下划线式驼峰。如：this_is_var。

(2) 全局变量：全部大写 + 下划线式驼峰。如：GLOBAL_VAR。

(3) 类名：首字母大写式驼峰。如：ClassName()。

(4) 下划线：

以单下划线开头，是弱内部使用标识，from M import * 时，将不会导入该对象。

以双下划线开头的变量名，主要用于类内部标识私有类型的变量，不能直接访问。

双下划线开头且双下划线结尾的命名方法一般不使用。


字符串 .mp4


list.wmv


range.wmv

练　　习

1. 已知：a = 2, b = 3，要求：将 a 和 b 的值调换，并输出结果。

2. 已知：a = 250, b ="250"，要求：阐述 a 和 b 所引用的对象的区别。

3. 计算：100 除以 3 得到的商、余数分别是多少？如果保留 3 位小数，结果是多少？

4. 请解释如下代码：

```
>>> round(2.675, 2)
2.67
>>>
```

5. 精确计算：2 除以 6，要求：以分数形式 (1/3) 输出最终结果。

6. 要求：在 print() 里面将“明月几时有”和“把酒问青天”两句分两行输入，但在同一行输出。

7. 编写程序，要求输入你的姓名和年龄，并且将年龄加 10 之后，与姓名一起输出。

8. 将字符串“map”的字符顺序倒转为“pam”。

9. 让用户输入一个单词，并显示这个单词的长度。

10. 已知字符串："Python is a widely used high-level, general-purpose, interpreted, dynamic programming language." 要求：将字符串中每个单词的第一个字母都变成大写字母，最终输出结果如下：

```
'Python Is A Widely Used High-level, General-purpose, Interpreted,
Dynamic Programming Language.'
```

11. 已知列表：["python", "java", "c", "c++", "lisp"]，要求：用切片方式将此列表中的第 1、3、5项取出来。

12. 生成一个由 100 以内能够被 5 整除的数组成的列表，然后将该列表的数字从大到小排序。

13. 已知两个列表：citys = ["suzhou", "shanghai", "hangzhou", "nanjing"]，codes = ["0512", "021", "0571", "025"]，要求：创建一个字典，以 citys 中的元素为 key，以 codes 中的元素为 value。

14. 已知：列表 lst1=[1,2,3,4,5,6], lst2=["a","b","c","d"]，要求：以 lst1 的元素为 key，以 lst2 的元素为 value 创建一个字典，并打印输出。

第 2 章 流程控制

任何复杂的计算机程序都是由顺序结构、选择 (if) 结构、循环 (for、while) 结构构成的。顺序结构就是按照代码逐行执行。下面主要介绍选择结构和循环结构。

2.1 选择结构

选择结构也称为判断结构，主要有 if-else、if-elif-else 等结构。

if-else 结构的特点是：当条件 condition 满足时，执行 if 下的语句块；当条件 condition 不满足时，执行 else 下的语句块。

if-else 语法结构如下：

```
if condition:
      block1
else:
      block2
```

if-else 结构说明如下。

- 在 if 和 else 行尾均要加冒号“:”。
- if 和 else 下的每条语句都要缩进，以显示逻辑层次关系，缩进四个空格或者一个 Tab 键，Tab 键和空格不能混用，以免程序出错。
- 每个语句尽可能各占一行。

if-else 的条件结构：

(1) 只有一个条件时，如果满足条件就执行 if 下的代码块。

```
In [1]: i = int(input('请输入数字: '))
   ...: if i>1:
   ...:     print('you have 200!')

请输入数字: 2
        you have 200!
```

(2) 当有两个条件时，可以使用 else，当条件满足时执行 if 下的语句块，条件不满足时执行 else 下的语句块，else 要与 if 对齐。

```
In [2]: i = int(input('请输入数字: '))
   ...: if i>1:
   ...:     print('%d大于1!'%i)
   ...: else:
   ...:     print('%d小于等于1.'%i)

请输入数字: 0
        0小于等于1.
```

(3) 如果多于两个条件，可以使用多个 elif。

```
In [3]: i = int(input('请输入数字: '))
   ...: if i>1:
   ...:     print('%d大于1!'%i)
   ...: elif i==1:
   ...:     print('%d等于1!'%i)
   ...: else:
   ...:     print('%d小于1! '%i)

请输入数字: 1
        1等于1!
```

2.2 循环结构

循环分遍历循环 (for) 和条件循环 (while) 两种。

2.2.1 for 循环

for 循环是指枚举一个序列 field 中的每个元素 var 时，都进行一次 block 处理，语法格式如下。

```
for var in field :
    block
```

(1) 常用于遍历列表，基本语法格式如下。

```
In [4]: for i in [2,3,4]:
   ...:     print(i,end=',')
        2,3,4,
```

上面代码的意思是将列表 [2,3,4] 中的每一个元素输出。在 print 语句中加上“end=',' ”表示将结果显示在一行中，并用逗号分隔开。

如果给定一个列表或元组，可以通过 for 循环来输出列表或元组中的每一个元素，这个过程称之为遍历，也称为迭代 (Iteration)。

(2)for 循环可以帮助处理字符串，假如想分别输出字符串中的每一个字母。

```
In [5]: for i in 'abc':
   ...:     print(i)
a
b
c
```

(3)for 经常和 range() 内置函数配合使用。

```
In [6]: for i in range(3):
   ...:     print(i)
0
1
2
```

【例 2-1】求 100 ～ 1000 的水仙花数。水仙花数是指一个 3 位数，它的每一位上的数字的 3 次幂之和等于它本身，如 $115^3=1^3+5^3+3^3$，称 153 为水仙花数。

```
In [7]: for i in range(100,1000):
   ...:     ge = i %10
   ...:     shi= i //10 %10
   ...:     bai= i // 100
   ...:     if ge**3+shi**3+bai**3 == i:
   ...:         print(i)
153
370
371
407
```

2.2.2 while 循环

while 循环用于在某种条件下循环地执行某段程序，以处理需要重复执行的相同任务，直到不满足循环条件时终止，其基本语法如下。

```
while <判断条件>:
    block
```

其中，block 可以是单个语句，也可以是语句块。判断条件可以是任何表达式，任何非零、非空 (null) 的值均为 True。当判断条件为 False 时，循环结束。

【例 2-2】在编辑器中输入代码并执行。

```
In [8]: i=0
   ...: while i<5:
   ...:     print('This is '+str(i)) #将i转化为字符型才能用“+”连接输出
   ...:     i+=1 #相当于i=i+1
```

运行上述代码，输出结果为如下。

```
This is 0
This is 1
This is 2
This is 3
This is 4
```

while 循环还有另外两个重要的语句 continue 和 break，用来跳过循环。此外“判断条件”还可以是个常量，表示循环必定成立，具体用法见下面 continue 语句和 break 语句。

2.2.3 continue 语句和 break 语句

在循环执行中，如果遇到 continue 语句，则跳过本轮循环，进入下一轮循环。如果有 10 次循环，运行到第 8 次时遇到了 continue 语句，则第 8 次循环中止，continue 语句后面的代码不再执行，继续开始第 9 次循环。

在循环执行中，只要遇到 break 语句，就停止循环，跳出整个 while 循环。

【例 2-3】continue 语句和 break 语句的用法。在编辑器中输入以下代码。

continue 语句的用法：

```
#1.将小于10的偶数输出
In [9]: i = 1
   ...: while i < 10:
   ...:     i += 1
   ...:     if i%2 > 0: #非偶数时跳过输出
   ...:         continue
   ...:     print(i) #输出偶数2、4、6、8、10
2
4
6
8
10
```

break 语句的用法：

```
In [10]: j = 1
    ...: while 1: #循环条件为1必定成立
    ...:     print(j)
    ...:     j += 1
    ...:     if j > 10: #当j大于10时跳出循环
    ...:         break
1
2
3
4
5
6
7
8
9
10
```

在使用 continue 语句编写循环语句时，要避免误入“死循环”，例如：

```
In [11]: i=1
    ...: while i<10:
    ...:     if i%2 == 0:
    ...:         continue
    ...:     print(i)
    ...:     i+= 1
        1
```

执行上述代码将输出小于 10 的奇数，但进入了死循环。因为当 i=2 时被整除，于是进入 if 下的 continue 语句行，其后面的 print(i) 和 i += 1 都不会被执行，但此时的 i 依然等于 2，所以继续进入 i=2，如此循环往复，一直都在输出 1，只有强行终止才能退出。上面代码稍作如下修改，即可运行：

```
In [12]: i=1
    ...: while i<10:
    ...:     if i%2 == 0:
```

```
    ...:          i+=1
    ...:          continue
    ...:      print(i)
    ...:      i+=1
1
3
5
7
9
```

2.3 常见的错误类型

Python 程序常见的错误类型如表 2-1 所示。

表 2-1 常见的错误类型

错误类型	描 述
NameError	尝试访问一个未声明的变量
ZeroDivisionError	除数为 0
SyntaxError	语法错误
IndexError	索引超出列表范围
KeyError	字典关键字不存在
IOError	输入、输出错误 (例如，要读的文件不存在)
AttributeError	访问未知的对象属性
IndentationError	冒号换行后没有缩进
ValueError	传给函数的参数类型不正确，比如给 int() 函数传入了字母

【例 2-4】一个常见的错误。

```
while 1:
    try:
    x = int(input('No.1: '))
    y = int(input('No.2: '))

    r= x/y
    print(r)
    except (Exception) as e:
        print(e)
        print('again')
    else:
        break

  File "<ipython-input-15-6f26b9b9c36e>", line 3
    x = int(input('No.1: '))
    ^
运行上述代码，将会出现如下的错误提示。
IndentationError: expected an indented block
```

例 2-4 的错误是因为 try 后有冒号，则下一行代码应该有缩进，后面的几行也都应该缩进。

下面对几种错误类型进行举例。

1. NameError：尝试访问一个未声明的变量

```
In [1]: print(v)
Traceback (most recent call last):

File "C:\Users\yubg\AppData\Local\Temp/ipykernel_18620/1609929102.py",
   line 1, in <module>
print(v)

NameError: name 'v' is not defined
```

2. ZeroDivisionError：除数为 0

```
In [2]: v = 1/0
Traceback (most recent call last):

File "C:\Users\yubg\AppData\Local\Temp/ipykernel_18620/2287803502.py",
   line 1, in <module>
v = 1/0

ZeroDivisionError: division by zero
```

3. SyntaxError：语法错误

```
In [3]: int 4
File "C:\Users\yubg\AppData\Local\Temp/ipykernel_18620/2342217697.py", line 1
int 4
^

SyntaxError: invalid syntax
```

4. IndexError：索引超出列表范围

```
In [4]: list1 = ['s','d']
   ...: list1[3]
Traceback (most recent call last):

File "C:\Users\yubg\AppData\Local\Temp/ipykernel_18620/2147726551.py",
   line 2, in <module>
list1[3]

IndexError: list index out of range
```

5. KeyError：字典关键字不存在

```
In [5]: Dic = {'1':'yes', '2':'no'}
   ...: Dic['3']
Traceback (most recent call last):

File "C:\Users\yubg\AppData\Local\Temp/ipykernel_18620/3917876188.py",
   line 2, in <module>
Dic['3']

KeyError: '3'
```

6. AttributeError：访问未知的对象属性

```
In [6]: str='123'
   ...: str.append()
Traceback (most recent call last):

File "C:\Users\yubg\AppData\Local\Temp/ipykernel_18620/16716481.py",
   line 2, in <module>
str.append()

AttributeError: 'str' object has no attribute 'append'
```

7. ValueError：数值错误

```
In [7]: int('d')
Traceback (most recent call last):

File "C:\Users\yubg\AppData\Local\Temp/ipykernel_18620/2890804116.py",
   line 1, in <module>
int('d')

ValueError: invalid literal for int() with base 10: 'd'
```

8. TypeError：类型错误

```
In [8]: print('123'+45)
Traceback (most recent call last):

File "C:\Users\yubg\AppData\Local\Temp/ipykernel_18620/2761622212.py",
   line 1, in <module>
print('123'+45)

TypeError: can only concatenate str (not "int") to str
```

初学 Python 时，想要彻底弄懂 Python 的错误信息的含义可能有点困难，下面列出了一些常见的程序运行时出现的异常，这些异常也是新手常犯的错误。

(1) 忘记在 if、elif、else、for、while、class、def 等末尾添加冒号 (导致 SyntaxError: invalid syntax 错误)。该错误发生在以下代码中：

```
if spam == 42
print('Hello!')
```

(2) 使用“ = ”而不是“ == ”(导致 SyntaxError: invalid syntax 错误)

“=”是赋值操作符，而“==”是判断相等的比较操作符。该错误发生在以下代码中：

```
if spam = 42:
    print('Hello!')
```

(3) 错误地使用缩进量 (导致 IndentationError: unexpected indent、IndentationError: unindent does not match any outer indetation level 以及 IndentationError: expected an indented block 错误)。

缩进只用在以“:”结束的语句之后，而之后必须恢复到之前的缩进格式，缩进是四个空格。该错误发生在以下代码中：

```
print('Hello!')
   print('Howdy!')
```

或者：

```
if spam == 42:
    print('Hello!')
 print('Howdy!')
```

或者：

```
if spam == 42:
print('Hello!')
```

缩进一定要按层次缩进，尽管 Python 3.x 的高版本已经做了很多优化，如“:”后换行有时可以只空一个空格，也能执行，但为了提高代码的可读性，应尽可能地空四格，以养成良好的代码编写习惯。

(4) 尝试修改 string 的值 (导致 TypeError: 'str' object does not support item assignment 错误)。

string 是一种不可变的数据类型，该错误发生在以下代码中：

```
spam = 'I have a pet cat.'
spam[13] = 'r'
```

正确的代码如下：

```
spam = 'I have a pet cat.'
spam = spam[:13] + 'r' + spam[14:]
```

(5) 尝试连接非字符串值与字符串。(导致 TypeError: Can't convert 'int' object to str implicitly 错误)。

该错误发生在以下代码中：

```
numEggs = 12
print('I have ' + numEggs + ' eggs.')
```

正确的代码如下：

```
numEggs = 12
print('I have ' + str(numEggs) + ' eggs.')
```

或者：

```
numEggs = 12
print('I have %s eggs.' % (numEggs))
```

(6) 在字符串首尾忘记加引号 (导致 SyntaxError: EOL while scanning string literal 错误)。
该错误发生在以下代码中：

```
print(Hello!')
```

或者：

```
print("Hello!)
```

或者：

```
myName = 'Al'
print('My name is ' + myName + . How are you?')
```

(7) 变量名或者函数名拼写错误 (导致 NameError: name 'fooba' is not defined 错误)。
该错误发生在以下代码中：

```
foobar = 'Al'
print('My name is ' + fooba)
```

或者：

```
spam = ruond(4.2)
```

或者：

```
spam = Round(4.2)
```

(8) 方法名拼写错误 (导致“AttributeError: 'str' object has no attribute 'lowerr' 错误)。
该错误发生在以下代码中：

```
spam = 'THIS IS IN LOWERCASE.'
spam = spam.lowerr()
```

(9) 引用超过列表最大范围 (导致 IndexError: list index out of range 错误)。
该错误发生在以下代码中：

```
spam = ['cat', 'dog', 'mouse']
print(spam[6])
```

(10) 使用字典中不存在的键名 (导致 KeyError: 'spam' 错误)。
该错误发生在以下代码中：

```
spam = {'cat': 'Zophie', 'dog': 'Basil', 'mouse': 'Whiskers'}
print('The name of my pet zebra is ' + spam['zebra'])
```

(11) 尝试使用 Python 关键字作为变量名 (导致 SyntaxError: invalid syntax 错误)。
Python 关键字不能用作变量名，该错误发生在以下代码中：

```
class = 'algebra'
```

Python 3.x 常见的关键字有：and、as、assert、break、class、continue、def、del、elif、else、except、False、finally、for、from、global、if、import、in、is、lambda、None、nonlocal、not、or、pass、raise、return、True、try、while、with、yield。

(12) 尝试使用 range() 创建整数列表 (导致 TypeError: 'range' object does not support item assignment 错误)。

如果要想得到一个有序的整数列表，那么使用 range() 函数好像是生成此列表的不错方式。然而，range() 函数返回的是“range object”，而不是列表类型。

该错误发生在以下代码中：

```
spam = range(10)
spam[4] = -1
```

正确的代码如下：

```
spam = list(range(10))
spam[4] = -1
```

(13) 在类中忘记为方法的第一个参数添加 self 参数 (导致 TypeError: myMethod() takes no arguments (1 given) 错误)。

该错误发生在以下代码中：

```
class Foo():
     def myMethod():
          print('Hello!')

a = Foo()
myMethod()
```

正确的代码如下：

```
class Foo():
     def myMethod(self):
          print('Hello!')

a = Foo()
a.myMethod()
```

2.4 异常处理

在 Python 中，执行代码时经常会遇到各种各样的异常情况，比如分母为 0 的除法错误，还有一些语法错误等。这些错误会导致程序无法运行，为了避免这种情况，并找到错误的根源，一般使用 try、assert、raise 等语句拦截和接收错误。

2.4.1 try语句

try-except 语句主要用于检测程序执行过程中可能会出现的一些异常情况，如语法错

误、数据除零错误、从未定义的变量上取值等；而 try-finally 语句则主要用于监控错误的环节。尽管 try-except 语句和 try-finally 语句的作用不同，但是在编程实践中通常可以把它们组合在一起，如以 try-except-else-finally 语句的形式来实现稳定性和灵活性更好的程序设计。

在 Python 中，try-except-else-finally 语句的完整格式如下：

```
try:
  block A
except x1:
  Exception A handle
except x2:
  Exception B handle
except:
  Other exception handle
else: #可无，若有，必有except x或except时存在，仅在try后无异常时执行
  block B
finally:  #此语句务必放在最后，并且也是必须执行的语句
  block C
```

说明：需要检测的程序放在 try 下的 block A 块中执行，在执行过程中如果发生了异常，则中断当前 block A 的执行，跳转到对应的异常处理块 except x1 或 except x2 中开始执行，Python 从 except x1 处开始查找，如果找到了对应的 exception 类型，则进入其提供的 Exception A handle 中进行处理，如果没有则依次进入 except x2，如果都没有找到，则直接进入 except 块进行处理。except 块是可选项，如果没有提供，该 exception 将会被提交给 Python 进行默认处理，处理方式则是终止应用程序并打印提示信息。

如果在 block A 执行过程中没有发生任何异常，则在执行完 block A 后会进入 else 下的 block B (若存在) 执行。

无论是否发生异常，若有 finally 语句，则以上 try-except-else 代码块的最后一步总是执行 finally 下的 block C。

执行 try 语句需要注意以下几项。

(1) 在上面所示的完整语句中，try-except-else-finally 所出现的顺序必须是 try-->except x-->except-->else-->finally，即所有的 except 必须在 else 和 finally 之前，else(若有) 必须在 finally 之前，而 except x(这里的 x 表示 x1、x2) 必须在 except 之前。否则就会出现语法错误。

(2) 对于前文所展示的 try-except 完整格式而言，else 语句和 finally 语句都是可选的，而不是必需的，finally(如果存在) 语句必须在整个语句的最后位置。

(3) 在前文的完整语句中，else 语句的存在必须以 except x 语句或者 except 语句为前提，如果没有 except 语句，单独使用 else 语句会引发语法错误。即 else 语句不能仅与 try-finally 语句配合使用。

【例 2-5】一个 try-except-else 的例子。

```
In [1]: while 1:
   ...:     try:
   ...:         x = int(input('No.1: '))
   ...:         y = int(input('No.2: '))
```

```
...:
...:         r= x/y
...:         print(r)
...:     except (Exception) as e: #不管什么异常，都捕获给e
...:         print(e)
...:         print('again')
...:     else:
...:         break
```

以上代码的意思是：从键盘接收两个输入，将其转化为 int 型数值做除法，如果符合除法规则，就输出结果，并跳转执行 else 语句；如果不符合除法规则，有错误，则进入 except 语句，打印错误原因，并重新来一次。输入和输出结果如下：

```
No.1: 1
No.2: 0
division by zero
again

No.1: 1
No.2: a
invalid literal for int() with base 10: 'a'
again

No.1: 1
No.2: 2
0.5
```

finally 放在 try 的最后，不管前面发生了什么，finally 后面的语句均执行。

2.4.2 assert语句

assert 是断言的关键字。执行该语句时，先判断表达式是否为真，如果表达式为真，则什么都不做；如果表达式为假，则抛出异常。例如：

```
In [2]: assert len('love') == len('like')

In [3]: assert len('love u') == len('like')
Traceback (most recent call last):

File "C:\Users\yubg\AppData\Local\Temp/ipykernel_18620/2999543666.py",
   line 1, in <module>
assert len('love u') == len('like')

AssertionError
```

可以看出，如果 assert 语句后面的表达式为真，则什么都不做；如果为假，就会抛出 AssertionError 异常。

2.4.3 raise语句

当程序出现错误，Python 会自动引发异常。也可以“人为的”通过 raise 显式地引发异常。一旦执行了 raise 语句，raise 后面的语句将不被执行。

```
In [1]: try:
   ...: s = None
   ...: if s is None:
    ...:          print("s 是空对象")
    ...:          raise NameError #如果NameError异常，后面的代码都将不执行
    ...:      print(len(s))
    ...: except TypeError:
    ...:      print("空对象没有长度")
s 是空对象
Traceback (most recent call last):

File "C:\Users\yubg\AppData\Local\Temp/ipykernel_18620/536627563.py",
   line 5, in <module>
raise NameError #如果NameError异常，后面代码都将不执行

NameError
```

本章小结

本章主要学习了 if、for、while 流程控制语句，尤其是使用 for 循环遍历的方法。重点内容有以下几点。

1. 使用 range() 函数产生序列。

2.for 和 while 循环的区别。

3. try 的使用方法。

try.wmv

练　　习

1. 在 0 ～ 9 之间随机选择 1 个整数，操作 100 次，统计共有几种数字，并用字典的方式输出每个数字的出现次数，键是随机选择整数，值是出现的次数。

2. 将整数 2016 的每个数字分离出来，依次打印输出。

3. 已知字典：{"name": "python", "book": "python", "lang":"english"}，要求：将该字典的键和值对换。(注意，字典中有键的值是重复的)

4. 已知：在一段程序中，用列表保存几个用户名，例如，['xiaoxifeng', 'cangcang', 'tom']，要求：通过终端输入新的用户名，判断输入的用户名是否为已经在列表中存在的用户名并且对判断结果给出友好的提示。如果不是，允许用户多次尝试输入，直到正确为止。

5. 找一段英文文本，统计该文本中单词的出现次数。比如：How are you. are you ok. 统计结果是：{"How":1, "are":2,"you":2,"ok":1}。

6. 已知：字符串 a = "aAsmr3idd4bgs7Dlsf9eAF"，要求：编写程序，完成如下任务：

(1) 请将字符串中的数字取出，并输出成一个新的字符串。

(2) 请统计字符串中每个字母出现的次数 (忽略大小写，a 与 A 是同一个字母)，并输出成一个字典。 例如，{'a':3,'b':1}。

(3) 请去除字符串中多次出现的字母，不区分大小写。如：'aAsmr3idd4bgs7Dlsf9eAF'，经过去除后，输出 'asmr3id4bg7lf9e'。

7. 有 100 个瓶子，分别从 1 一直标号到 100。现在有人拿枪从第一个开始射击，每枪击破一个跳过一个，一直到一轮完成。接着在剩下的瓶子里面再次击破第一个，间隔一个再击破一个，请问最后剩下完整的瓶子是这 100 个瓶子里的第几个？

8. 编写一段程序代码，能够实现如下功能。

(1) 输入一些英文的姓名。

(2) 按照首字母顺序将所有姓名排序。

(3) 输入完毕，将排序结果打印出来。

第 3 章 函数与类

Python 中函数的应用非常广泛，前文已经接触过多个函数，比如 input()、print()、range()、len() 等，这些都是 Python 的内置函数，可以直接使用。除了可以直接使用内置函数外，Python 还支持自定义函数，即将一段有规律的、可重复使用的代码定义成函数，从而达到一次编写多次使用的目的。

本章将介绍几个特殊的函数以及自定义函数和类。

3.1 常用函数

3.1.1 zip() 函数

zip(t,s) 是将 t 和 s 两个序列对应匹配构成一个二元元组的容器，若 t 和 s 长度不等，则其长度等于 t 和 s 中较短的一个。

```
In [1]: t0 = "abc"
   ...: t = list(t0)
   ...: t
Out[1]: ['a', 'b', 'c']

In [2]: s=[1,2,3]
   ...: z=zip(t,s) #zip()函数的结果是一个容器，需用list或tuple来调用
   ...: print(z)
        <zip object at 0x000001C77C500AC0>

In [3]: list(z)
Out[3]: [('a', 1), ('b', 2), ('c', 3)]
```

若长度不够，以最短的为主，例如：

```
In [4]: list(zip("abcd","123"))
Out[4]: [('a', '1'), ('b', '2'), ('c', '3')]
```

dict() 和 zip() 可以组合使用生成字典，例如：

```
In [5]: d= dict(zip('abc', range(1,4)))
   ...: d
Out[5]: {'a': 1, 'b': 2, 'c': 3}
```

3.1.2 enumerate() 函数

enumerate(t) 函数返回序列 t 的 index 和元素对，t 可以是字符串、列表、元组、字典等，若是字典返回的则是键名。

```
In [6]: t={'first':'j','second':'h','third':'abc'}
   ...: for i,k in enumerate(t):
```

```
   ...:     print(i,k)
0 first
1 second
2 third
```

3.1.3 eval() 函数

eval() 函数将字符串当成有效的表达式来求值并返回计算结果。

```
In [1]: x = 1
   ...: eval('x+1')
Out[1]: 2

In [2]: a = "[[1,2], [3,4], [5,6], [7,8], [9,0]]" #注意a是字符串
   ...: b = eval(a)
   ...: b
Out[2]: [[1, 2], [3, 4], [5, 6], [7, 8], [9, 0]]

In [3]: type(b)
Out[3]: list

In [4]: c = "{1: 'a', 2: 'b'}" #注意c是字符串
   ...: d = eval(c)
   ...: d
Out[4]: {1: 'a', 2: 'b'}

In [5]: type(d)
Out[5]: dict

In [6]: e = "([1,2], [3,4], [5,6], [7,8], (9,0))"
   ...: f = eval(e)
   ...: f
Out[6]: ([1, 2], [3, 4], [5, 6], [7, 8], (9, 0))
```

但下面的代码存在很大的风险：

```
>>> __import__('os').system('dir >dir.txt')
0
>>> open('dir.txt').read()
' 驱动器 D 中的卷是 soft\n 卷的序列号是 0046-527B\n\n D:\\python35
的目录\n\n2016-08-13  01:19    <DIR>          .\n2016-08-13  01:19
<DIR>          ..\n2016-06-21  22:51               406 0.py\n2016-
08-13  01:19                 0 dir.txt\n2016-06-15  22:07    <DIR>
DLLs\n2016-06-15  22:07    <DIR>          Doc\n2016-06-15  22:06
<DIR>          include\n2016-06-15  22:07    <DIR>          Lib\n2016-06-15
22:07    <DIR>          libs\n2015-12-07  19:56            30,338
LICENSE.txt\n2015-12-06  06:32           310,468 NEWS.txt\n2015-12-06
01:40            38,680 python.exe\n2015-12-06  01:39            51,480
python3.dll\n2015-12-06  01:39         3,122,968 python35.dll\n2015-12-06
01:40            38,680 pythonw.exe\n2015-11-22  22:58             8,269
README.txt\n2016-06-15  22:08    <DIR>          Scripts\n2016-06-15  22:08
<DIR>          tcl\n2016-06-25  11:37               317 test.py\n2016-06-
25  11:37               317 test1.py\n2016-06-25  18:59               150
TestClass.py\n2016-06-15  22:07    <DIR>          Tools\n2015-06-25  23:34
85,328 vcruntime140.dll\n2016-06-25  11:38    <DIR>          __pycache__\n
```

```
13 个文件       3,687,401 字节\n                11 个目录 35,201,863,680 可用字节\n'
    >>>
```

执行上面的两句代码，其实就已经在 Python 3.x 安装目录下建立了一个名为 "dir.txt" 的文件。若再运行下面这两行代码，则可以将新建的 dir.txt 文件删除。

```
>>> import os  #导入os模块
>>> os.system('del dir.txt /q')
0
>>>
```

上面新建的 dir.txt 文件已经被删除了，也就是说，这两行代码可以删除本台计算机上的任何文件。

下面的代码请自行测试：

```
>>>eval("__import__('os').system(r'md c:\\testtest')")
>>>eval("__import__('os').system(r'rd/s/q c:\\testtest')")
>>>eval("__import__('os').startfile(r'c:\windows\\notepad.exe')")
```

3.1.4 判断方法与函数

常用于判断的函数有如下几个。

1. in

in 常用于字符串、列表、元组、字典、集合中，用来判断一个字符串或者一个元素是否属于字符串或者列表等。同样，对应的还有 not in。

```
In [7]: a={'a':2,'b':4,'c':6}

In [8]: 'a' in a
Out[8]: True

In [9]: 'c' not in a
Out[9]: False
```

startswith()、endswith()：进行文本处理时用来判断字符串开始和结束的位置。

基本语法格式如下：

```
S.startswith(prefix[, start[, end]])
S.endswith(suffix[, start[, end]])
```

示例代码如下：

```
In [10]: "fish".startswith('fi')
Out[10]: True

In [11]: "fish".startswith('fi',1) #此处的1表示从索引位置1开始
Out[11]: False

In [12]: "fish".endswith('sh')
Out[12]: True

In [13]: "fish".endswith('sh',3)
```

```
Out[13]: False
```

Python 的这两个字符串方法有一个特别的地方——其参数 prefix 和 suffix 不仅可以是字符串，还可以是一个元组。只要其中一个成立，就返回 True，也就是一种“或”的关系。例如：

```
if filename.endswith(('.gif', '.jpg', '.tiff')):
   print("%s是一个图片文件" %filename)
```

上面两行代码根据文件扩展名是否为 gif、jpg 或 tiff 之一来决定文件是不是图片文件。这两行代码也可以写成如下形式：

```
if filename.endswith(".gif") or filename.endswith(".jpg") or \ #续行
filename.endswith(".tiff"):
   print("%s是一个图片文件"%filename)
```

不过，这样写并不简洁。值得注意的是，别忘了元组的括号。

2. isalnum() 函数

isalnum() 函数用来检测字符串是否仅由字母和数字组成，如果字符串中的字符仅包含字母和数字，则返回 True，若其间夹杂有空格、标点符号或者其他字符，则返回 False。

```
In [14]: str = "this2009"
    ...: print(str.isalnum())
         True

In [15]: str = "this is string example....wow!!!"
    ...: print(str.isalnum())
         False

In [16]: str1 = "hello"
    ...: print(str1.isalnum())
         True
```

3. isalpha() 函数

isalpha() 函数用来检测字符串是否只由字母组成。如果字符串中所有字符都是字母，则返回 True，否则返回 False。

```
In [17]: str = "this"
    ...: print(str.isalpha())
         True

In [18]: str = "this is string example....wow!!!"
    ...: print(str.isalpha())
         False
```

4. isdigit() 函数

isdigit() 函数用来检查字符串是否只包含数字 (全由数字组成)。如果字符串中所有字符都是数字，则返回 True，否则返回 False。

```
In [19]: str = "123456"
    ...: print(str.isdigit())
         True

In [20]: str = "this is string example....wow!!!"
    ...: print(str.isdigit())
         False
```

3.1.5 其他内置函数

其他内置函数如表 3-1 所示。

表 3-1 其他内置函数

函 数	描 述
abs(x)	求绝对值。 (1) 参数可以是整型，也可以是复数 (2) 若参数是复数，则返回复数的模
divmod(a, b)	分别取 a/b 的商和余数。 注意：整型、浮点型都可以
int([x[, base]])	将一个字符转换为 int 类型，base 表示进制
pow(x, y[, z])	返回 x 的 y 次幂
range([start], stop[, step])	产生一个序列，默认从 0 开始
round(x[, n])	四舍五入
sum(iterable[, start])	对集合求和
chr(i)	返回整数 i 对应的 ASCII 字符
bin(x)	将整数 x 转换为二进制字符串
format(value [, format_spec])	格式化输出字符串。 格式化的参数顺序从 0 开始，如“I am {0},I like {1}”
enumerate(sequence [, start = 0])	返回一个可枚举的对象 , 该对象的 next() 方法将返回一个元组
max(iterable[, args...][key])	返回集合中的最大值
min(iterable[, args...][key])	返回集合中的最小值
dict([arg])	创建数据字典
list([iterable])	将一个集合类转换为另外一个集合类
set()	set 对象实例化
str([object])	转换为 string 类型
sorted(iterable[,cmp[,key[, reverse]]])	对集合列表排序，不对原集合列表排序，返回新的集合列表
tuple([iterable])	生成一个元组类型
eval(expression [, globals [, locals]])	计算表达式 expression 的值
id(object)	返回对象的唯一标识
len(s)	返回集合长度
locals()	返回当前的变量列表
map(function, iterable, ...)	遍历每个元素，执行 function 操作
next(iterator[, default])	类似于 iterator.next()

续表

函　数	描　述
reduce(function,iterable[, initializer])	合并操作，从第一个开始是前两个参数，然后是前两个的结果与第三个合并进行处理，以此类推
type(object)	返回该 object 的类型
zip([iterable, ...])	返回一个以元组为元素的列表
input([prompt])	获取用户输入，返回的结果是字符型
open(name[,mode[, buffering]])	打开文件
print()	打印函数，可以带参数 end

3.2　自定义函数

在编写程序代码时，有些功能经常会重复出现，实现这些功能的代码就可以拿出来进行单独编写，并给它一个名称，在需要的时候直接使用它的名称和几个参数进行调用即可，免去了重复写代码的工作，并使得代码简单易读。这就是自定义函数。

3.2.1　自定义函数语法

在 Python 中，自定义函数的基本形式如下：

```
def function(params):
    """
    函数说明文档，用于help的调用，此处可以省略。
    """
    block
    return expression/value
```

说明：

(1) 在 Python 中采用 def 关键字进行函数的定义，不用指定返回值的类型，另外注意 def 行尾的“:”；函数的命名一般首字母不需要大写，以与类名区分。

(2) 参数 params 可以没有、一个或者多个，同样，参数也不用指定类型，因为在 Python 中变量都是弱类型，Python 会自动根据值来维护其类型。

(3)return 语句是可选的，它可以在函数体内任何地方出现，表示函数调用执行到此结束；如果没有 return 语句，会自动返回 None，如果有 return 语句，但 return 后面没有表达式或者函数值，也会返回 None。返回的值也就是输出的功能结果。return 可以返回多个值，以逗号分隔，如 return a,b,c，相当于返回一个 return (a,b,c)。

请注意，函数体 block 内部的语句在执行时，一旦执行到 return 时，函数就执行完毕，并将结果返回。因此，函数内部通过条件判断和循环可以实现非常复杂的逻辑。如果没有 return 语句，函数执行完毕后也会返回结果，只是结果为 None。return None 可以简写为 return。

(4) 一般在函数中还包含一个注释体，即函数文档，其功能是解释这个函数的功用，

用三引号引起来放在 block 最前面，也是为了方便能够用 help() 函数查询，也可以省略。

【例 3-1】自定义函数。

```
In [21]: def printHello():
    ...:     print('hello')
    ...:
    ...: def readNum():
    ...:     """
    ...:     利用range函数输出0,1,2,3,4
    ...:     """
    ...:     for i in range(0,5):
    ...:         print(i)
    ...:     return
    ...:
    ...: def add(a,b):
    ...:     return a+b

In [22]: print(printHello())
         hello
None

In [23]: print(readNum())
0
1
2
3
4
None

In [24]: print(add(1,2))
         3
```

3.2.2 调用自定义函数

在 Python 中，函数的使用有严格的规定，函数不允许向前引用，即函数必须定义在前，使用在后。例如：

```
In [25]: print(add2(1,2))
    ...: def add2(a,b):
    ...:     return a+b
Traceback (most recent call last):

File "C:\Users\yubg\AppData\Local\Temp/ipykernel_15092/1286047753.py",
   line 1, in <module>
print(add2(1,2))

NameError: name 'add2' is not defined
```

从报错中可以看出，命名为 "add2" 的函数未进行定义。因此，在调用某个函数时，必须确保此函数定义在调用之前，即先定义再调用函数。上述程序可修改为：

```
In [26]: def add2(a,b):
    ...:     return a+b
    ...: print(add2(1,2))
         3
```

3.2.3 形参和实参

在使用 def 定义函数时，函数名后面括号里的变量称作形式参数 (简称形参)。在调用函数时提供的值或者变量称作实际参数 (简称实参)。

【例 3-2】形参和实参。

```
In [27]: def add(a,b): #这里的a和b是形参
    ...:     return a+b

In [28]: add(1,2) #这里的1和2是实参
Out[28]: 3

In [29]: x=2
    ...: y=3
    ...: add(x,y) #这里的x和y是实参
Out[29]: 5
```

注意，定义 add(a,b) 函数里面的参数 a 和 b 要注意位置，当我们把实参 x、y 放进函数中时，会按照先后顺序，如 add(x,y) 是将 x 赋值给 a，y 赋值给 b；而 add(y,x) 则是将 x 赋值给 b，y 赋值给 a，即严格按照位置顺序进行赋值。

3.2.4 参数传递

在大多数高级语言中，对参数的传递方式的理解一直是难点和重点，因为它理解起来并不是那么直观明了。

在讨论此问题之前，需要明确的是，在 Python 中一切皆对象，包括之前用到的字符串常量、整型常量等都是对象，变量中存放的是对象的引用。验证如下：

```
In [30]: print(id(5))
         1531243030960

In [31]: print(id('python'))
         1531288900144

In [32]: x=2

In [33]: print(id(x))
         1531243030864

In [34]: y='hello'

In [35]: print(id(y))
         1531621149488
```

先解释一下函数 id() 的作用。id(object) 函数返回对象 object 在其生命周期内位于内

存中的地址，id 函数的参数类型是一个对象，因此对于语句 id(2) 没有报错，就可以知道 2 在这里是一个对象。例如：

```
In [36]: id(2)
Out[36]: 1531243030864

In [37]: id('hello')
Out[37]:1531621149488
```

从结果可以看出，id(x) 和 id(2) 的值是一样的，id(y) 和 id('hello') 的值是一样的。

在 Python 中一切皆对象，像 2、'hello' 这样的值都是对象，只不过 2 是一个整型对象，而 'hello' 是一个字符串对象。上面的 x=2，在 Python 中实际的处理过程是这样的：先申请一段内存分配给一个整型对象来存储整型值 2，然后让变量 x 去指向这个对象，实际上就是指向这段内存。而 id(2) 和 id(x) 的结果一样，说明 id 函数在作用于变量时，其返回的是变量指向的对象的地址。因为变量也是对象，所以在这里可以将 x 看作对象 2 的一个引用。

下面再看一个例子。

```
In [38]: x=2
    ...: print(id(x))
         1531243030864

In [39]: y=2
    ...: print(id(y))
         1531243030864

In [40]: s='hello'
    ...: print(id(s))
         1531621149488

In [41]: t=s
    ...: print(id(t))
         1531621149488
```

从运行结果可以看出 id(x) 和 id(y) 的结果是相同的，id(s) 和 id(t) 的结果是相同的。这说明变量 x 和变量 y 指向的是同一个对象，而变量 t 和变量 s 指向的也是同一个对象。x=2 让变量 x 指向了 int 类型的对象 2，而 y=2 执行时，并不重新为 2 分配空间，而是让变量 y 直接指向了已经存在的 int 类型的对象 2。这个很好理解，因为本身只是想给变量 y 赋一个值 2，而在内存中已经存在这样一个 int 类型的对象 2，所以就直接让变量 y 指向了已经存在的对象。这样一来不仅能达到目的，而且能节约内存空间。t=s 变量互相赋值，也相当于让变量 t 指向了已经存在的字符串类型的对象 'hello'。

下面就来讨论一下函数的参数传递和改变这个问题。

在 Python 中参数传递采用的是值传递。下面先看一个例子：

```
In [42]: def modify1(m,K):
    ...:     m=2
    ...:     K=[4,5,6]
```

```
    ...:     return
    ...:
    ...: def modify2(m,K):
    ...:     m=2
    ...:     K[0]=0
    ...:     return

In [43]: n=100
    ...: L=[1,2,3]
    ...: modify1(n,L)

In [44]: print(n,L)
         100 [1, 2, 3]

In [45]: modify2(n,L)
    ...: print(n,L)
         100 [0, 2, 3]
```

从结果可以看出，执行 modify1() 函数之后再打印 n 和 L，发现 n 和 L 都没有发生任何改变；执行 modify2() 函数后，n 还是没有改变，L 发生了改变。因为在 Python 中参数传递采用的是值传递方式，在执行函数 modify1() 时，先获取 n 和 L 的 id() 值，然后为形参 m 和 K 分配空间，让 m 和 K 分别指向对象 100 和对象 [1,2,3]。m=2 让 m 重新指向对象 2，而 K=[4,5,6] 让 K 重新指向对象 [4,5,6]。这种改变并不会影响到实参 n 和 L，所以在执行 modify1() 之后，n 和 L 没有发生任何改变；同理，在执行函数 modify2 时，让 m 和 K 分别指向对象 2 和对象 [1,2,3]，然而 K[0]=0 让 K[0] 重新指向了对象 0(注意这里 K 和 L 指向的是同一段内存)，所以对 K 指向的内存数据进行的任何改变也会影响到 L，因此在执行 modify2() 后，L 发生了改变。

3.2.5 变量的作用域

在 Python 中，也存在作用域的问题，会为每个层次生成一个符号表，里层能调用外层中的变量，而外层不能调用里层中的变量，并且当外层和里层有同名变量时，外层变量会被里层变量屏蔽掉。

【例 3-3】不同作用域中的变量。

```
In [46]: def function():
    ...:     x=2
    ...:     count=2
    ...:     while count>0:
    ...:         x=3
    ...:         print(x)
    ...:         count -= 1
    ...:
    ...: function()
3
3
```

在函数 function 中，while 循环外部和内部都有变量 x，此时 while 循环外部的变量 x 会被屏蔽掉。注意在函数内部定义的变量作用域都仅限于函数内部，在函数外部是不

能够被调用的，一般称这种变量为局部变量。

还有一种变量称为全局变量，它是在函数外部定义的，其作用域是整个程序。全局变量可以直接在函数内部调用，但是如果要在函数内部改变全局变量的值，必须使用 global 关键字进行声明。

【例 3-4】全局变量。

```
In [47]: x=2
    ...: def fun1():
    ...:     print(x)
    ...:
    ...: def fun2():
    ...:     global x #global语句用于声明一个或多个全局变量
    ...:     x=3      #在函数内部改变全局变量的值
    ...:     print(x)

In [48]: fun1()
         2

In [49]: fun2()
         3

In [50]: print(x)
         3
```

函数 def 定义的变量只能在 def 内部被使用，不能在函数外部被使用。一个在 def 外部被赋值的变量 x 与一个在 def 内部被赋值的变量 x 是完全不同的两个变量。Python 变量可以分为本地变量 (def 内部，除非用 global 声明)、全局变量 (模块内部)、内置变量 (预定义的 __builtin__ 模块)。全局声明 global 会将变量名映射到模块文件内部的作用域。变量名的引用将依次查找本地变量、全局变量、内置变量。例如：

```
In [51]: x = 99
    ...: def add(y):
    ...:     z = x + y
    ...:     return z
    ...:
    ...: print(add(1))
         100
```

从结果可以看出，add(1) 在运行时，其内部的 x 用的是函数 add() 外部的 x 的值。

global 语句用于声明一个或多个全局变量。例如：

```
In [52]: x = 88
    ...: def func():
    ...:     global x
    ...:     x = 99

In [53]: func()

In [54]: x
Out[54]: 99
```

执行 func() 之后，x 的值变成了 99，说明在函数内部经过 global 对 x 的声明，改变

x 的值会影响到函数外部 x 的值。

再例如：

```
In [55]: y,z = 1,2
    ...:
    ...: def func():
    ...:     global x
    ...:     x = y + z

In [56]: func()
    ...: print(x,y,z)
         3 1 2
```

3.2.6 函数参数的类型

之前我们接触到的函数的参数称为位置参数，即参数是通过位置进行匹配的，从左到右依次进行，对参数的位置和个数都有严格的对应要求。而在 Python 中还有一种是通过参数的名称来匹配的，这种匹配方式不需要严格按照参数定义时的位置来传递，这种参数称为关键字参数。

```
In [57]: def display(a,b):
    ...:     print(a)
    ...:     print(b)

In [58]: display('hello','world')
hello
world
```

上面这段程序是想输出 'hello world'，可以正常运行。如果是下面这段代码，可能就得不到预期的结果：

```
In [59]: display('hello')            #这样会报错，少了一个参数
Traceback (most recent call last):

File "C:\Users\yubg\AppData\Local\Temp/ipykernel_15092/4085202510.py",
   line 1, in <module>
display('hello') #这样会报错

TypeError: display() missing 1 required positional argument: 'b'


In [60]: display('world','hello')     #这样会输出'world hello'
world
hello
```

可以看出，在 Python 中默认是采用位置参数来匹配，所以在调用函数时必须严格按照函数定义时参数的个数和位置来传递，否则将会出现预想不到的结果。下面这段代码采用关键字参数进行传递：

```
In [61]: display(a='world',b='hello')
world
```

```
hello

In [62]: display(b='hello',a='world')
world
hello
```

上面两段代码的效果是一样的。从上面程序的输出结果可知，通过指定参数名称传递参数时，参数位置对结果没有影响。另外，关键字参数最优越的地方在于它能够给函数参数提供默认值。例如：

```
In [63]: def display(a='hello',b='world'):
    ...:     print(a+b)

In [64]: display()
         helloworld

In [65]: display(b='world')
         helloworld

In [66]: display(a='hello')
         helloworld

In [67]: display('world')
         worldworld
```

在上面的代码中，分别给 a 和 b 指定了默认参数，即如果不给 a 或 b 传递参数时，它们就分别采用默认值。在给参数指定了默认值后，如果传递参数时不指定参数名，则会从左到右依次进行参数传递，例如，display('world') 没有指定 'world' 是传递给 a 还是 b，则默认从左向右匹配，即传递给 a。另外，默认参数一般靠右。

使用默认参数固然方便，但在重复调用函数时，默认形参会继承前一次调用结束之后该形参的值。例如：

```
In [68]: def insert(a,L=[]):
    ...:     L.append(a)
    ...:     print(L)

In [69]: insert('hello')
         ['hello']

In [70]: insert('world') #形参L继承了前一次调用结束之后的值['hello']
         ['hello', 'world']
```

3.2.7 任意个数参数

一般情况下，在定义函数时，函数参数的个数是确定的，然而在某些情况下，参数的个数是不确定的。例如，某系统要存储用户的姓名及其小名，有些用户小名可能有两个或者更多个，此时无法确定参数的个数，就可以使用任意多个参数 (收集参数)。使用收集参数只需在参数前面加上 '*' 或者 '**'。

```
In [71]: def storename(name,*nickName):
```

```
    ...:     print('real name is %s' %name)
    ...:     for nickname in nickName:
    ...:         print(nickname)

In [72]: storename('jack')
real name is jack

In [73]: storename('詹姆斯','小皇帝')
real name is 詹姆斯
小皇帝

In [74]: storename('奥尼尔','大鲨鱼','三不沾')
real name is 奥尼尔
大鲨鱼
三不沾
```

'*' 和 '**' 表示能够接受 0 到任意多个参数，'*' 表示将没有匹配的值都放在同一个元组中，'**' 表示将没有匹配的值都放在一个 dictionary 中。

```
In [75]: def printvalue(a,*s,**d):
    ...:     print(a,s,d)

In [76]: printvalue(1,2,c=3)
         1 (2,) {'c': 3}

In [77]: printvalue(1,3,4,2,c=3,f="l")
         1 (3, 4, 2) {'c': 3, 'f': 'l'}
```

需要补充一点：在 Python 中，函数是可以返回多个值的，这样会将多个值放在一个元组或者其他类型的集合中返回。例如：

```
In [78]: def function():
    ...:     x=2
    ...:     y=[3,4]
    ...:     return x,y

In [79]: function()
         (2, [3, 4])
```

3.2.8 函数调用

对于已经编辑好的函数代码，保存成 .py 文件后，Python 就可以调用其内的所有函数，方案如下。

(1) 将 a.py 文件和正在编辑的文件 (该文件将要调用 a.py 中的函数) 放在同一个目录下。

(2) 在调用文件头引入：from a import *。

这样就可以使用 a.py 文件内所有的函数和变量了。

【例 3-5】在 prin.py 文件中调用文件 tel.py 中的变量。

文件 tel.py 的代码如下：

```
#文件tel.py的代码内容
name=["Ben","Jone","Jhon","Jerry","Anny","Ivy","Jan","Wong"]
tel=[6601,6602,6603,6604,6605,6606,6607,6608]

Tellbook={}
for i in range(len(name)):
    d1="{}".format(name[i])
    d2="{}".format(tel[i])
    Tellbook[d1]=d2
```

正在编辑的文件 prin.py的代码如下：

```
In [1]: from tel import *          #从tel.py文件中导入所有的函数变量

In [2]: print(Tellbook)
{'Ben': '6601', 'Jone': '6602', 'Jhon': '6603', 'Jerry': '6604',
   'Anny': '6605', 'Ivy': '6606', 'Jan': '6607', 'Wong': '6608'}

In [3]: for i,j in zip(name,tel):
   ...: print(i,": ",j)
Ben :  6601
Jone :  6602
Jhon :  6603
Jerry :  6604
Anny :  6605
Ivy :  6606
Jan :  6607
Wong :  6608
```

上面的 prin.py 文件要做两件事情，先把 tel.py 文件中的所有变量导入到 prin.py 中并将 Tellbook 打印输出，再将 tel.py 文件中的 name 和 tel 用 zip 合并成一组相对应的序列，并将序列中的每个元素打印出来。

对于文件中的函数调用也一样，当我们调用函数时也需要使用 import 来导入。

【例 3-6】函数的导入调用。

yu.py 文件的代码如下：

```
#yu.py文件的代码内容
def add(a=0,b=0):
    '''
    此函数是计算两个数的和
    当不输入参数时,默认的是0+0
    '''
    c=a+b
    print(c)
def gb(m,K=0,*tup,**dic):
    print('m:',m)
    print('K:',K)
    print('tup:',tup)
    print('dic:',dic)
    return
```

在下面的 test.py 文件中调用 yu.py 文件中的 add(a,b) 函数。

test.py 文件代码如下：

```
In [4]: from yu import add
   ...: a=add(1,2)
        3
```

这里的 from yu import add 的意思是从 yu.py 文件中导入 add(a,b) 函数。当然，如果导入 yu.py 文件中所有的函数，则可以用 from yu import *，为了避免导入所有的函数占内存，所以一般使用什么函数就导入什么函数，除非导入的比较多，才使用 *。另外，为了防止导入的多个模块中有相同的函数名而引起混乱，也不建议使用 *。

有时候，我们用 import yu 方式导入，这时候使用 yu.py 中的 add() 函数时，则需要说明是来自哪里的 add() 函数，即 yu.add()，说明调用的是 yu.py 文件中的 add() 函数。

```
In [5]: import yu
   ...: yu.add(1,2)
         3
```

使用 from yu import add 和 import yu 的区别如下。

当使用 from yu import add 时，表示当前要编辑的代码中要使用的 add() 函数都来自 yu.py 文件中的 add()，而不是系统内置的或者其他的，在调用时，直接使用 add(1,2)；而使用 import yu 时，表示已经导入了 yu.py 文件，至于用其中的某个函数时，则在使用的函数前加“yu.”即可，如使用其中的 add() 函数，则为 yu.add(1,2)。关于 import 导入模块的问题在 3.4.1 节中详细介绍。

说明：写函数时要养成良好的习惯——写函数文档，写清楚此函数的功能是什么、怎么用，必要时给出示例，并把文档内容用三引号注释起来，它不是函数体的执行代码，它的作用是给 help 提供查询，如查询上例 yu.py 文件的功能：

```
In [6]: help(add)
Help on function add in module yu:

add(a=0, b=0)
此函数是计算两个数的和
当不输入参数时,默认的是0+0
```

还是上例，输入如下代码，并观察结果：

```
In [7]: from yu import gb
   ...: gb('a1')
m: a1
K: 0
tup: ()
dic: {}

In [8]: gb(1,3)
m: 1
K: 3
tup: ()
dic: {}

In [9]: gb('a1',K=2)
m: a1
K: 2
```

```
tup: ()
dic: {}

In [10]: gb('a1',2,3,6,fname='yu',name='bg')
m: a1
K: 2
tup: (3, 6)
dic: {'fname': 'yu', 'name': 'bg'}

Out[11]: gb('a1',K=2,3,6,fname='yu',name='bg')     #注意参数顺序
File "C:\Users\yubg\AppData\Local\Temp/ipykernel_10488/3189193839.py",
   line 1
gb('a1',K=2,3,6,fname='yu',name='bg')
^
         SyntaxError: positional argument follows keyword argument

In [12]: gb('a1',K=2,fname='yu',name='bg')
m: a1
K: 2
tup: ()
dic: {'fname': 'yu', 'name': 'bg'}

In [13]: gb(K=2,fname='yu',name='bg')              #该函数的第一个参数不能少
Traceback (most recent call last):

File "C:\Users\yubg\AppData\Local\Temp/ipykernel_10488/1917465081.py",
   line 1, in <module>
gb(K=2,fname='yu',name='bg')

         TypeError: gb() missing 1 required positional argument: 'm'
```

请比较一下出错的原因。

关于函数名含义的一个补充：函数名其实就是指向一个函数对象的引用，完全可以把函数名赋给一个变量，相当于给这个函数起了一个“别名”，例如：

```
In [14]: a = int       #变量a指向int函数

In [15]: a('2')         #可以通过a调用int函数
Out[15]: 2
```

3.3 特殊函数

3.3.1 lambda() 函数

lambda() 为匿名函数，也叫行内函数或临时函数。

先来看看下面定义的两个函数。

```
In [1]: f = lambda x : x**2+1          #定义了一个函数f(x) = x**2+1
   ...: g = lambda x,y : x+y           #定义了一个函数g(x,y) = x+y
```

```
In [2]: f(3)
Out[2]: 10

In [3]: g(1,2)
Out[3]: 3
```

lambda 只是一个表达式，函数体比 def 简单很多。lambda 表达式运行起来像一个函数，其用途为如下两点。

(1) 对于单行函数，使用 lambda 可以省去定义函数的过程，让代码更加精简。

(2) 在非多次调用函数的情况下，lambda 表达式即用即得，提高性能。

注意：如果 for..in..if 能做的，最好不要选择 lambda。

3.3.2 filter() 函数

filter() 函数用于过滤序列。

filter() 接收一个函数和一个序列，并把传入的函数依次作用于序列的每个元素，然后根据返回值 True 或者 False，来决定保留还是丢弃该元素。

例如，在一个列表（list）中，删掉偶数，只保留奇数，代码如下：

```
In [1]: def is_odd(n):
   ...:     return n % 2 == 1
   ...: list(filter(is_odd, [1, 2, 4, 5, 6, 9, 10, 15]))
Out[1]: [1, 5, 9, 15]
```

再如，把一个序列中的空字符串删掉，代码如下：

```
In [2]: def not_empty(s):
   ...:     return s and s.strip()
   ...: list(filter(not_empty, ['A', '', 'B', None, 'C', ' ']))
Out[2]: ['A', 'B', 'C']
```

可见，使用 filter() 这个高阶函数时，关键在于正确地实现一个“筛选”函数。

filter() 是一个容器，返回时需要用 list 或 tuple 调用才显示数据。

3.3.3 map() 函数

map(func,S) 将传入的函数 func() 依次作用到序列 S 的每个元素，并把结果作为新的序列返回。函数 func() 在 S 域上遍历，map() 是一个容器，返回时需要用 list 或 tuple 调用才显示数据，显示的是 func() 函数作用后的结果数据。

【例 3-7】比较 map() 函数和 filter() 函数。

```
In [1]: list(map(lambda x:x**2,[1,2,3]))
Out[1]: [1, 4, 9]

In [2]: list(filter(lambda x:x**2,[1,2,3]))
Out[2]: [1, 2, 3]
```

说明：map() 函数返回的是 func() 函数作用后的结果数据，而 filter() 函数是通过 func() 函数筛选出作用域的数据。map() 函数还可以接受多个参数的函数。

```
In [3]: list(map(lambda x,y:x*y+x,[1,2,3],[4,5,6]))
          #x取自于[1,2,3]，y取自于[4,5,6]
Out[3]: [5, 12, 21]
```

3.3.4 行函数

行函数，也叫列表解析式或列表推导式，语法格式如下：

```
[ <expr1> for k in L if <expr2> ]     #<expr1>和<expr2>均表示一个表达式。
```

【例 3-8】将列表 [1,2,3,6] 中能被 2 整除的数提取出来并加上 2。

```
In [1]: lis=[1,2,3,6]
   ...: A=[k+2 for k in lis if k % 2 == 0 ]
   ...: print(A)
        [4, 8]
```

可用一行代码实现：

```
In [2]: [k+2 for k in [1,2,3,6] if k % 2 == 0 ]
Out[2]: [4,8]
```

列表推导式 (list comprehension) 是利用其他序列创建新列表的一种方法。其工作方式类似于 for 循环。

```
In [3]: [x*x for x in range(10)]
Out[3]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

for 与 if 连用表示按条件来生成列表。如果只想打印出能被 3 整除的数的平方，只需要通过添加一个 if 部分，在推导式中就可以完成：

```
In [4]: [x*x for x in range(10) if x % 3 == 0]
Out[4]: [0, 9, 36, 81]
```

也可以增加更多的 for 语句：

```
In [5]: [(x,y) for x in range(3) for y in range(3)]
Out[5]: [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2,
           1), (2, 2)]

In [6]: [[x,y] for x in range(2) for y in range(2)]
Out[6]: [[0, 0], [0, 1], [1, 0], [1, 1]]
```

列表推导式可以利用 range() 函数生成一个倒序列表。

```
In [7]: [i for i in range(5, 0, -1)] #步长为负数表示倒序
Out[7]: [5, 4, 3, 2, 1]

In [8]: list(range(5, 0, -2))     #步长为-2
Out[8]: [5, 3, 1]
```

range() 函数也可以倒序，格式为 range(a,b,-1)，a 要大于 b。

filter() 函数也可以实现行函数功能，例如：

```
In [9]: b = [ i for i in range(1,10) if i>5 and i<8 ]
   ...: b
```

```
Out[9]: [6, 7]
```

用 filter() 函数改写如下：

```
In [10]: list(filter(lambda x: x>5 and x<8, range(1,10)))
Out[10]: [6, 7]
```

3.4 模块和包

下面，我们先来看一个例子：

```
In [1]: a=[1.23e+18, 1, -1.23e+18]
   ...: sum(a)      #把a中所有元素加和
Out[1]: 0.0
```

怎么会是 0 ？再执行下面的代码：

```
In [2]: import math
   ...: math.fsum(a)
Out[2]: 1.09
```

计算机由于浮点数的运算问题，会导致上面代码的运算结果有差异。但是，在引入一个 math 模块后，计算结果就正常了。

3.4.1 模块

模块 (module)是包含函数和其他语句的 Python 脚本文件，它以“.py”为后缀名，即 Python 脚本的后缀名。表现形式为编写的代码保存为文件，这个文件就是一个模块，如前文的 yu.py 文件，其中文件名“yu”为模块名称。

在 Python 中可以导入模块，然后使用其模块中提供的函数或者变量。模块的导入方法以 math 模块为例：

(1)import math　　# 导入 math 模块

(2)import math as m　　# 导入 math 模块并取别名为“m”

(3)from math import exp as e　# 导入 math 库中 exp 函数并取别名为“e”

要想使用“import 模块名”模式导入模块中的函数，则必须以“模块名 . 函数名”的形式调用函数，或者使用“import 模块名 as 别名”模式以“别名 . 函数名”的形式调用函数；而 from 是将模块中某个函数导入，所以使用 from 导入模块中的某个函数，可以直接使用函数名调用，不必在前面加上模块名称。如上例引入 math 模块可以通过以下方式调用：

```
In [3]: import math as m          #给math模块取个别名m，使用时用m替代math
   ...: a=[1.23e+18, 1, -1.23e+18]
   ...: m.fsum(a)
Out[3]: 1.0

In [4]: from math import fsum     #这里直接导入了math模块中的fsum函数
```

```
   ...: a=[1.23e+18, 1, -1.23e+18]
   ...: fsum(a)                    #直接使用fsum()，不再使用math.fsum()
Out[4]: 1.0
```

使用 from 导入模块中的函数后，再使用模块中的函数会方便很多，不再使用模块名。如果要想将多个模块中的所有函数都采用这种方式导入，则可以在 from 中使用“*”通配符，表示导入模块中的所有函数，但不建议这样使用。其代码如下：

```
In [5]: from math import sqrt #仅导入了sqrt函数
   ...: sqrt(4)
Out[5]: 2.0

In [6]: cos(4)                #仅导入了sqrt函数，所以cos函数不能直接用
Traceback (most recent call last):
File "C:\Users\yubg\AppData\Local\Temp/ipykernel_8284/63647921.py",
  line 1, in <module>
cos(4)
        NameError: name 'cos' is not defined

In [7]: from math import *    #将math中的所有函数全部导入
   ...: sqrt(4)
Out[7]: 2.0

In [8]: cos(4)                #上面已经将math中的所有函数都导入了，可以直接使用
Out[8]: -0.6536436208636119
```

模块就是一个扩展名为 .py 的程序文件。我们可以直接调用它，这样可以节省时间和精力，无须重复写同样的代码。Python 的开源，也说明了“他山之石可以攻玉，他为我用”的思想。调用模块时最好将被调用文件和调用文件置于同一个文件夹下，也可以用临时访问文件的方法，如当前文件需要调用 E:/yubg/python 中的 ybg.py 文件：

```
import sys
sys.path.append('E:/yubg/python')
import ybg
```

3.4.2 包

Python 包 (package) 是一个有层次的文件目录结构，它定义了由 n 个模块或 n 个子包组成的 Python 应用程序执行环境。简单来说，包是一个包含 __init__.py 文件的目录，该目录下一定得有 __init__.py 文件和其他模块或子包，也就是带有 __init__.py 的文件夹，并不在乎里面有什么。

多个关系密切的模块组成一个包，以便于维护和使用。这项技术能有效地避免名称空间冲突。创建一个名为包名的文件夹，并在该文件夹下创建一个 __init__.py 文件就定义了一个包。可以根据需要在该文件夹下存放资源文件、已编译扩展及子包。举例来说，一个包可能有以下结构：

```
yubg/
   __init__.py
   index.py
   Primitive/
```

```
            __init__.py
            lines.py
            fill.py
            text.py
            ...
        yubg_1/
            __init__.py
            plot2d.py
            ...
        yubg2/
            __init__.py
            plot3d.py
            ...
```

import 语句使用以下几种方式导入包中的模块：

```
import yubg.Primitive.fill
#导入模块yubg.Primitive.fill，只能以全名访问模块属性
#例如yubg.Primitive.fill.floodfill(img,x,y,color)

from yubg.Primitive import fill
#导入模块fill，只能以 fill.属性名这种方式访问模块属性
#例如 fill.floodfill(img,x,y,color)

from yubg.Primitive.fill import floodfill
#导入fill，并将函数floodfill放入当前名称空间，直接访问被导入的属性
#例如floodfill(img,x,y,color)
```

无论一个包的哪个部分被导入，在 __init__.py 文件中的代码都会运行。这个文件的内容允许为空，不过通常情况下用它来存放包的初始化代码。导入过程遇到的所有 __init__.py 文件都被运行。因此，import yubg.Primitive.fill 语句会按顺序运行 yubg 和 Primitive 文件夹下的 __init__.py 文件。

下面的语句有歧义：

```
from yubg.Primitive import *
```

本语句的原意是想将 yubg.Primitive 包下的所有模块导入当前的名称空间。然而，由于不同平台之间文件命名规则不同 (如大小写、敏感问题等)，Python 不能正确判断哪些模块要被导入。语句只会按顺序运行 yubg 和 Primitive 文件夹下的 __init__.py 文件。要解决这个问题，应在 Primitive 文件夹下的 __init__.py 文件中，定义一个名为 all 的列表，例如：

```
# yubg/Primitive/__init__.py
__all__ = ["lines","text","fill",...]
```

这样，上面的语句就可以导入列表中所有的模块。

3.4.3 time 模块、datetime 模块和 calendar 模块

在 Python 中，与处理时间有关的模块包括：time、datetime、calendar 等。

一些术语和约定的解释如下。

时间戳 (timestamp)：通常来说，时间戳表示的是从 1970 年 1 月 1 日 00:00:00 开始按秒计算的偏移量 (time.gmtime(0))，此模块中的函数无法处理 1970 年以前的日期和时间或太遥远的未来。

协调世界时 (Coordinated Universal Time，UTC)：也称格林尼治标准时间，是世界标准时间。在中国为 UTC+8。

1. time 模块

DST(Daylight Saving Time)：即夏令时的意思。

time 模块提供各种与时间相关的功能。

【例 3-9】time 模块的各种用法。

```
In [1]: import time     #导入时间模块
   ...: t1=time.time() #返回现在的时间，但返回的是时间戳
   ...: t1
Out[1]: 1658327039.0026648

In [2]: t2=time.ctime() #返回现在的时间，正常的时间格式
   ...: t2
Out[2]: 'Wed Jul 20 22:24:03 2022'

In [3]: t3=time.ctime(t1) #可以将时间戳作为参数，返回正常时间格式
   ...: t3
Out[3]: 'Wed Jul 20 22:23:59 2022'

In [4]: t4=time.localtime() #返回当前的时间元组格式，具体见说明
   ...: t4
Out[4]: time.struct_time(tm_year=2022, tm_mon=7, tm_mday=20, tm_hour=22,
            tm_min=24, tm_sec=10, tm_wday=2, tm_yday=201, tm_isdst=0)

In [5]: t5=time.asctime() #返回现在的时间，正常的时间格式
   ...: t5
Out[5]: 'Wed Jul 20 22:24:13 2022'

In [6]: t6=time.asctime(t4) #可将时间元组作为参数，返回正常时间格式
   ...: t6
Out[6]: 'Wed Jul 20 22:24:10 2022'

In [7]: time.strftime('%y/%m/%d') #返回当前日期，以“/”分隔，也可换成以“,”分隔
Out[7]: '22/07/20'
```

说明：

```
tm_year   年
tm_mon    月
tm_mday   日
tm_hour   时
tm_min    分
tm_sec    秒
tm_wday   一周中的第几天
tm_yday   一年中的第几天
tm_isdst  夏令时(-1代表夏令时)
```

2. datetime 模块

datetime 模块重新封装了 time 模块，提供了更多接口，提供的类有：date、time、datetime、timedelta(时间加减) 等。

```
In [8]: import datetime
   ...: datetime.date.today() #返回当前日期
Out[8]: datetime.date(2022, 7, 20)

In [9]: datetime.date(2016, 4, 10)
Out[9]: datetime.date(2016, 4, 10)

In [10]: datetime.date.today().ctime() #返回当前日期时间
Out[10]: 'Wed Jul 20 00:00:00 2022'

In [11]: datetime.date.today().timetuple() #返回当前的时间元组格式
Out[11]: time.struct_time(tm_year=2022, tm_mon=7, tm_mday=20, tm_hour=0,
             tm_min=0, tm_sec=0, tm_wday=2, tm_yday=201, tm_isdst=-1)

In [12]: print(datetime.datetime.now()) #返回当前正常的日期时间
         2022-07-20 22:28:41.956324

In [13]: t = datetime.datetime.now() #取当前日期时间

In [14]: m = t + datetime.timedelta(5) #在t时刻上增加5天(默认是天)

In [15]: m
Out[15]: datetime.datetime(2022, 7, 25, 22, 28, 43, 396117)

In [16]: n = t + datetime.timedelta(weeks=5)
    ...: print(n)
         2022-08-24 22:28:43.396117
```

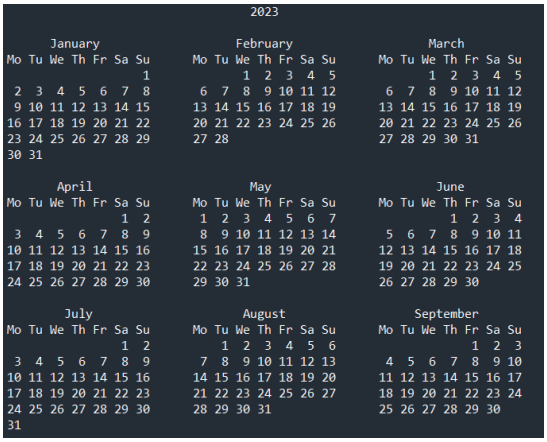
说明：timedelta() 的参数还可以是 hours=5，或者是 weeks=5、minutes=5、seconds=5，但没有 years 和 months，因为年和月的时间不确定，如有的月份天数是 30 天或 31 天。

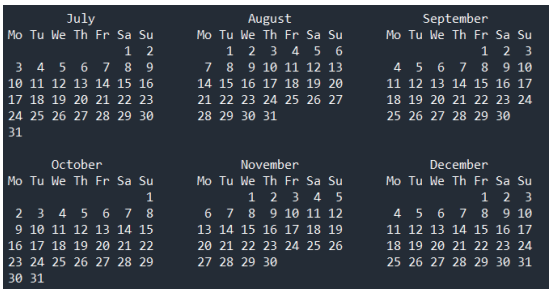
3. calendar 模块

calendar 模块的函数都与日历有关，例如，打印某月的字符月历。星期一默认为每周的第一天，星期天默认为每周的最后一天。更改设置需调用 calendar.setfirstweekday() 函数。

```
In [17]: import calendar
    ...: m = calendar.month(2016,4)#返回某年某月的日历
    ...: print(m)
April 2016
Mo Tu We Th Fr Sa Su
1  2  3
4   5  6  7   8  9  10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30

In [18]: n = calendar.calendar(2023,w=2,l=1,c=6) #见注
    ...: print(n)
```





注：calendar.calendar(2023,w=2,l=1,c=6) 返回某年的年历，三个月一行，间隔距离为 c，每日宽度间隔字符为 w，每行字符长度为 21*w+18+2*c，l 是每星期行间距。

calendar 模块还可以处理闰年的问题。判断是否闰年、两个年份之间闰年的个数。

```
In [19]: import calendar
    ...: print(calendar.isleap(2012))
         True

In [20]: print(calendar.leapdays(2010, 2015))
         1
```

3.4.4 urllib 模块

当下载网上的文档、数据时，也就是大家常说的爬虫，经常要用到 urllib 模块，具体代码如下：

```
In [21]: import urllib.request
    ...: ur = urllib.request.urlopen("http://www.baidu.com")
    ...: content = ur.read()
    ...: mystr = content.decode("utf8")
    ...: ur.close()
    ...: print(mystr)
<!DOCTYPE html><!--STATUS OK-->

<html><head><meta http-equiv="Content-Type" content="text/
    html;charset=utf-8"><meta http-equiv="X-UA-Compatible" content
    ="IE=edge,chrome=1"><meta content="always" name="referrer"><meta
    name="theme-color" content="#ffffff"><meta name="description"
    content="全球领先的中文搜索引擎、致力于让网民更便捷地获取信息，找到所求。百度
    超过千亿的中文网页数据库，可以瞬间找到相关的搜索结果。"><link rel="shortcut
    icon" href="/favicon.ico" type="image/x-icon" /><link rel="search"
    type="application/opensearchdescription+xml" href="/content-search.
    xml" title="百度搜索" /><link rel="icon" sizes="any" mask href="//
    www.baidu.com/img
…
<script src="http://ss.bdimg.com/static/superman/js/components/
```

```
    hotsearch-b24aa44c42.js"></script>
<script defer src="//hectorstatic.baidu.com/cd37ed75a9387c5b.js">
    </script>
</body>
</html>
```

由于网页内容太多，这里只截取了部分内容。当然，如果想要获取更多的内容，就要使用更多的技术。这里仅仅把首页上的内容“抓”了下来。需要注意以下代码。

```
urllib.request.urlopen("http://www.baidu.com")
```

这行代码里的参数是网址，“http://”不能少，否则会报错，关于更多、更深层次的爬虫技术后文再做介绍。

3.5 类

面向对象编程 (object oriented programming，OOP) 是一种程序设计思想。OOP 把对象作为程序的基本单元，一个对象是包含了数据和操作数据的函数。

在 Python 中，所有数据类型都可以视为对象，当然也可以自定义对象。自定义的对象数据类型就是面向对象中的类 (class) 的概念。

类是所有面向对象语言中最难理解的一项内容。Python 中的类是一个抽象的概念，比函数还要抽象，这也是 Python 的核心概念，是一个非常重要的知识点，我们可以把它简单地看作由数据以及存取、操作这些数据的方法所组成的一个集合。我们在学习函数之后，知道了函数可以重复使用代码，那为什么还要用类来取代函数呢？类有以下几项优点。

(1) 类对象是多态的：也就是多种形态，这意味着我们可以对不同的类对象使用同样的操作方法，而不需要额外写代码。

(2) 类的封装：封装之后，可以直接调用类的对象，来操作类内部的一些方法，不需要让使用者看到代码工作的细节。

(3) 类的继承：类可以从其他类或者元类中继承它们的方法，直接使用。

定义类的语法格式如下：

```
class NameClass(object):
    def fname(self, name):
        self.name = name
```

第一行语法是类后面紧接类的名称，最后带上“：”。类的名称首字母最好是大写。

第二行开始是类的方法，和函数非常相似，但是与普通函数不同的是，它的内部有一个“self”参数，它的作用是对对象自身的引用。

举一个例子来说明面向过程和面向对象在程序流程上的不同之处。假设要处理学生的成绩表，为了表示一个学生的成绩，面向过程的程序可以用一个字典表示：

```
In [1]: std1 = { 'name': 'Michael', 'score': 98 }
   ...: std2 = { 'name': 'Bob', 'score': 81 }
```

而处理学生成绩可以通过函数实现，例如，打印学生的成绩：

```
In [2]: def print_score(std):
   ...:     print('{0},{1}'.format(std['name'], std['score']))

In [3]: print_score(std1)
        Michael,98
```

如果采用面向对象的程序设计思想，我们首先思考的不是程序的执行流程，而是 Student 这种数据类型应该被视为一个对象，这个对象拥有 name 和 score 这两个属性 (property)。如果要打印一个学生的成绩，首先必须创建出这个学生对应的对象，然后给对象发一个 print_score 消息，让对象自行把数据打印出来。

```
In [4]: class Student(object):
   ...:     def __init__(self, name, score):
   ...:         self.name = name
   ...:         self.score = score
   ...:     def print_score(self):
   ...:         print('{0},{1}'.format(self.name,self.score))
```

给对象发消息实际上就是调用对象对应的关联函数，我们称之为对象的方法 (method)。面向对象的程序写出来如下所示：

```
In [5]: bart = Student('Bart Simpson', 59)
   ...: lisa = Student('Lisa Simpson', 87)
   ...: bart.print_score()
        Bart Simpson,59

In [6]: lisa.print_score()
        Lisa Simpson,87
```

面向对象的设计思想源于自然界，因为在自然界中，类和实例 (instance) 的概念是很自然的。类是一个抽象概念，例如，我们定义的 class Student，是指学生这个概念，而实例 (instance) 则是一个个具体的 Student，例如，Bart Simpson 和 Lisa Simpson 是两个具体的 Student。

所以，面向对象的设计思想是抽象出类，再根据类创建实例 (instance)。

类的方法与普通的函数只有一个特别的区别——它们必须有一个额外的第一个参数，但是在调用此方法的时候，不用为这个参数赋值，Python 会提供这个值。这个特别的变量指对象本身，它就是 self。

Python 如何给 self 赋值以及为何不需要给它赋值？举一个例子，假如有一个 MyClass 类和它的一个实例 MyObj。当调用这个对象的方法 MyObj.method(arg1,arg2) 的时候，会由 Python 自动转化为 MyClass.method(MyObj, arg1, arg2)——这就是 self 的赋值原理。

Python 中类的概念可以比作某种类型集合的描述，打个比方：类就是烤饼干的模子，而一个个的饼干就是一个个实例，或者说类就是一个工厂，实例就是一个个产品。

创建类时，可以定义一个特定的方法，名为 __init__()，只要创建这个类的一个实例就会运行这个方法。可以向 __init__() 方法传递参数，这样创建对象时就可以把属性设置为我们希望的值，__init__() 这个方法会在创建对象时完成初始化。代码如下：

```
In [7]: class Peo:
   ...:     def __init__(self,name,age,sex):
   ...:         self.Name = name
   ...:         self.Age = age
   ...:         self.Sex = sex
   ...: def speak(self):
   ...:     print("my name: %s" %self.Name)
```

实例化这个类的对象时，代码如下：

```
In [8]: zhangsan=Peo("zhangsan",24,'man')
   ...: print(zhangsan.Age)
        24

In [9]: print(zhangsan.Name)
        zhangsan

In [10]: print(zhangsan.Sex)
         man

In [11]: zhangsan.speak()
         my name: zhangsan
```

本章小结

本章主要学习了自定义函数和面向对象的类，尤其是自定义函数带有默认参数的传参方法。重点内容有以下几点。

1. 常用的内置函数。
2. 自定义函数 def。
3. 模块的导入与使用。
4. 类的实现。


函数 .wmv

练　　习

1. 编写一个函数，实现摄氏温度和华氏温度之间的换算，换算公式：F=9C/5 + 32，F 表示华氏温度，C 表示摄氏温度。要求输入单位是摄氏度的值，能够显示相应的华氏度的值，反之亦然。

2. 制作一个加法计算器，要求用户先后输入两个数字，能够计算出结果，并打印出加法算式。

3. 为老师们编写一个处理全班考试成绩的程序，要求如下。

(1) 能够依次录入班级同学的姓名和分数。

(2) 录入完毕，则打印出全班同学的平均分，以及最高分同学的姓名和分数。

4. 编写工资额计算器，要求如下。

(1) 确定每月的基本工资。

(2) 输入每月的实际工作天数。

(3) 输入当月的请假天数，如果请假天数小于等于 2 天，对工资无影响；大于 2 天小于等于 7 天，扣除当月基本工资的 10%；大于 7 天小于等于 14 天，扣除当月基本工资的 50%；大于 14 天，扣除全月工资。

(4) 如果当月实际工作天数和应该工作天数一样 (不算加班)，则增加基本工资的 20%。

(5) 如果当月有加班，则按照加班的天数和当月的日工资 (基本工资 / 实际工作天数) 计算加班费。

(6) 输出最终应得的工资。

5. 有多少个三位数字能被 17 整除？编写程序，将这些数字显示出来。

6. 编写一个猜数游戏，要求如下。

(1) 用户可以输入无限多次数字。

(2) 如果猜中了数字，则要输出用户猜测的次数和数字结果。

7. 编写程序，判断一个数字是否为素数。

8. 创建类 PayCalculator，拥有属性 pay_rate，以人民币数量 / 小时为单位。拥有方法 compute_pay(hours)，计算给定工作时间的报酬，并返回。

9. 创建类 SchoolKid，初始化小孩的姓名、年龄。要有访问每个属性的方法和修改属性的方法。然后创建类 ExaggeratingKid，继承类 SchoolKid，子类中覆盖访问年龄的方法，并将实际年龄加 2。

第 4章 数据处理

数据分析的首要工作就是数据处理。曾有人说，数据分析百分之七八十的工作都是在做数据清洗，由此可见，数据处理是多么重要的一步。

4.1 Numpy

Python 用列表 (list)保存一组值，可以当作数组使用，不过列表的元素可以是任何对象，因此，列表中所保存的是对象的指针。为了保存一个简单的列表 [1,2,3]，需要有 3 个指针和 3 个整数对象。对于数值运算来说，这种结构显然浪费内存和 CPU 计算时间。

Python 提供了一个 array 模块。array 对象和列表不同，它直接保存数值，但是它不支持多维数组对象，也没有各种运算函数，因此不适合做数值运算。

Numpy 库的诞生弥补了这些不足。Numpy 库是 Python 中科学计算的基础软件包。它可以提供多维数组对象、多种派生对象 (如掩码数组、矩阵) 以及用于快速操作数组的函数及 API，包括数学、逻辑、数组形状变换、排序、选择、I/O、离散傅立叶变换、基本线性代数、基本统计运算、随机模拟等。

Numpy 库是 Python 中的一个线性代数库。对每一个数据科学或机器学习的 Python 库而言，它都是一个非常重要的库，SciPy(Scientific Python)、Matplotlib(Plotting Library)、Scikit-learn 等在一定程度上都依赖 Numpy 库。

在导入 Numpy 库时，我们通过 as 将 Numpy 的别名记作 np，导入方式如下：

```
import numpy as np
```

4.1.1 数组的创建

先从 Python 列表中创建 Numpy 数组。

```
In [1]: import numpy as np
   ...: my_list = [1, 2, 3, 4, 5]
   ...: my_numpy_list = np.array(my_list)
```

通过这个列表，我们已经简单地创建了一个名为 my_numpy_list 的 Numpy 数组，显示结果如下：

```
In [2]: my_numpy_list
Out[2]: array([1, 2, 3, 4, 5])
```

我们已将一个列表转换成一维数组。要想得到二维数组，则需要创建一个包含列表为元素的列表，代码如下：

```
In [3]: second_list = [[1,2,3], [5,4,1], [3,6,7]]
   ...: new_2d_arr = np.array(second_list)
```

```
   ...: new_2d_arr
Out[3]:
          array([[1, 2, 3],
                 [5, 4, 1],
                 [3, 6, 7]])
```

已经成功创建了一个 3 行 3 列的二维数组。有时为了方便数据操作，我们需要将数组转化为列表，使用 tolist() 函数即可。

```
In [4]: c = np.array([[1, 2, 3, 4],[4, 5, 6, 7], [7, 8, 9, 10]])
   ...: c
Out[4]:
           array([[ 1, 2, 3, 4],
                  [ 4, 5, 6, 7],
                  [ 7, 8, 9, 10]])

In [5]: c.tolist()
Out[5]: [[1, 2, 3, 4], [4, 5, 6, 7], [7, 8, 9, 10]]
```

我们还可以通过给 array 函数传递 Python 的序列对象来创建数组，如果传递的是多层嵌套的序列，将创建多维数组，如下面的变量 c。

```
In [4]: a = np.array([1, 2, 3, 4])
   ...: b = np.array((5, 6, 7, 8))
   ...: c = np.array([[1, 2, 3, 4],[4, 5, 6, 7], [7, 8, 9, 10]])
   ...: b
Out[4]: array([5, 6, 7, 8])

In [5]: c
Out[5]:
          array([[ 1, 2, 3, 4],
                 [ 4, 5, 6, 7],
                 [ 7, 8, 9, 10]])

In [6]: c.dtype              #查看c的数据类型
Out[6]: dtype('int32')
```

数组的大小可以通过其 shape 属性获得。

```
In [7]: a.shape                    #查看a的数组维度
Out[7]: (4,)

In [8]: c.shape
Out[8]: (3, 4)
```

数组 a 的 shape 只有一个元素，因此它是一维数组。而数组 c 的 shape 有两个元素，因此它是二维数组，其中第 0 轴的长度为 3，第 1 轴的长度为 4，如图 4-1 所示。

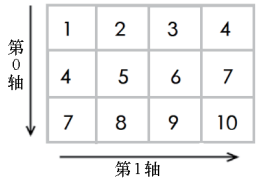


图 4-1　二维数组轴

还可以通过修改数组的 shape 属性，在保持数组元素个数不变的情况下，改变数组每个轴的长度。下面的例子将数组 c 的 shape 改为 (4,3)，注意从 (3,4) 改为 (4,3) 并不是对数组进行转置，而是改变每个轴的大小，数组元素在内存中的位置并没有改变。

```
In [9]: c.shape = 4,3
```

```
   ...: c
Out[9]:
          array([[ 1, 2, 3],
                 [ 4, 4, 5],
                 [ 6, 7, 7],
                 [ 8, 9, 10]])
```

当某个轴的长度为 –1 时，相当于占位符，这个 –1 位置上将根据数组元素的个数自动计算此轴的长度，因此下面的代码将数组 c 的 shape 改为了 (2,6)，但这里的 6 不需要人工去计算，以 –1 替代，由计算机自动计算填充。

```
In [10]: c.shape = 2,-1
    ...: c
Out[10]:
           array([[ 1, 2, 3, 4, 4, 5],
                  [ 6, 7, 7, 8, 9, 10]])
```

使用数组的 reshape 方法，可以创建一个改变尺寸的新数组，原数组的 shape 保持不变。代码如下：

```
In [11]: d = a.reshape((2,2))
    ...: d
Out[11]:
           array([[1, 2],
                  [3, 4]])

In [12]: a
Out[12]: array([1, 2, 3, 4])
```

使用 reshape 方法新生成的数组和原数组共用一个内存，不管改变哪个数组都会互相影响。其实数组 a 和数组 d 是共享数据存储内存区域的，因此，修改其中任意一个数组的元素都会同时修改另外一个数组的内容。代码如下：

```
In [13]: a[1] = 100   #将数组a索引为1的元素改为100
    ...: d            #注意数组d中的2也被改变了
Out[13]:
           array([[ 1, 100],
                  [ 3, 4]])
```

数组的元素类型可以通过 dtype 属性获得。上面例子中的参数序列的元素都是整数，因此，所创建的数组的元素类型也是整数，并且是 32 位的长整型。可以通过 dtype 参数在创建时指定元素类型。

```
In [14]: np.array([[1,2,3,4],[4,5,6,7], [7,8,9,10]], dtype=np.float)
Out[14]:
           array([[ 1., 2., 3., 4.],
                  [ 4., 5., 6., 7.],
                  [ 7., 8., 9., 10.]])

In [15]: np.array([[1,2,3,4],[4,5,6,7], [7,8,9,10]], dtype=np.complex)
Out[15]:
           array([[ 1.+0.j, 2.+0.j, 3.+0.j, 4.+0.j],
                   [ 4.+0.j, 5.+0.j, 6.+0.j, 7.+0.j],
                   [ 7.+0.j, 8.+0.j, 9.+0.j, 10.+0.j]])
```

当想了解一个数组包含多少个数据时，可以使用 size 来查阅。代码如下：

```
In [16]: d=np.array([[ 1, 100],[ 3, 4]])
    ...: d.size
Out[16]: 4

In [31]: len(d)
Out[31]: 2
```

注意：len 和 size 的区别，len 是指元素的个数，而 size 是指数据的个数，也就是说一个元素可以包含多个数据。

上面的例子都是先创建一个 Python 序列，然后通过 array() 函数将其转换为数组，这样做显然效率不高。因此，Numpy 库提供了很多专门用来创建数组的函数。下面的每个函数都有一些关键字参数，具体用法请查看函数说明。

在第 1 章中我们学习过 range() 函数。该函数通过指定的开始值、终止值和步长生成一个整数序列，但如果要生成一个小数序列呢？这就要用到 Numpy 库中的 arange() 函数。arange() 函数类似于 Python 的内置函数 range()。使用 arange() 函数需要先导入 Numpy 库。例如，产生一个 0 ～ 1 的步长为 0.1 的序列，其代码如下：

```
In [16]: np.arange(0,1,0.1)
Out[16]: array([ 0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

linspace() 函数通过指定开始值、终止值和元素个数来创建一维数组，且数组中的元素构成等差数列，可以通过 endpoint 关键字指定是否包括终止值。默认设置为包括终止值，这一点要特别注意。其代码如下：

```
In [17]: np.linspace(0, 1, 12)
Out[17]:
         array([ 0. , 0.09090909, 0.18181818, 0.27272727, 0.36363636,
                0.45454545, 0.54545455, 0.63636364, 0.72727273, 0.81818182,
                0.90909091, 1. ])
```

logspace() 函数和 linspace() 函数类似，通过它可以创建等比数列，下面的代码产生 1(10**0) ～ 100(10**2) 中 20 个元素的等比数列，其代码如下：

```
In [18]: np.logspace(0, 2, 20)
Out[18]:
           array([ 1. , 1.27427499, 1.62377674, 2.06913808,
                  2.6366509 , 3.35981829, 4.2813324 , 5.45559478,
                  6.95192796, 8.8586679 , 11.28837892, 14.38449888,
                  18.32980711, 23.35721469, 29.76351442, 37.92690191,
                  48.32930239, 61.58482111, 78.47599704, 100. ])
```

还可以通过 zeros() 函数和 ones() 函数等来创建多维数组，其代码如下：

```
In [19]: import numpy as np
    ...: my_zeros = np.zeros(5)

In [20]: my_zeros
Out[20]: array([ 0., 0., 0., 0., 0.])

In [21]: my_ones = np.ones(5)
```

```
In [22]: my_ones
Out[22]: array([ 1., 1., 1., 1., 1.])

In [23]: two_zeros = np.zeros((3,5))
    ...: two_zeros
Out[23]:
           array([[ 0., 0., 0., 0., 0.],
                  [ 0., 0., 0., 0., 0.],
                  [ 0., 0., 0., 0., 0.]])

In [24]: two_ones = np.ones((5,3))
    ...: two_ones
Out[24]:
           array([[ 1., 1., 1.],
                  [ 1., 1., 1.],
                  [ 1., 1., 1.],
                  [ 1., 1., 1.],
                  [ 1., 1., 1.]])
```

创建一个一维数组，并且把元素 3 重复 4 次，可以使用 repeat() 函数，代码如下：

```
In [25]: np.repeat(3, 4)
Out[25]: array([3, 3, 3, 3])
```

还可以使用 np.full(shape, val) 函数创建多维数组，每个元素值均填充为 val，代码如下：

```
In [26]: np.full((2,3),8)
Out[26]:
           array([[8, 8, 8],
                  [8, 8, 8]])
```

在处理线性代数时，单位矩阵是非常有用的。单位矩阵是一个二维的方阵，即在这个矩阵中列数与行数相等，它的对角线都是 1，其他均为 0。单位矩阵可以使用 eye() 函数来创建，代码如下：

```
In [27]: my_matrx = np.eye(6)

In [28]: my_matrx
Out[28]:
           array([[ 1., 0., 0., 0., 0., 0.],
                  [ 0., 1., 0., 0., 0., 0.],
                  [ 0., 0., 1., 0., 0., 0.],
                  [ 0., 0., 0., 1., 0., 0.],
                  [ 0., 0., 0., 0., 1., 0.],
                  [ 0., 0., 0., 0., 0., 1.]])
```

在处理数据时，有时会用到随机数组成的数组，这时可以使用 rand()、randn() 或 randint() 函数生成。

(1) np.random.rand() 可以生成一个从 0 ～ 1 均匀产生的随机数组成的数组。例如，如果想生成一个由 4 个对象组成的一维数组，且这 4 个对象均匀分布在 0 ～ 1，代码如下。

```
In [1]: import numpy as np
   ...: my_rand = np.random.rand(4)
   ...: my_rand
Out[1]: array([ 0.8038377 , 0.82393353, 0.07511963, 0.28900456])
```

如果想要创建一个 5 行 4 列的随机二维数组，代码如下：

```
In [2]: my_rand = np.random.rand(5, 4)
   ...: my_rand
Out[2]:
          array([[ 0.23075524, 0.37075683, 0.02791661, 0.59149501],
                 [ 0.19525257, 0.20225569, 0.03901862, 0.32141019],
                 [ 0.59996611, 0.95734781, 0.15140956, 0.43600606],
                 [ 0.42776634, 0.8688988 , 0.75872595, 0.36019754],
                 [ 0.88073936, 0.51553821, 0.44954604, 0.93475329]])
```

(2) np.random.randn() 可以从以 0 为中心的标准正态分布或高斯分布中产生随机样本。生成 7 个随机数代码如下：

```
In [3]: my_randn = np.random.randn(7)
   ...: my_randn
Out[3]:
          array([-0.69841501, -1.18251376, -0.26387785, -0.1519803,
          -1.12398459,-1.01932536, -0.09537881])
```

根据数据绘制后得到一个正态分布曲线。

同样地，如果想要创建一个 3 行 5 列的二维数组，代码如下：

```
In [4]: np.random.randn(3,5)
Out[4]:
   array([[-0.66033972, -0.82280485, -0.08232885, 1.14664427, 0.01316381],
          [-0.55195999, -0.59205497, 0.93660669, 2.85397242, 0.61310109],
          [0.21420844, 0.04403698, 0.97300744, 0.87568263, -0.67880206]])
```

(3) np.random.randint() 在半闭半开区间 [low,high) 上生成均匀分布的离散整数值；若 high=None，则取值区间变为 [0,low)。

```
In [5]: np.random.randint(20) #在[0,20)上产生1个整数
Out[5]: 10

In [6]: np.random.randint(2, 20) #在[2,20)上产生1个整数
Out[6]: 10

In [7]: np.random.randint(2, 20, 7) #在[2,20)上产生7个整数
Out[7]: array([12, 16, 9, 17, 11, 14, 10])

In [8]: np.random.randint(10, high=None, size=(2,3))
                                  #在[0,10)上产生2行3列的整数数组
Out[8]:
          array([[7, 1, 3],
                 [9, 9, 9]])
```

其他创建数组的方法如下。

np.empty(m,n)：创建 m 行 n 列未初始化的二维数组。

np.ones_like(a)：根据数组 a 的形状生成一个元素全为 1 的数组。

np.zeros_like(a)：根据数组 a 的形状生成一个元素全为 0 的数组。
np.full_like(a,val)：根据数组 a 的形状生成一个元素全为 val 的数组。
np.empty((2,3),np.int)：只分配内存，不进行初始化。
关于各个创建数组方法的使用可以通过 help() 函数来查询。

```
In [9]: help(np.full_like)
Help on function full_like in module numpy.core.numeric:

full_like(a, fill_value, dtype=None, order='K', subok=True)
Return a full array with the same shape and type as a given array.
Parameters
----------
...
Examples
--------
>>> x = np.arange(6, dtype=np.int)
>>> np.full_like(x, 1)
array([1, 1, 1, 1, 1, 1])
>>> np.full_like(x, 0.1)
array([0, 0, 0, 0, 0, 0])
>>> np.full_like(x, 0.1, dtype=np.double)
array([ 0.1, 0.1, 0.1, 0.1, 0.1, 0.1])
>>> np.full_like(x, np.nan, dtype=np.double)
array([ nan, nan, nan, nan, nan, nan])
>>> y = np.arange(6, dtype=np.double)
>>> np.full_like(y, 0.1)
array([ 0.1, 0.1, 0.1, 0.1, 0.1, 0.1])
```

4.1.2 数组的操作

1. 访问数组

对数组元素进行操作，首先要能够索引元素，即查询访问。
索引：每个维度一个索引值，用逗号分隔，其代码如下：

```
In [1]: import numpy as np
   ...: a = np.random.randint(2, 100, 24).reshape((3,8))
   ...: a
Out[1]:
          array([[72, 11, 2, 63, 84, 9, 57, 59],
                 [85, 8, 7, 87, 81, 71, 46, 59],
                 [56, 50, 44, 30, 71, 73, 15, 5]])

In [2]: a[2,6]  #访问行索引号为2，列索引号为6的位置。
Out[2]: 15

In [3]: b = a.reshape((2,3,4))#将a改为三维数组
   ...: b
Out[3]:
          array([[[72, 11, 2, 63],
                  [84, 9, 57, 59],
```

```
                    [85, 8, 7, 87]],
                    [[81, 71, 46, 59],
                    [56, 50, 44, 30],
                    [71, 73, 15, 5]]])

In [4]: b[1,2,3] #访问索引号为[1,2,3]位置上的元素5
Out[4]: 5
```

多维数组的切片：每个维度一个切片值，用逗号分隔，其代码如下：

```
In [5]: b[:,1:,2] #访问元素57、7、44、15
Out[5]:
          array([[57, 7],
                 [44, 15]])
```

访问数组元素的操作代码如下：

```
In [1]: import numpy as np
   ...: c = np.array([[1, 2, 3, 4],[4, 5, 6, 7], [7, 8, 9, 10]])
   ...: c
Out[1]:
          array([[ 1, 2, 3, 4],
                 [ 4, 5, 6, 7],
                 [ 7, 8, 9, 10]])

In [2]: c[1][3]            #访问行索引为1、列索引为3的元素
Out[2]: 7

In [3]: c[:,[1,3]]         #访问c的所有行中的列索引为1、3的元素
Out[3]:
          array([[ 2, 4],
                 [ 5, 7],
                 [ 8, 10]])
```

更多的时候是访问符合条件的元素，若条件为 c[x][y]，则 x 和 y 为条件。

```
In [4]: c[: , 2][c[: , 0] < 5]
Out[4]: array([3, 6])
```

说明如下。

a [x] [y]：表示访问符合 x、y 条件的 a 的元素。

[: , 2]：表示取所有行的第 3 列 (第 3 列索引号为 2)，[c[: , 0] < 5] 表示取第一列 (第 1 列索引号为 0) 中值小于 5 所在的行 (第 1、2 行)，最终表示取第 1、2 行的第 3 列，得到结果 array([3, 6]) 这个“子”数组。

在访问数组时，经常用到查找符合条件元素的位置，此时，可以使用 where() 函数。

```
In [5]: c
Out[5]:
          array([[ 1, 2, 3, 4],
                 [ 4, 5, 6, 7],
                 [ 7, 8, 9, 10]])

In [6]: np.where(c == 4) #查询数据为4的位置
Out[6]: (array([0, 1], dtype=int64), array([3, 0], dtype=int64))
```

这里需要注意的是 [0, 1] 和 [3, 0] 并不是找到的位置的下标，而是表示列表，第一个列表 [0, 1] 表示查询结果的行标，第二个列表 [3, 0] 表示查询结果的列标，即找到的位置为：c[0, 3] 和 c[1, 0](或者 c[0][3] 和 c[1][0])。

2. 数组元素类型转换

当需要对数组中的数据进行类型转换时，常用 astype() 方法。

(1) 浮点数转换为整数。

如果将浮点数转换为整数，则小数部分会被截断。

```
In [1]: import numpy as np
   ...: q = np.array([1.1, 2.2, 3.3, 4.4, 5.53221])
   ...: q
Out[1]: array([ 1.1 , 2.2 , 3.3 , 4.4 , 5.53221])

In [2]: q.dtype
Out[2]: dtype('float64')

In [3]: q.astype(int)
Out[3]: array([1, 2, 3, 4, 5])
```

(2) 字符串数组转换为数值型。

```
In [4]: s = np.array(['1.2','2.3','3.2141'])
   ...: s
Out[4]:
          array(['1.2', '2.3', '3.2141'],
                    dtype='<U6')

In [5]: s.astype(float)
Out[5]: array([ 1.2 , 2.3 , 3.2141])
```

此处给的是 float，而不是 np.float64，Numpy 很智能，会将 Python 类型映射到等价的 dtype 上。

3. 缺失值检测

在进行数据处理前，一般都会对数据进行检测，看是否有缺失项，对缺失值一般要做删除或者填补处理。

np.isnan(a)：检测是不是空值 nan，返回布尔值。

```
In [1]: import numpy as np
   ...: c = np.array([[1, 2, 3, 4],[4, 5, 6, 7], [np.nan, 8, 9, 10]])
   ...: c
Out[1]:
          array([[ 1., 2., 3., 4.],
                 [ 4., 5., 6., 7.],
                 [ nan, 8., 9., 10.]])

In [2]: np.isnan(c)
Out[2]:
          array([[False, False, False, False],
```

```
                    [False, False, False, False],
                    [ True, False, False, False]], dtype=bool)
```

当检测出有缺失值时，可以对缺失值用 0 填补。nan_to_num 可用来将 nan 替换成 0。

```
In [4]: np.nan_to_num(c)
Out[4]:
          array([[ 1., 2., 3., 4.],
                 [ 4., 5., 6., 7.],
                 [ 0., 8., 9., 10.]])
```

4. 查找最大值

在数据分析中，经常会用到查找数据的最大值、最小值，并返回最值的位置。

np.argmax(a, axis=0)：查找每列的最大值位置。

np.argmin(a, axis=0)：查找每列的最小值位置。

a.max(axis=0)：查找每列的最大值。

a.min(axis=0)：查找每列的最小值。

```
In [1]: import numpy as np
   ...: a = np.array([[1,3],[4,2],[8,6]])
   ...: a
Out[1]:
          array([[1, 3],
                 [4, 2],
                 [8, 6]])

In [2]: np.argmax(a,axis=0) #对列进行查找最大数据的位置(索引号)
Out[2]: array([2, 2], dtype=int64)

In [3]: a.max()         #对所有数据进行查找
Out[3]: 8

In [4]: a.max(axis=0)  #对每列查找最大的数据
Out[4]: array([8, 6])
```

4.1.3 数组统计基础

统计分析常用的统计函数如表 4-1 所示。

表 4-1 常用统计函数

函 数	说 明
sum	计算数组中的和
mean	计算数组中的均值
var	计算数组中的方差。方差是元素与元素的平均数差的平方的平均数 var=mean(abs(x-x.mean())**2)
std	计算数组中的标准差。标准差 (standard deviation) 也称为标准偏差，在概率统计中常用于统计分布程度 (statistical dispersion) 上的测量。标准差定义是总体各单位标准值与其平均数离差平方的算术平均数的平方根。它反映组内个体间的离散程度

续表

函　数	说　明
max	计算数组中的最大值
min	计算数组中的最小值
argmax	返回数组中最大元素的索引
argmin	返回数组中最小元素的索引
cumsum	计算数组中所有元素的累计和
cumprod	计算数组中所有元素的累计积

注意：每个统计函数都可以按行和列来统计计算；当 axis=1 时，表示沿着横轴 (行) 计算；当 axis=0 时，表示沿着纵轴 (列) 计算。

```
In [1]: import numpy as np
   ...: c = np.array([[1, 2, 3, 4],[4, 5, 6, 7], [7, 8, 9, 10]])

In [2]: np.sum(c)
Out[2]: 66

In [3]: np.sum(c,axis=0)
Out[3]: array([12, 15, 18, 21])

In [4]: np.sum(c,axis=1)
Out[4]: array([10, 22, 34])

In [5]: np.cumsum(c)
Out[5]: array([ 1, 3, 6, 10, 14, 19, 25, 32, 39, 47, 56, 66])

In [6]: np.cumsum(c,axis=0)
Out[6]:
array([[ 1, 2, 3, 4],
[ 5, 7, 9, 11],
[12, 15, 18, 21]])

In [7]: np.cumsum(c,axis=1)
Out[7]:
array([[ 1, 3, 6, 10],
[ 4, 9, 15, 22],
[ 7, 15, 24, 34]])
```

4.2　Pandas

Pandas 是 Python 的一个数据分析工具包，最初由 AQR 资本管理公司 (AQR Capital Management) 于 2008 年 4 月开发，并于 2009 年年底开源面市。Pandas 最初作为金融数据分析工具被开发出来，因此，Pandas 为时间序列分析提供了很好的支持。

Pandas 中引入了两种新的数据结构—— Series 和 DataFrame，这两种数据结构都建立在 Numpy 库的基础之上。

Series 为一维数组系列，也称序列，与 Numpy 库中的一维 array 类似。二者与 Python 基本的数据结构 list 也很相近。

DataFrame 为二维表格型数据结构。很多功能与 R 语言中的 data.frame 类似。可以将 DataFrame 理解为 Series 的容器。

4.2.1 Series

Series 系列是用于存储一行或一列的数据，以及与之相关的索引的集合。使用方法如下：

```
Series([数据1, 数据2,…], index=[索引1, 索引2,…])
```

例如：

```
In [1]:from pandas import Series
X = Series(['a',2,'螃蟹'],index=[1,2,3])

In [2]:X
Out[2]:
1     a
2     2
3    螃蟹
dtype: object

In [3]:X[3]
Out[3]:'螃蟹'
```

一个序列允许存放多种数据类型，索引也可以省略；可以通过位置或者索引访问数据，如 X[3]，返回 '螃蟹'。

Series 的索引 index 可以省略，默认从 0 开始，也可以指定索引。

在 Spyder 编辑器中写入如下代码：

```
In [1]: from pandas import Series
   ...: A=Series([1,2,3]) #定义系列的时候，数据类型不限
   ...: print(A)
0 1
1 2
2 3
dtype: int64

In [2]: A=Series([1,2,3],index=[1,2,3]) #可自定义索引，如：123、ABCD等
   ...: print(A)
1 1
2 2
3 3
dtype: int64

In [3]: A=Series([1,2,3],index=['A','B','C'])
   ...: print(A)
A 1
B 2
C 3
```

```
dtype: int64
```

注意：容易犯的错误，例如：

```
In [4]: A=Series([1,2,3],index=[A,B,C])
   ...: print(A)
Traceback (most recent call last):

File "C:\Users\yubg\AppData\Local\Temp/ipykernel_11752/3801843389.py",
   line 1, in <module>
A=Series([1,2,3],index=[A,B,C])

        NameError: name 'B' is not defined
```

因为这里 A、B、C 都是字符串，需要使用引号。

访问序列值时，需要通过索引来访问，序列的索引 (index) 和序列值是一一对应的关系，如表 4-2 所示。

表 4-2　序列索引与序列值对应关系

序列索引	序列值
0	14
1	26
2	31

```
In [5]: A=Series([14,26,31])
   ...: print(A)
   ...: print( A[1]) #序列的索引是从0开始计数的

0 14
1 26
2 31
dtype: int64
26

In [6]: print( A[5]) #超出index的总长度会报错
Traceback (most recent call last):
   File "D:\app-soft\anaconda\lib\site-packages\pandas\core\indexes\
     range.py", line 385, in get_loc
    return self._range.index(new_key)
   File "D:\app-soft\anaconda\lib\site-packages\pandas\core\indexes\
     range.py", line 387, in get_loc
    raise KeyError(key) from err

        KeyError: 5

In [7]: A=Series([14,26,31],index=['first','second','third'])
   ...: print(A)
first 14
second 26
third 31
dtype: int64
```

```
In [8]: print(A['second'])#如果设置了index参数，也可通过参数来访问系列值
        26
```

执行下面的代码，看看运行的结果：

```
In [1]: from pandas import Series
   ...: x = Series(['a', True, 1],
   ...:            index=['first', 'second', 'third'])#可以混合定义一个序列

In [2]: x[1]#访问
Out[2]: True

In [3]: x['second']   #根据index访问
Out[3]: True

In [4]: x[3]          #不能越界访问，超出了索引的范围
Traceback (most recent call last):
   File "C:\Users\yubg\AppData\Local\Temp/ipykernel_14236/3918130673.
     py", line 1, in <module>
     x[3]   #不能越界访问
   File "D:\app-soft\anaconda\lib\site-packages\pandas\core\series.py",
     line 939, in __getitem__
     return self._values[key]
IndexError: index 3 is out of bounds for axis 0 with size 3

In [5]: x.append('2')  #不能追加单个元素，但可以追加系列
Traceback (most recent call last):
   File "D:\app-soft\anaconda\lib\site-packages\pandas\core\reshape\
     concat.py", line 294, in concat
     op = _Concatenator
   File "D:\app-soft\anaconda\lib\site-packages\pandas\core\reshape\
     concat.py", line 384, in __init__
           raise TypeError(msg)
TypeError: cannot concatenate object of type '<class 'str'>'; only
    Series and DataFrame objs are valid

In [6]: n = Series(['2'])#追加一个系列
   ...: x.append(n)
Out[6]:
first        a
second    True
third        1
0            2
dtype: object

In [7]: x = x.append(n) #x.append(n)返回的是一个新序列

In [8]: #判断值是否存在，数字和逻辑型(True/False)是不需要加引号的
   ...: 2 in x.values
Out[8]: False

In [9]: '2' in x.values
Out[9]: True

In [10]: x[1:3]#切片
```

```
Out[10]:
second    True
third        1
dtype: object

In [11]: x.drop(0)  #根据index删除，返回新的系列，原系列不变
Out[11]:
first        a
second    True
third        1
dtype: object

In [12]: x.drop('first')
Out[12]:
second    True
third        1
0            2
dtype: object

In [13]: x.drop(x.index[3])  #根据位置删除，返回新的序列
Out[13]:
first        a
second    True
third        1
dtype: object

In [14]: #根据值删除，显示值不等于2的系列，即删除2，返回新序列
    ...: x[2!=x.values]
Out[14]:
first        a
second    True
third        1
0            2
dtype: object

In [15]: x.index[x.values=='a']  #通过值访问系列号index
Out[15]: Index(['first'], dtype='object')

In [16]: #修改series中的index: 可以通过赋值更改，也可以通过reindex()方法
    ...: x.index=[0,1,2,3]

In [17]: x
Out[17]:
0       a
1    True
2       1
3       2
dtype: object

In [18]: x[[0, 2, 1]]  #定位获取，这个方法经常用于随机抽样
Out[18]:
0       a
2       1
1    True
dtype: object
```

```
In [19]: s=Series({'a':1,'b':2,'c':3}) #可将字典转化为Series

In [20]: s
Out[20]:
a    1
b    2
c    3
dtype: int64
```

Series 的 sort_index(ascending=True) 方法可以对 index 进行排序操作，ascending 参数用于控制升序或降序，默认为升序。也可使用 reindex() 方法重新排序。

在 Series 上调用 reindex 重排数据，使得它符合新的索引，如果索引的值不存在，就引入缺失数据值：

```
#reindex重排序
In [21]:
   ...: obj = Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
   ...: obj
Out[21]:
d 4.5
b 7.2
a -5.3
c 3.6
dtype: float64

In [22]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
    ...: obj2
Out[22]:
a -5.3
b 7.2
c 3.6
d 4.5
e NaN
dtype: float64

In [23]: obj.reindex(['a', 'b', 'c', 'd', 'e'], fill_value=0)
Out[23]:
a -5.3
b 7.2
c 3.6
d 4.5
e 0.0
dtype: float64
```

Series 对象本质上是一个 Numpy 的数组，因此 Numpy 的数组处理函数可以直接对 Series 进行处理。但是 Series 除了可以使用位置作为下标存取元素之外，还可以使用标签存取元素，这一点和字典相似。每个 Series 对象实际上都由两个数组组成。

index 是从 Numpy 数组继承的 index 对象，保存标签信息。

values 是保存值的 Numpy 数组。

处理数组函数时应注意以下几点。

(1) Series 是一种类似于一维数组 (数组：ndarray) 的对象。

(2) 它的数据类型没有限制 (各种 Numpy 数据类型)。

(3) 它有索引，把索引当作数据的标签 (key) 看待，类似于字典 (只是类似，实质上是数组)。

(4) Series 同时具有数组和字典的功能，因此它也支持一些字典的方法。

4.2.2 DataFrame

DataFrame 是用于存储多行和多列的数据集合，是 Series 的容器。使用方式如下，其中数据行列位置如图 4-2 所示。

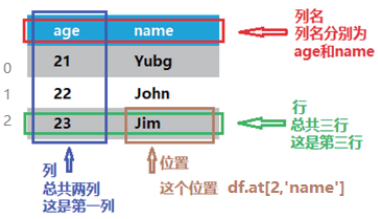


图 4-2 DataFrame 数据行列位置

```
Dataframe(columnsMap)
```

例如：

```
df=DataFrame({'age':Series([26,29,24]),'name':Series(['Ken','Jerry','B
   en'])},
   index=[0,1,2])
In [1] : from pandas import Series
    ...: from pandas import DataFrame
    ... : df=DataFrame({'age':Series([26,29,24]),
                        'name' :Series(['Ken','Jerry','Ben'])})
    ...: print(df)

  age   name
0 26    Ken
1 29    Jerry
2 24    Ben
```

注意：DataFrame 单词的驼峰写法。索引不指定时也可以省略，使用数据框时，要先从 pandas 中导入 DataFrame，数据框中的数据访问方式如表 4-3 所示。

表 4-3 数据框的访问方式

访问位置	方 法	备 注
访问列	变量名 [列名]	访问对应的列。如：df['name']
访问行	变量名 [n：m]	访问 n 行到 m-1 行的数据。如：df[2:3]
访问块 (行和列)	变量名 .iloc[n1:n2,m1:m2]	访问 n1 到 (n2-1) 行，m1 到 (m2-1) 列的数据。 如：df.iloc[0:3,0:2]
访问位置	变量名 .at[行名 , 列名]	访问 (行名 , 列名) 位置的数据。如：df.at[1, 'name']

具体示例如下：

```
In [2]: A=df['age'] #获取age的列值
   ...: print(A)
0 26
1 29
2 24
Name: age, dtype: int64
```

```
In [3]: B=df[1:2] #获取序号为1的行的值
   ...: print(B)
  age name
1 29  Jerry

In [4]: C=df.iloc[0:2,0:2] #获取行序号为0到2(不含)与列序号为0到2(不含)的块
   ...: print(C)
  age name
0 26  Ken
1 29  Jerry

In [5]: D=df.at[0,'name'] #获取行序号为0的行与name列的交叉值
   ...: print(D)
        Ken
```

执行下面的代码并看运行结果：

```
In [6]: from pandas import DataFrame
   ...: df1 = DataFrame({
   ...:     'age': [21, 22, 23],
   ...:     'name': ['KEN', 'John', 'JIMI']});
   ...:
   ...: df2 = DataFrame(data={
   ...:     'age': [21, 22, 23],
   ...:     'name': ['KEN', 'John', 'JIMI']},
   ...:     index=['first', 'second', 'third']);

In [7]: df['age']  #按列访问
Out[7]:
0    26
1    29
2    24
Name: age, dtype: int64

In [8]: df1[1:2]   #按行访问
Out[8]:
   age  name
1   22  John

In [9]: df1.iloc[0:1, 0:1]  #按行列索引号访问
Out[9]:
   age
0   21

In [10]: df2.at["first", 'name']#按行索引名、列名访问
Out[10]: 'KEN'
#说明: 索引号是指默认从0开始的顺序数，索引名是指给定的索引名称，如'first'、'second'等。

In [11]: df1.columns=['age2', 'name2']  #修改列名

In [12]: df1.index = range(1,4)          #修改行索引

In [13]: #访问指定列的值
   ... : df1[df1.columns[0:2]]#等价于column_names=df1.columns,df1[column_
          names[0:2]]
```

```
Out[13]:
   age2 name2
1    21   KEN
2    22  John
3    23  JIMI

In [14]: #根据行索引删除
    ...: df1.drop(1, axis=0) #axis=0表示行轴，也可以省略
Out[14]:
   age2 name2
2    22  John
3    23  JIMI

In [15]: #根据列名进行删除
    ...: df1.drop('age2', axis=1)  #axis=1表示列轴，不可省略
Out[15]:
  name2
1   KEN
2  John
3  JIMI

In [16]: del df1['age2']     #第二种删除列的方法

In [17]: df1['newColumn'] = [2, 4, 6] #增加列
In [18]: df1
Out[18]:
  name2  newColumn
1   KEN          2
2  John          4
3  JIMI          6

In [19]: df2.loc[len(df2)] = [24, "KENKEN"]  #增加行。这种方法效率比较低

In [20]: df2
Out[20]:
        age    name
first    21     KEN
second   22    John
third    23    JIMI
3        24  KENKEN
```

增加行的办法可以通过合并两个 DataFrame 来解决。例如：

```
In [21]: df = DataFrame([[1, 2], [3, 4]], columns=list('AB'))
    ...: df
Out[21]:
  A B
0 1 2
1 3 4

In [22]: df2 = DataFrame([[5, 6], [7, 8]], columns=list('AB'))
    ...: df2
Out[22]:
  A B
0 5 6
```

```
1 7 8

#合并两个数据框，并生成一个新的数据框，简单地“叠加”，但不修改index
In [23]: df.append(df2) #仅把df和df2“叠”起来了，没有修改合并后df2部分的index
Out[23]:
  A B
0 1 2
1 3 4
0 5 6
1 7 8

#添加参数ignore_index=True可修改index
In [24]: df.append(df2, ignore_index=True) #修改index，对df2部分重新索引了
Out[24]:
  A B
0 1 2
1 3 4
2 5 6
3 7 8
```

4.2.3 数据导入

数据存在的形式多种多样，有文件 (txt、csv、excel) 和数据库 (mysql、access、sql server) 等形式。

1. 导入 txt 文件

导入 txt 文件的语法格式如下：

```
read_table(file,names=[列名1，列名2，...],sep="",…)
```

其中，file 为文件路径与文件名；names 为列名，默认为文件中的第一行作为列名；sep 为分隔符，默认为空，表示默认导入为一列。

【例 4-1】读取 (导入)txt 文件。

```
In [1]: from pandas import read_table
   ...: df = read_table(r'D:\book\rz1.txt',
    ...:                names=['YHM','DLSJ','TCSJ','YWXT','IP','REMARK'],
   ...:                sep=",")
   ...: print(df)
     YHM               DLSJ ...            IP              REMARK
0 S1402048 2014-11-04 08:44:46 ... 221.205.98.55 单点登录研究生系统成功!
1 S1411023 2014-11-04 08:45:06 ... 183.184.226.205 单点登录研究生系统成功!
2 S1402048 2014-11-04 08:46:39 ... 221.205.98.55 用户名或密码错误。
3 20031509 2014-11-04 08:47:41 ... 222.31.51.200 统一身份用户登录成功!
4 S1405010 2014-11-04 08:49:03 ... 120.207.64.3 单点登录研究生系统成功!

[5 rows x 6 columns]
```

注意：txt 文件要保存成 UTF-8 格式才不会报错。

2. 导入 csv 文件

导入 csv 文件的语法格式如下：

```
read_csv(file,names=[列名1,列名2,...],sep="",…)
```

其中，file 为文件路径与文件名；names 为列名，默认为文件中的第一行作为列名；sep 为分隔符，默认为空，表示默认导入为一列。

【例 4-2】读取 (导入)csv 文件。

```
In [2]: from pandas import read_csv
   ...: df = read_csv(r'D:\book\rz20.csv',
   ...:       names=['YHM','DLSJ','TCSJ','YWXT','IP','REMARK'],sep=",")
   ...: print(df)

YHM DLSJ TCSJ YWXT IP REMARK
0 id band num price NaN NaN
1 1 130联通 123 159 NaN NaN
2 2 131     124 753 NaN NaN
3 3 132     125 456 NaN NaN
4 4 133电信 126 852 NaN NaN
```

使用 read_table 命令也能执行，结果与 read_csv 一致：

```
In [3]: from pandas import read_table
   ...: df = read_table(r'D:\book\rz20.csv',
   ...:       names=['YHM','DLSJ','TCSJ','YWXT','IP','REMARK'],sep=",")
   ...: print(df)

YHM DLSJ TCSJ YWXT IP REMARK
0 id band num price NaN NaN
1 1 130联通 123 159 NaN NaN
2 2 131     124 753 NaN NaN
3 3 132     125 456 NaN NaN
4 4 133电信 126 852 NaN NaN
```

3. 导入 excel 文件

导入 excel 文件的语法格式如下：

```
read_excel(file, sheet_name,header=0)
```

其中，file 为文件路径与文件名；sheet_name 为 sheet 的名称，如：sheet1；header 为列名，默认为 0，文件的第一行作为列名。只接受布尔型 0 和 1。

【例 4-3】读取 (导入)excel 文件。

```
In [4]: from pandas import read_excel
   ...: df = read_excel(r'D:\book\rz1.xlsx',
   ...:                 sheet_name='Sheet2',
   ...:                 header=1)
   ...: print(df)

  S1411023 2014-11-04 08:45:06 ... 183.184.226.205 单点登录研究生系统成功!
0 S1402048 2014-11-04 08:46:39 ... 221.205.98.55 用户名或密码错误。
1 20031509 2014-11-04 08:47:41 ... 222.31.51.200 统一身份用户登录成功!
```

```
2 S1405010 2014-11-04 08:49:03 ... 120.207.64.3 单点登录研究生系统成功!
3 20031509 2014-11-04 08:47:41 ... 222.31.51.200 统一身份用户登录成功!
4 S1405010 2014-11-04 08:49:03 ... 120.207.64.3 单点登录研究生系统成功!

[5 rows x 5 columns]
```

注意：header 取 0 和 1 的差别，取 0 表示第一行作为表头显示，取 1 表示第一行丢弃不作为表头显示。有时可以跳过首行或者读取多个表，例如：

```
df = pd.read_excel(filefullpath, sheet_name=[0,2],skiprows=[0])
```

sheet_name 可以指定读取几个 sheet，sheet 数目从 0 开始，如果 sheet_name=[0,2]，则代表读取索引号 0 和 2 的 sheet；skiprows=[0] 代表读取时跳过索引号为 0 的行。

4. 导入 MySQL 库

导入 MySQL 库的语法格式如下：

```
read_sql(sql,con=数据库)
```

其中，sql 为从数据库中查询数据的 SQL 语句；con 为数据库的连接对象，需要在程序中选“创建”。

示例代码如下：

```
import pandas
import MySQLdb
connection  = MySQLdb.connect(
    host = '127.0.0.1',#本地机的访问地址
    user = 'root',     #登录名
    passwd = '',       #访问密码，此处无密码
    db = 'test',       #访问的数据库
    port = 5029,       #访问端口
    charset = 'utf8')  #编码格式
data = pandas.read_sql("select * from t_user;",con = connection )
                       #t_user是test库中的表
connection.close()     #调用完要关闭数据库
```

4.2.4 数据导出

1. 导出 csv 文件

导出 csv 文件的语法格式如下：

```
to_csv(file_path,sep= ", ",index=TRUE,header=TRUE)
```

其中，file_path 为文件路径；sep 为分隔符，默认是逗号；index 为是否导出行序号，默认是 TRUE，导出行序号；header 为是否导出列名，默认是 TRUES 导出列名。

【例 4-4】导出 csv 文件。

```
In [1]: from pandas import DataFrame
   ...: from pandas import Series
   ...: df = DataFrame({'age':Series([26,85,64]),
   ...:                 'name':Series(['Ben','John','Jerry'])})
```

```
   ...: print(df)

  age name
0 26  Ben
1 85  John
2 64  Jerry

In [2]: df.to_csv(r'd:\\01.csv') #默认带上index
   ...: df.to_csv(r'd:\\02.csv',index=False) #无index
```

导出数据 01.csv 和 02.csv 结果如图 4-3 所示。

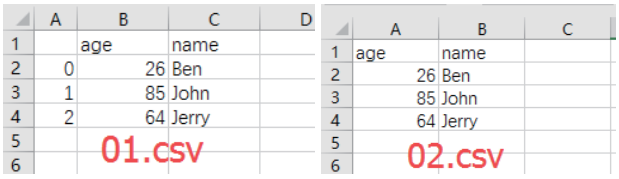


图 4-3　导出数据 01.csv 和 02.csv 结果

2. 导出 excel 文件

导出 excel 文件的语法格式如下：

```
to_excel(file_path, index=TRUE,header=TRUE)
```

其中，file_path 为文件路径；index 为是否导出行序号，默认是 TRUE，导出行序号；header 为是否导出列名，默认是 TRUE，导出列名。

【例 4-5】导出 excel 文件。

```
In [3]: from pandas import DataFrame
   ...: from pandas import Series
   ...: df = DataFrame({'age':Series([26,85,64]),
   ...:                 'name':Series(['Ben','John','Jerry'])})

In [4]: df.to_excel('d:\\01.xlsx') #默认带上index
   ...: df.to_excel('d:\\02.xlsx',index=False) #无index
```

导出数据 01.xlsx 和 02.xlsx 结果如图 4-4 所示。

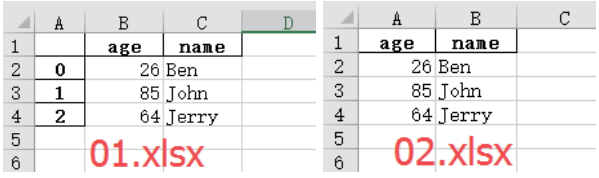


图 4-4　导出数据 01.xlsx 和 02.xlsx 结果

注意：打开 Excel 或 csv 文件出错或乱码的解决办法如下。

(1) 在 read_csv() 中添加参数 engine='python'。

eg：data = pd.read_csv(r'c:\yubg\Airlines.csv',engine='python')

(2) 保存的 csv 文件打开时乱码，则在保存为 csv 文件前加参数 encoding='utf-8_sig'。

eg：df.to_csv('Result.csv',header=1,encoding='utf-8_sig')

3. 导出到 MySQL 库

导出 MySQL 库的语法格式如下：

```
to_sql(tableName, con=数据库链接)
```

其中，tableName 为数据库中的表名；con 为数据库的连接对象，需要在程序中选“创建”。

示例代码如下：

```
# -*- coding: utf-8 -*-
import MySQLdb
from pandas import DataFrame

connection = MySQLdb.connect(
    host='127.0.0.1',
    port=5029,
    user='root',
    passwd='',
    db='test',
    charset='utf8')

connection.autocommit(True) #自动递交数据连接
df = DataFrame({'age': [30, 22, 43],
                'name': ['Jhon', 'jerry', 'Ben']});
df.to_sql("table_1", connection, flavor='mysql', if_exists='append')
#导入MySQL数据库test库下的table_1表中，以append追加的模式
connection.close()
```

4.3 正则表达式

正则表达式 (regular expression)，又称正规表达式、规则表达式等，在代码中常简写为 regex 或 RE。在很多文本编辑器里，正则表达式通常被用来检索、替换那些匹配某个模式的文本，如提取某个网页中所有的 E-mail。

Python 中正则表达式的模块为 re，用 import re 导入。它是一种用来匹配字符串的强有力的武器。其设计思想是用一种描述性的语言来给字符串定义一个规则，凡是符合规则的字符串，我们就认为它“匹配”了，否则该字符串就不合规则。

比如在网上填表时，经常需要填写手机号码，当只有输入正确的格式时才被接收，这就是用正则表达式去匹配数字。

在 Python 的正则模块里，一个数字可以用“\d”匹配；一个既可以是字母又可以是数字的字符，可以用“\w”匹配，如身份证的最后一位；“.”可以匹配任意字符。

先来看看如下的匹配模式。

'00\d'：可以匹配 '007'，但无法匹配 '00A'，也就是说 '00' 后面只能是数字。

'\d\d\d' ：可以匹配 '010'，只可匹配三位数字。

'\w\w\d' ：可以匹配 'py3'，前两位可以是数字或字母，但是第三位只能是数字，如

a12、3a1、223，但不可以匹配 y1w 或者 27f。

'py.' ：可以匹配 'pyc'、'py2'、'py!' 等，最后一位可以是任意的字符。

像 .、\w、\d等有特殊用途、不代表本身字符意义的符号称之为元字符。利用元字符进行组合可以匹配各种字符串。常用的元字符匹配规则如表 4-4 所示。

表 4-4　元字符匹配规则

字　符	描　述
\	将下一个字符标记为一个特殊字符，或一个原义字符，或一个向后引用，或一个八进制转义符。例如，'n' 匹配字符 "n"。'\n' 匹配一个换行符。序列 '\\' 匹配 "\" ，而 "\(" 则匹配 "("
^	匹配输入字符串的开始位置。如果设置了 RegExp 对象的 Multiline 属性，^ 也匹配 '\n' 或 '\r' 之后的位置
$	匹配输入字符串的结束位置。如果设置了 RegExp 对象的 Multiline 属性，$ 也匹配 '\n' 或 '\r' 之前的位置
*	匹配前面的子表达式零次或多次。例如，zo* 能匹配 "z" 以及 "zoo"。* 等价于 {0,}
+	匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}
?	匹配前面的子表达式零次或一次。例如，"do(es)?" 可以匹配 "do" 或 "does" 。? 等价于 {0,1}； 当该字符紧跟在任何一个其他限制符 (*, +, ?, {n}, {n,}, {n,m}) 后面时，匹配模式是非贪婪的。非贪婪模式尽可能少地匹配所搜索的字符串，而默认的贪婪模式则尽可能多地匹配所搜索的字符串。例如，对于字符串 "oooo"，'o+?' 将匹配单个 "o"，而 'o+' 将匹配所有 'o'
{n}	n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但能匹配 "food" 中的两个 o
{n,}	n 是一个非负整数。至少匹配 n 次。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "foooood" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o*'
{n,m}	m 和 n 均为非负整数，其中 n <= m。最少匹配 n 次且最多匹配 m 次。例如，"o{1,3}" 将匹配 "fooooood" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格
.	匹配除换行符 (\n、\r) 之外的任何单个字符。要匹配包括 '\n' 在内的任何字符，请使用像 "(.|\n)" 的模式
(pattern)	匹配 pattern 并获取这一匹配。要匹配圆括号字符，请使用 '\(' 或 '\)'
(?:pattern)	匹配 pattern 但不获取匹配结果，也就是说，这是一个非获取匹配，不进行存储供以后使用。这在使用 " 或 " 字符 (|) 来组合一个模式的各个部分很有用。例如，'industr(?:y|ies)' 就是一个比 'industry|industries' 更简略的表达式
(?=pattern)	正向肯定预查 (look ahead positive assert)，在任何匹配 pattern 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如，"Windows(?=95|98|NT|2000)" 能匹配 "Windows2000" 中的 "Windows"，但不能匹配 "Windows3.1" 中的 "Windows"。预查不消耗字符，也就是说，在一次匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始

续表

字　符	描　述
(?!pattern)	正向否定预查(negative assert)，在任何不匹配 pattern 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如，"Windows(?!95|98|NT|2000)" 能匹配 "Windows3.1" 中的 "Windows"，但不能匹配 "Windows2000" 中的 "Windows"。预查不消耗字符，也就是说，在一次匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始
(?<=pattern)	反向(look behind)肯定预查，与正向肯定预查类似，只是方向相反。例如，"(?<=95|98|NT|2000)Windows" 能匹配 "2000Windows" 中的 "Windows"，但不能匹配 "3.1Windows" 中的 "Windows"
(?<!pattern)	反向否定预查，与正向否定预查类似，只是方向相反。例如，"(?<!95|98|NT|2000)Windows" 能匹配 "3.1Windows" 中的 "Windows"，但不能匹配 "2000Windows" 中的 "Windows"
x|y	匹配 x 或 y。例如，'z|food' 能匹配 "z" 或 "food"。'(z|f)ood' 则匹配 "zood" 或 "food"
[xyz]	字符集合。匹配所包含的任意一个字符。例如，'[abc]' 可以匹配 "plain" 中的 'a'
[^xyz]	负值字符集合。匹配未包含的任意字符。例如，'[^abc]' 可以匹配 "plain" 中的 'p'、'l'、'i'、'n'
[a-z]	字符范围。匹配指定范围内的任意字符。例如，'[a-z]' 可以匹配 'a' 到 'z' 范围内的任意小写字母字符
[^a-z]	负值字符范围。匹配任何不在指定范围内的任意字符。例如，'[^a-z]' 可以匹配任何不在 'a' 到 'z' 范围内的任意字符
\b	匹配一个单词边界，也就是指单词和空格间开始和结束的位置。例如，'er\b' 可以匹配 "never" 中的 'er'，但不能匹配 "verb" 中的 'er'
\B	匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'
\cx	匹配由 x 指明的控制字符。例如，\cM 匹配一个 Control-M 或回车符。x 的值必须为 A ～ Z 或 a ～ z 之一。否则，将 c 视为一个原义的 'c' 字符
\d	匹配一个数字字符。等价于 [0-9]
\D	匹配一个非数字字符。等价于 [^0-9]
\f	匹配一个换页符。等价于 \x0c 和 \cL
\n	匹配一个换行符。等价于 \x0a 和 \cJ
\r	匹配一个回车符。等价于 \x0d 和 \cM
\s	匹配任何空白字符，包括空格、制表符、换页符等。等价于 [\f\n\r\t\v]
\S	匹配任何非空白字符。等价于 [^ \f\n\r\t\v]
\t	匹配一个制表符。等价于 \x09 和 \cI
\v	匹配一个垂直制表符。等价于 \x0b 和 \cK
\w	匹配字母、数字、下划线。等价于 '[A-Za-z0-9_]'
\W	匹配非字母、数字、下划线。等价于 '[^A-Za-z0-9_]'

例如，要匹配出文本“email 120487362@qq.com 1234”中的 E-mail 正则表达式，则匹配模式为：\b[\w.%+-]+@[\w.-]+\.[a-zA-Z]{2,4}\b，E-mail 正则表达式解析如图 4-5 所示。

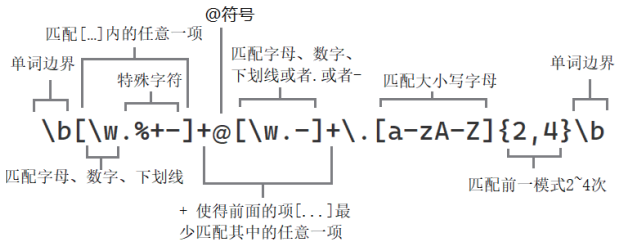


图 4-5　某种 E-mail 正则表达式

当在正则表达式中某个元字符前面放置了一个反斜杠时，就表示反斜杠去掉了元字符的特殊意义，使字符恢复其本身的含义。例如，想在正则表达式中表示字符串“?”，那么需要对元字符“?”进行转义，使用的方法是把一个反斜杠放在元字符“？”前面，即“\?”，这样元字符“？”就失去了其特殊的意义 (“?”表示匹配前面的子表达式零次或一次)，还原回它本身代表的字符。

例如，要提取字符串 " 这些书合计 79.8000 万元还是 79.0000 万元 " 中的数字，可以用如下的匹配模式：

```
(\d{2}\.\d{1,2}0?)
```

\d{2}：表示匹配两位数字。

\.：表示匹配小数点号，由于点号是元字符，所以这里要加“\”转义。

\d{1,2}：表示匹配一到两位数字。

0?：表示匹配 0 的次数为 0 ～ 1 次，结果为 0 或者 00。

所以，上式最终匹配的结果为 79.800 和 79.000。

在正则表达式中还经常出现以下几种组合。

(1).*：表示贪心算法，即尽可能多地匹配，表示匹配任意一个字符的任意次数。

```
In [1]: import re                  #导入re模块
   ...: s = "hello, 中国"
   ...: re.findall(", .*", s)   # findall()函数为在s中匹配给定的匹配模式，后文细讲
Out[1]: [', 中国']                 #从s中按照", .*"匹配的结果

In [2]: s = "hello, 中"
   ...: re.findall(", .*", s)
Out[2]: [', 中']

In [3]: s = "hello, "
   ...: re.findall(", .*", s)
Out[3]: [', ']
```

从上面代码可以看出，.* 匹配的是 . 的 0 次或任意次。所以上面代码的“, .*”表示的是逗号后面可以出现所有字符，也可以不出现任何字符。再如“中 *”匹配出的结果可以是“中”后面任意的字符，也可以匹配出连“中”都不出现的空值。

(2).*?：表示非贪心算法，表示要精确地配对。

```
In [4]: a = 'xxIxxjshdxxlovexxsffaxxpythonxx'
   ...: re.findall('xx(.*?)xx', a)
Out[4]: ['I', 'love', 'python']
```

```
In [5]: a = 'xxIxxjshdxxlovexxsffaxxpythonxx'
   ...: re.findall('xx(.*)xx', a)
Out[5]: ['Ixxjshdxxlovexxsffaxxpython']
```

从 a 中用 'xx(.*?)xx' 去匹配，符合检索条件的有：

```
xxIxx
xxlovexx
xxpythonxx
```

由于匹配模式为 'xx(.*?)xx'，表示只需要获取 () 内的数据，所以最终的结果为：

```
['I', 'love', 'python']
```

如果使用“.*”表达式，则返回仅去掉了头尾的 xx 之后的全部数据，即贪心模式，结果如下：

```
['Ixxjshdxxlovexxsffaxxpython']
```

说明：如果匹配给定文本中的任意汉字，则可以使用 [\u4e00-\u9fa5]；若想匹配连续的多个汉字，则可以使用元字符“+”，即 [\u4e00-\u9fa5]+。

4.3.1 match() 方法

match() 方法用于从字符串的开始位置进行匹配，第一个字符能匹配上的才算匹配成功，返回匹配结果，否则返回 None。

```
match(pattern,string,[flags])
```

参数说明如下。

pattern：表示匹配模式，由正则表达式转换而来，即在正则表达式前加 r 防止转义。

string：表示要被匹配的字符串。

flags：可选参数，用于控制匹配方式，如是否区分字母大小写等。其中 re.A 表示对 \w、\W、\b、\B、\d、\D、\s、\S 只进行 ASCII 匹配；re.I 表示执行不区分字母大小写；re.S 表示匹配所有字符，包括换行符。

【例 4-6】match() 方法。

```
In [1]: import re
   ...: s = "没有一种工作是钱多事少离家近，钱多事少的工作是没有的。"
   ...: pat = "钱多事少"
   ...: res_1 = re.match(pat,s)
   ...: print(res_1)
        None
```

匹配模式“钱多事少”不在 s 的开头部分，而是在 s 的中间，所以匹配不成功，返回的是 None。

```
In [2]: pat = "没有"
   ...: res_2 = re.match(pat,s)
   ...: print(res_2)
        <re.Match object; span=(0, 2), match='没有'>
```

匹配模式“没有”在 s 的开头部分，所以匹配成功，返回结果。结果中显示了匹配成功的起止位置，以及能匹配的字符。要想直接获取匹配的起止位置，可执行 span() 方

法；获取匹配的字符用 group() 方法，示例如下：

```
In [3]: res_2.span()
Out[3]: (0, 2)

In [4]: res_2.group()
Out[4]: '没有'
```

再如，匹配前面的 E-mail。

```
In [5]: s = "120487362@qq.com 1234,  nuc@qq.com"
   ...: pat = r"\b[\w.%+-]+@[\w.-]+\.[a-zA-Z]{2,4}\b"  #需要原始正则表达式加r
   ...: re.match(pat,s,re.I)
Out[5]: <re.Match object; span=(0, 16), match='120487362@qq.com'>
```

注意：匹配模式参数需要防止被转义。也可以使用 re.compile(pattern [, flag]) 进行编译后匹配。

```
In [6]: s = "3wEdf2Sdrf"
   ...: rp = re.compile(r"\d.[a-zA-Z]")
   ...: rp.match(s).group()
Out[6]: '3wE'
```

4.3.2 search() 方法

search() 方法是在字符串中从左至右进行匹配，仅返回第一个匹配上的结果。

```
search(pattern,string,[flags])
```

其参数含义同 match() 方法。

re.search() 和 re.match() 类似，不过 re.search() 不会限制我们只从字符串的开头部分查找匹配，但两者返回的“匹配对象”实际上是一个关于匹配子串的包装类，可通过 span() 和 group() 得到匹配的子串的相关信息。

【例 4-7】search() 方法。

```
In [1]: import re
   ...: s = "没有一种工作是钱多事少离家近，钱多事少的工作是没有的。"
   ...: pat = "钱多事少"
   ...: res_3 = re.search(pat,s)
   ...: print(res_3)
        <re.Match object; span=(7, 11), match='钱多事少'>

In [2]: res_3.group()
Out[2]: '钱多事少'

In [3]: res_3.span()
Out[3]: (7, 11)
```

在字符串 s 中有两处“钱多事少”，但是只返回了第一次匹配上的字符串，即索引值为 7 ～ 11。

4.3.3 findall() 方法

对于 match() 和 search() 方法，总觉得搜索不够完美，对于一般的搜索匹配来说，则希望把能匹配到的都找出来。findall() 方法刚好满足了这个需求，它将匹配成功的子串以列表的形式返回。

```
findall(pattern, string, flags=0)
```

【例 4-8】findall() 方法。

```
In [1]: import re
   ...: s = "没有一种工作是钱多事少离家近，钱多事少的工作是没有的。"
   ...: pat = "钱多事少"
   ...: res_3 = re.findall(pat,s)
   ...: print(res_3)
        ['钱多事少', '钱多事少']

In [2]: str = 'aabbabaabbaac2.b'

In [3]: print(re.findall(r'a.b',str))#符号.就是匹配除\n (换行符)以外的任意一个字符
        ['aab', 'aab']

In [4]: print(re.findall(r'a*b',str))#符号 * 前面的字符出现0次或多次
        ['aab', 'b', 'ab', 'aab', 'b', 'b']

In [5]: print(re.findall(r'a.*b',str))
        #符号.*贪婪匹配，从.*前面开始到后面为结束的所有内容
        ['aabbabaabbaac2.b']

In [6]: print(re.findall(r'a.*?b',str))#符号.*? 非贪婪匹配
        #遇到开始和结束就进行截取，因此截取多次符合的结果，中间没有字符也会被截取
Out[6]: ['aab', 'ab', 'aab', 'aac2.b']

In [7]: print(re.findall(r'a(.*?)b',str))#符号(.*?) 非贪婪匹配
        #与上面一样，只是与上面的形式相比多了一对括号，表示只返回括号里面的内容
Out[7]: ['a', '', 'a', 'ac2.']
```

注意：

(1) In(6) 和 In(7) 的区别。

(2) findall() 方法一旦匹配成功，再次匹配时是将已经匹配成功的截去之后再开始匹配，也可以理解为匹配成功的字符串不再参与下次匹配，匹配出所有的字符串并返回一个列表。而 match() 方法是从字符串的开头位置匹配，类似于“^”功能，search() 方法则是从全局匹配，返回第一个匹配成功的字符串。

4.3.4 查找和替换 re.sub()

在进行数据处理时，经常需要对字符串或文本进行查找和替换。re.sub(a, b, string) 可将匹配模式 a 在 string 中所匹配到的所有字符都替换成 b。

【例 4-9】查找和替换。下面的变量 a，有字母、单引号 (')、换行符 (\n)、数字、冒号 (：)、逗号 (，)，目标是只保留其中的数字和字母。操作如下：

```
In [1]: import re
   ...: a = 'eew \' eawr,2 fd\n sa:21'
   ...: b = re.sub(r'[\':\s ,]*', '', a)
   ...: print(b)
        eeweawr2fdsa21
```

上面正则表达式 r'[\':\s ,]*'的含义有如下几点。

(1) 添加 r，说明该字符串中全为普通字符 (可参考：以 r 或 u 开头的字符串用于防转义)，常用于正则表达式。

(2) [ ] 内是一个字符集，字符集内的任何一个字符被匹配，都算匹配成功，如 r'a[bcd]e'，可以匹配到 'abe'、'ace'、'ade'。

(3) * 代表匹配前一个字符 0 次或多次。

(4) \s 代表的是空白字符，比如空格、换行符、制表符等。

于是 r'[\':\s ,]*'组合起来就是匹配字符串中所有的单引号 (')、换行符 (\n)、冒号 (：)、逗号 (，) 以及空格。

re.sub(r'[\':\s ,]*', '', a) 就是将匹配到的单引号 (')、换行符 (\n)、冒号 (：)、逗号 (，) 以及空格都替换成空。

4.4 数据操作

4.4.1 数据清洗

数据分析的第一步是提高数据质量。数据清洗就是处理缺失数据以及清除无意义的信息。这是数据价值链中最关键的步骤。垃圾数据，即使是通过最好的分析，也将产生错误的结果，并误导业务本身。因此在数据分析过程中，数据清洗占据了大量的工作。

1. 重复值的处理

drop_duplicates() 把数据结构中重复的行数据去除 (保留其中的一行)。

【例 4-10】数据去重。

```
In [1]: from pandas import DataFrame
   ...: from pandas import read_excel
   ...: df = read_excel(r'D:\book\rz2.xlsx')
   ...: df
Out[1]:
     YHM        TCSJ ...           IP                DLSJ
0 S1402048 1.892225e+10 ... 221.205.98.55 2014-11-04 08:44:46
1 S1411023 1.352226e+10 ... 183.184.226.205 2014-11-04 08:45:06
2 S1402048 1.342226e+10 ... 221.205.98.55 2014-11-04 08:46:39
3 20031509 1.882226e+10 ... 222.31.51.200 2014-11-04 08:47:41
4 S1405010 1.892225e+10 ... 120.207.64.3 2014-11-04 08:49:03
5 20140007         NaN ... 222.31.51.200 2014-11-04 08:50:06
6 S1404095 1.382225e+10 ... 222.31.59.220 2014-11-04 08:50:02
7 S1402048 1.332225e+10 ... 221.205.98.55 2014-11-04 08:49:18
8 S1405011 1.892226e+10 ... 183.184.230.38 2014-11-04 08:14:55
```

```
9 S1402048 1.332225e+10 ... 221.205.98.55 2014-11-04 08:49:18
10 S1405011 1.892226e+10 ... 183.184.230.38 2014-11-04 08:14:55

[11 rows x 5 columns]

In [2]: newDF=df.drop_duplicates()
   ...: newDF
Out[2]:
    YHM       TCSJ           YWXT        IP             DLSJ
0 S1402048 1.892225e+10 1.225790e+17 221.205.98.55 2014-11-04 08:44:46
1 S1411023 1.352226e+10 1.225790e+17 183.184.226.205 2014-11-04 08:45:06
2 S1402048 1.342226e+10 NaN 221.205.98.55 2014-11-04 08:46:39
3 20031509 1.882226e+10 NaN 222.31.51.200 2014-11-04 08:47:41
4 S1405010 1.892225e+10 1.225790e+17 120.207.64.3 2014-11-04 08:49:03
5 20140007         NaN 1.225790e+17 222.31.51.200 2014-11-04 08:50:06
6 S1404095 1.382225e+10 1.225790e+17 222.31.59.220 2014-11-04 08:50:02
7 S1402048 1.332225e+10 1.225790e+17 221.205.98.55 2014-11-04 08:49:18
8 S1405011 1.892226e+10 1.225790e+17 183.184.230.38 2014-11-04 08:14:55
```

上面的 df 中索引为 7 和 9 的行数据相同，8 和 10 的行数据相同，去重后原索引为 7、9 和 8、10 行的重复行各保留一行数据。

2. 缺失值的处理

对于缺失数据的处理有数据补齐、删除对应行、不处理等几种方式。

(1) dropna() 方法可以去除数据结构中值为空的数据行。

【例 4-11】删除数据为空所对应的行。

```
In [3]: from pandas import DataFrame
   ...: from pandas import read_excel
   ...: df = read_excel(r'D:\book\rz2.xlsx')
   ...: newDF=df.dropna()
   ...: newDF
Out[3]:
    YHM       TCSJ ...            IP              DLSJ
0 S1402048 1.892225e+10 ... 221.205.98.55 2014-11-04 08:44:46
1 S1411023 1.352226e+10 ... 183.184.226.205 2014-11-04 08:45:06
4 S1405010 1.892225e+10 ... 120.207.64.3 2014-11-04 08:49:03
6 S1404095 1.382225e+10 ... 222.31.59.220 2014-11-04 08:50:02
7 S1402048 1.332225e+10 ... 221.205.98.55 2014-11-04 08:49:18
8 S1405011 1.892226e+10 ... 183.184.230.38 2014-11-04 08:14:55
9 S1402048 1.332225e+10 ... 221.205.98.55 2014-11-04 08:49:18
10 S1405011 1.892226e+10 ... 183.184.230.38 2014-11-04 08:14:55

[8 rows x 5 columns]
```

例 4-10 中的索引为 2、3、5 的行有空值 NaN，已经被删除。

(2) df.fillna() 方法可使用其他数值替代 NaN

有时直接删除空数据会影响数据分析的结果，这时可以对数据进行填补。

【例 4-12】使用数值或者任意字符替代缺失值。

```
In [4]: from pandas import DataFrame
   ...: from pandas import read_excel
```

```
   ...: df = read_excel(r'D:\book\rz2.xlsx')
   ...: df.fillna('?')
Out[4]:
    YHM         TCSJ ...           IP               DLSJ
0 S1402048 18922254812.0 ... 221.205.98.55 2014-11-04 08:44:46
1 S1411023 13522255003.0 ... 183.184.226.205 2014-11-04 08:45:06
2 S1402048 13422259938.0 ... 221.205.98.55 2014-11-04 08:46:39
3 20031509 18822256753.0 ... 222.31.51.200 2014-11-04 08:47:41
4 S1405010 18922253721.0 ... 120.207.64.3 2014-11-04 08:49:03
5 20140007 ? ... 222.31.51.200 2014-11-04 08:50:06
6 S1404095 13822254373.0 ... 222.31.59.220 2014-11-04 08:50:02
7 S1402048 13322252452.0 ... 221.205.98.55 2014-11-04 08:49:18
8 S1405011 18922257681.0 ... 183.184.230.38 2014-11-04 08:14:55
9 S1402048 13322252452.0 ... 221.205.98.55 2014-11-04 08:49:18
10 S1405011 18922257681.0 ... 183.184.230.38 2014-11-04 08:14:55

[11 rows x 5 columns]
```

如 2、3、5 行有空数据，用“？”替代了缺失值。

(3) df.fillna(method='pad') 可以使用前一个数据值替代 NaN。

【例 4-13】用前一个数据值替代缺失值。

```
In [5]: from pandas import DataFrame
   ...: from pandas import read_excel
   ...: df = read_excel(r'D:\book\rz2.xlsx')
   ...: df.fillna(method='pad')
Out[5]:
    YHM        TCSJ ...           IP               DLSJ
0 S1402048 1.892225e+10 ... 221.205.98.55 2014-11-04 08:44:46
1 S1411023 1.352226e+10 ... 183.184.226.205 2014-11-04 08:45:06
2 S1402048 1.342226e+10 ... 221.205.98.55 2014-11-04 08:46:39
3 20031509 1.882226e+10 ... 222.31.51.200 2014-11-04 08:47:41
4 S1405010 1.892225e+10 ... 120.207.64.3 2014-11-04 08:49:03
5 20140007 1.892225e+10 ... 222.31.51.200 2014-11-04 08:50:06
6 S1404095 1.382225e+10 ... 222.31.59.220 2014-11-04 08:50:02
7 S1402048 1.332225e+10 ... 221.205.98.55 2014-11-04 08:49:18
8 S1405011 1.892226e+10 ... 183.184.230.38 2014-11-04 08:14:55
9 S1402048 1.332225e+10 ... 221.205.98.55 2014-11-04 08:49:18
10 S1405011 1.892226e+10 ... 183.184.230.38 2014-11-04 08:14:55

[11 rows x 5 columns]
```

(4) df.fillna(method='bfill') 可以使用后一个数据值替代 NaN。

与 pad 相反，bfill 表示用后一个数据替代 NaN。可以用 limit 限制每列可以替代 NaN 的数目。

【例 4-14】用后一个数据值替代缺失值。

```
In [6]: from pandas import DataFrame
   ...: from pandas import read_excel
   ...: df = read_excel(r'D:\book\rz2.xlsx')
   ...: df.fillna(method='bfill')
Out[6]:
    YHM        TCSJ ...           IP               DLSJ
0 S1402048 1.892225e+10 ... 221.205.98.55 2014-11-04 08:44:46
```

```
1 S1411023 1.352226e+10 ... 183.184.226.205 2014-11-04 08:45:06
2 S1402048 1.342226e+10 ... 221.205.98.55 2014-11-04 08:46:39
3 20031509 1.882226e+10 ... 222.31.51.200 2014-11-04 08:47:41
4 S1405010 1.892225e+10 ... 120.207.64.3 2014-11-04 08:49:03
5 20140007 1.382225e+10 ... 222.31.51.200 2014-11-04 08:50:06
6 S1404095 1.382225e+10 ... 222.31.59.220 2014-11-04 08:50:02
7 S1402048 1.332225e+10 ... 221.205.98.55 2014-11-04 08:49:18
8 S1405011 1.892226e+10 ... 183.184.230.38 2014-11-04 08:14:55
9 S1402048 1.332225e+10 ... 221.205.98.55 2014-11-04 08:49:18
10 S1405011 1.892226e+10 ... 183.184.230.38 2014-11-04 08:14:55

[11 rows x 5 columns]
```

(5) df.fillna(df.mean()) 可以使用平均数或者其他描述性统计量来替代 NaN。

【例 4-15】使用均值来填补数据。

```
In [7]: from pandas import DataFrame
   ...: from pandas import read_excel
   ...: df = read_excel(r'D:\book\rz2.xlsx')
   ...: df
Out[7]:
    YHM       TCSJ ...            IP               DLSJ
0 S1402048 1.892225e+10 ... 221.205.98.55 2014-11-04 08:44:46
1 S1411023 1.352226e+10 ... 183.184.226.205 2014-11-04 08:45:06
2 S1402048 1.342226e+10 ... 221.205.98.55 2014-11-04 08:46:39
3 20031509 1.882226e+10 ... 222.31.51.200 2014-11-04 08:47:41
4 S1405010 1.892225e+10 ... 120.207.64.3 2014-11-04 08:49:03
5 20140007 NaN ... 222.31.51.200 2014-11-04 08:50:06
6 S1404095 1.382225e+10 ... 222.31.59.220 2014-11-04 08:50:02
7 S1402048 1.332225e+10 ... 221.205.98.55 2014-11-04 08:49:18
8 S1405011 1.892226e+10 ... 183.184.230.38 2014-11-04 08:14:55
9 S1402048 1.332225e+10 ... 221.205.98.55 2014-11-04 08:49:18
10 S1405011 1.892226e+10 ... 183.184.230.38 2014-11-04 08:14:55

[11 rows x 5 columns]

In [8]: df.fillna(df.mean())
Out[8]:
    YHM       TCSJ ...            IP               DLSJ
0 S1402048 1.892225e+10 ... 221.205.98.55 2014-11-04 08:44:46
1 S1411023 1.352226e+10 ... 183.184.226.205 2014-11-04 08:45:06
2 S1402048 1.342226e+10 ... 221.205.98.55 2014-11-04 08:46:39
3 20031509 1.882226e+10 ... 222.31.51.200 2014-11-04 08:47:41
4 S1405010 1.892225e+10 ... 120.207.64.3 2014-11-04 08:49:03
5 20140007 1.619226e+10 ... 222.31.51.200 2014-11-04 08:50:06
6 S1404095 1.382225e+10 ... 222.31.59.220 2014-11-04 08:50:02
7 S1402048 1.332225e+10 ... 221.205.98.55 2014-11-04 08:49:18
8 S1405011 1.892226e+10 ... 183.184.230.38 2014-11-04 08:14:55
9 S1402048 1.332225e+10 ... 221.205.98.55 2014-11-04 08:49:18
10 S1405011 1.892226e+10 ... 183.184.230.38 2014-11-04 08:14:55

[11 rows x 5 columns]
```

(6) df.fillna(df.mean()['math': 'physical']) 可以选择列进行缺失值的处理。

【例 4-16】为某列使用该列的均值来填补数据。

```
In [9]: from pandas import DataFrame
   ...: from pandas import read_excel
   ...: df = read_excel(r'D:\book\rz2_0.xlsx')
   ...: df.fillna(df.mean()['math':'physical'])
Out[9]:
No math       physical Chinese
0 1       76.0   85     78.0
1 2       85.0   56     NaN
2 3       76.0   95     85.0
3 4       81.0   75     58.0
4 5       87.0   52     68.0
```

(7) strip() 可以清除字符型数据左、右或首尾指定的字符，默认为空格，中间的不清除。

【例 4-17】删除字符串左、右或首尾指定的字符。

```
In [10]: from pandas import DataFrame
    ...: from pandas import read_excel
    ...: df = read_excel(r'D:\book\rz2.xlsx')
    ...: newDF=df['IP'].str.strip()  #因为IP是一个对象，所以先转换为字符串。
    ...: newDF
Out[10]:
0 221.205.98.55
1 183.184.226.205
2 221.205.98.55
3 222.31.51.200
4 120.207.64.3
5 222.31.51.200
6 222.31.59.220
7 221.205.98.55
8 183.184.230.38
9 221.205.98.55
10 183.184.230.38
Name: IP, dtype: object
```

4.4.2 数据抽取

1. 字段抽取

字段抽取是指抽出某列上指定位置的数据做成新的列，其语法格式如下。

```
slice(start,stop)
```

参数说明如下。

start：开始位置。

stop：结束位置。

【例 4-18】从数据中抽出某列。

```
In [11]: from pandas import DataFrame
    ...: from pandas import read_excel
```

```
    ...: df = read_excel(r'D:\book\rz2.xlsx')
    ...: df['TCSJ']=df['TCSJ'].astype(str) #astype()转化类型
    ...: df['TCSJ']
Out[11]:
0 18922254812.0
1 13522255003.0
2 13422259938.0
3 18822256753.0
4 18922253721.0
5 nan
6 13822254373.0
7 13322252452.0
8 18922257681.0
9 13322252452.0
10 18922257681.0
Name: TCSJ, dtype: object

In [12]: bands = df['TCSJ'].str.slice(0,3)
    ...: bands
Out[12]:
0 189
1 135
2 134
3 188
4 189
5 nan
6 138
7 133
8 189
9 133
10 189
Name: TCSJ, dtype: object

In [13]: areas= df['TCSJ'].str.slice(3,7)
    ...: areas
Out[13]:
0 2225
1 2225
2 2225
3 2225
4 2225
5
6 2225
7 2225
8 2225
9 2225
10 2225
Name: TCSJ, dtype: object

In [14]: tell= df['TCSJ'].str.slice(7,11)
    ...: tell
Out[14]:
0 4812
1 5003
2 9938
```

```
3 6753
4 3721
5
6 4373
7 2452
8 7681
9 2452
10 7681
Name: TCSJ, dtype: object
```

2. 字段拆分

字段拆分是按指定的字符 sep，拆分已有的字符串，其语法格式如下。

```
split(sep,n,expand=False)
```

参数说明如下。

sep：用于分隔字符串的分隔符。

n：分隔后新增的列数。

expand：是否展开为数据框，默认为 False。

返回值：expand 为 True，返回 DataFrame；expand 为 False，返回 Series。

【例 4-19】拆分字符串为指定的列数。

```
In [15]: from pandas import DataFrame
    ...: from pandas import read_excel
    ...: df = read_excel(r'D:\book\rz2.xlsx')
    ...: newDF=df['IP'].str.strip() #IP先转换为字符型，再删除首位空格
    ...: newDF= df['IP'].str.split('.',1,True)
         #按第一个"."分成两列，1表示新增的列数
    ...: newDF
Out[15]:
   0    1
0 221 205.98.55
1 183 184.226.205
2 221 205.98.55
3 222 31.51.200
4 120 207.64.3
5 222 31.51.200
6 222 31.59.220
7 221 205.98.55
8 183 184.230.38
9 221 205.98.55
10 183 184.230.38

In [16]: newDF.columns = ['IP1','IP2-4']     #给第一列、第二列增加列名称
    ...: newDF
Out[16]:
  IP1 IP2-4
0 221 205.98.55
1 183 184.226.205
2 221 205.98.55
3 222 31.51.200
4 120 207.64.3
```

```
5 222 31.51.200
6 222 31.59.220
7 221 205.98.55
8 183 184.230.38
9 221 205.98.55
10 183 184.230.38
```

3. 记录抽取

记录抽取是指根据一定的条件，对数据进行抽取，其语法格式如下。

```
dataframe[condition]
```

其中，condition 为过滤条件。

返回值：DataFrame。

常用的 condition 类型主要有以下几种。

比较运算：<、>、>=、<=、!=，如 df[df.comments>10000)]。

范围运算：between(left,right)，如 df[df.comments.between(1000,10000)]。

空值运算：pandas.isnull(column) ，如 df[df.title.isnull()]。

字符匹配：str.contains(pattern,na = False)，如 df[df.title.str.contains(' 电台 ',na=False)]。

逻辑运算：&(与)、|(或)、not(取反)，如 df[(df.comments>=1000)&(df.comments<=10000)] 与 df[df.comments.between(1000,10000)] 等价。

【例 4-20】按条件抽取数据。

```
In [17]: import pandas
    ...: from pandas import read_excel
    ...: df = read_excel(r'D:\book\rz2.xlsx')
    ...: df[df.TCSJ==13322252452]
Out[17]:
    YHM       TCSJ           YWXT          IP              DLSJ
7 S1402048 1.332225e+10 1.225790e+17 221.205.98.55 2014-11-04 08:49:18
9 S1402048 1.332225e+10 1.225790e+17 221.205.98.55 2014-11-04 08:49:18

In [18]: df[df.TCSJ>13500000000]
Out[18]:
    YHM       TCSJ ...            IP              DLSJ
0 S1402048 1.892225e+10 ... 221.205.98.55 2014-11-04 08:44:46
1 S1411023 1.352226e+10 ... 183.184.226.205 2014-11-04 08:45:06
3 20031509 1.882226e+10 ... 222.31.51.200 2014-11-04 08:47:41
4 S1405010 1.892225e+10 ... 120.207.64.3 2014-11-04 08:49:03
6 S1404095 1.382225e+10 ... 222.31.59.220 2014-11-04 08:50:02
8 S1405011 1.892226e+10 ... 183.184.230.38 2014-11-04 08:14:55
10 S1405011 1.892226e+10 ... 183.184.230.38 2014-11-04 08:14:55

[7 rows x 5 columns]

In [19]: df[df.TCSJ.between(13400000000,13999999999)]
Out[19]:
    YHM       TCSJ          YWXT            IP            DLSJ
1 S1411023 1.352226e+10 1.225790e+17 183.184.226.205 2014-11-04 08:45:06
2 S1402048 1.342226e+10 NaN 221.205.98.55 2014-11-04 08:46:39
```

```
6 S1404095 1.382225e+10 1.225790e+17 222.31.59.220 2014-11-04 08:50:02


In [20]: df[df.YWXT.isnull()]
Out[20]:
    YHM       TCSJ     YWXT         IP           DLSJ
2 S1402048 1.342226e+10 NaN 221.205.98.55 2014-11-04 08:46:39
3 20031509 1.882226e+10 NaN 222.31.51.200 2014-11-04 08:47:41

In [21]: df[df.IP.str.contains('222.',na=False)]
Out[21]:
    YHM       TCSJ     YWXT      IP              DLSJ
3 20031509 1.882226e+10 NaN 222.31.51.200 2014-11-04 08:47:41
5 20140007 NaN 1.225790e+17 222.31.51.200 2014-11-04 08:50:06
6 S1404095 1.382225e+10 1.225790e+17 222.31.59.220 2014-11-04 08:50:02
```

4. 随机抽样

随机抽样是指随机从数据中按照一定的行数或者比例抽取数据，其语法格式如下。

```
numpy.random.randint(start,end,num)
```

参数说明如下。

start：范围的开始值。

end：范围的结束值。

num：抽样个数。

返回值：行的索引值序列。

【例 4-21】随机抽取数据。

```
In [22]: import numpy
    ...: import pandas
    ...: from pandas import read_excel
    ...: df = read_excel(r'D:\book\rz2.xlsx')
    ...: df
Out[22]:
    YHM       TCSJ ...           IP              DLSJ
0 S1402048 1.892225e+10 ... 221.205.98.55 2014-11-04 08:44:46
1 S1411023 1.352226e+10 ... 183.184.226.205 2014-11-04 08:45:06
2 S1402048 1.342226e+10 ... 221.205.98.55 2014-11-04 08:46:39
3 20031509 1.882226e+10 ... 222.31.51.200 2014-11-04 08:47:41
4 S1405010 1.892225e+10 ... 120.207.64.3 2014-11-04 08:49:03
5 20140007 NaN ... 222.31.51.200 2014-11-04 08:50:06
6 S1404095 1.382225e+10 ... 222.31.59.220 2014-11-04 08:50:02
7 S1402048 1.332225e+10 ... 221.205.98.55 2014-11-04 08:49:18
8 S1405011 1.892226e+10 ... 183.184.230.38 2014-11-04 08:14:55
9 S1402048 1.332225e+10 ... 221.205.98.55 2014-11-04 08:49:18
10 S1405011 1.892226e+10 ... 183.184.230.38 2014-11-04 08:14:55

[11 rows x 5 columns]

In [23]: r = numpy.random.randint(0,10,3)
    ...: r
Out[23]: array([2, 0, 9])
```

```
In [24]: df.loc[r,:] #抽取r行数据
Out[24]:
    YHM       TCSJ     YWXT       IP               DLSJ
2 S1402048 1.342226e+10 NaN 221.205.98.55 2014-11-04 08:46:39
0 S1402048 1.892225e+10 1.225790e+17 221.205.98.55 2014-11-04 08:44:46
9 S1402048 1.332225e+10 1.225790e+17 221.205.98.55 2014-11-04 08:49:18
```

按照指定条件抽取数据。

(1) 使用 index 标签选取数据：df.loc[行标签，列标签]

```
df.loc['a':'b']            #选取a到b行的数据，假设a、b为行索引
df.loc['a':'b','TCSJ']     #选取a到b行的TCSJ列的数据
```

```
In [25]: df.loc[3:6]
Out[25]:
    YHM       TCSJ     YWXT       IP               DLSJ
3 20031509 1.882226e+10 NaN 222.31.51.200 2014-11-04 08:47:41
4 S1405010 1.892225e+10 1.225790e+17 120.207.64.3 2014-11-04 08:49:03
5 20140007 NaN 1.225790e+17 222.31.51.200 2014-11-04 08:50:06
6 S1404095 1.382225e+10 1.225790e+17 222.31.59.220 2014-11-04 08:50:02

In [26]: df.loc[:,'TCSJ'] #选取TCSJ列的数据
Out[26]:
0 1.892225e+10
1 1.352226e+10
2 1.342226e+10
3 1.882226e+10
4 1.892225e+10
5 NaN
6 1.382225e+10
7 1.332225e+10
8 1.892226e+10
9 1.332225e+10
10 1.892226e+10
Name: TCSJ, dtype: float64
```

df.loc 的第一个参数是行标签，第二个参数是列标签 (可选参数，默认为所有列标签)，两个参数既可以是列表也可以是字符，如果两个参数都为列表，则返回的是 DataFrame，否则为 Series。

(2) 使用切片位置选取数据：df.iloc[行位置，列位置]。

```
df.iloc[1,1]        #选取第二行、第二列的值，返回的为单个值
df.iloc[[0,2],:]    #选取第一行和第三行的数据
df.iloc[0:2,:]      #选取第一行到第三行(不包含)的数据
df.iloc[:,1]        #选取所有记录的第一列的值，返回的是一个Series
df.iloc[1,:]        #选取第一行数据，返回的是一个Series
```

说明：

① loc 为 location 的缩写，iloc 则为 integer & location 的缩写。iloc 为整型索引，即默认的从 0 开始的自然数索引；loc 为给定的字符串索引，即索引名。

② Python 默认的行序号是从 0 开始的自然数，我们称为行位置。我们在计数时，实际上 0 开始的行为第 1 行，也称为行号，行号是从 1 开始；有时 index 是被命名的，如 'one'、'two'、'three'、'four' 或 'a'、'b'、'c'、'd' 等字符串，我们称之为标签即前面说的索引名。

loc 索引的是行号、标签，即索引名，不是行位置，如下例中 In[30] 行的 df2.loc[1] 索引的是第一行 (行号为 1)，其实位置为 0 行；iloc 索引的是位置，即索引名，不能是标签或行号。

```
In [27]: import pandas as pd
    ...: index_loc = ['a','b']
    ...: index_iloc = [1,2]
    ...: data = [[1,2,3,4],[5,6,7,8]]
    ...: columns = ['one','two','three','four']
    ...: df1 = pd.DataFrame(data=data,index=index_loc,columns=columns)
    ...: df2 = pd.DataFrame(data=data,index=index_iloc,columns=columns)
    ...:
    ...: print(df1.loc['a'])

one 1
two 2
three 3
four 4
Name: a, dtype: int64

In [28]: print(df1.iloc['a']) #iloc不能用索引名
Traceback (most recent call last):
File "C:\Users\yubg\AppData\Local\Temp/ipykernel_21312/3204539269.py",
   line 1, in <module>
print(df1.iloc['a']) #iloc不能用索引名
File "D:\app-soft\anaconda\lib\site-packages\pandas\core\indexing.py",
   line 1563, in _getitem_axis
raise TypeError("Cannot index by location index with a non-integer
   key")

TypeError: Cannot index by location index with a non-integer key

In [29]: print(df2.iloc[1]) #iloc索引的是行位置，即索引号

one 5
two 6
three 7
four 8
Name: 2, dtype: int64

In [30]: print(df2.loc[1]) #loc[1]索引的是行号，对应的行位置为0行

one 1
two 2
three 3
four 4
Name: 1, dtype: int64
```

(3) 通过逻辑指针进行数据切片：df [逻辑条件]。

```
In [31]: df[df. TCSJ >= 18822256753] #单个逻辑条件
Out[31]:
     YHM      TCSJ           YWXT          IP              DLSJ
0 S1402048 1.892225e+10 1.225790e+17 221.205.98.55 2014-11-04 08:44:46
3 20031509 1.882226e+10 NaN 222.31.51.200 2014-11-04 08:47:41
```

```
4 S1405010 1.892225e+10 1.225790e+17 120.207.64.3 2014-11-04 08:49:03
8 S1405011 1.892226e+10 1.225790e+17 183.184.230.38 2014-11-04 08:14:55
10  S1405011  1.892226e+10  1.225790e+17  183.184.230.38  2014-11-04
    08:14:55

In [32]: df[(df. TCSJ >=13422259938 )&(df. TCSJ < 13822254373)]
         #多个逻辑条件组合
Out[32]:
    YHM       TCSJ         YWXT             IP            DLSJ
1 S1411023 1.352226e+10 1.225790e+17 183.184.226.205 2014-11-04 08:45:06
2 S1402048 1.342226e+10 NaN 221.205.98.55 2014-11-04 08:46:39
```

这种方式获取的数据切片都是 DataFrame。

```
In [33]: df[df.TCSJ >= 18822256753]
Out[33]:
     YHM      TCSJ           YWXT         IP              DLSJ
0 S1402048 1.892225e+10 1.225790e+17 221.205.98.55 2014-11-04 08:44:46
3 20031509 1.882226e+10 NaN 222.31.51.200 2014-11-04 08:47:41
4 S1405010 1.892225e+10 1.225790e+17 120.207.64.3 2014-11-04 08:49:03
8 S1405011 1.892226e+10 1.225790e+17 183.184.230.38 2014-11-04 08:14:55
10  S1405011  1.892226e+10  1.225790e+17  183.184.230.38  2014-11-04
    08:14:55
```

5. 字典数据抽取

将字典数据抽取为 DataFrame，有三种方法。

(1) 将字典的 key 和 value 各作为一列。

```
In [34]: import pandas
    ...: from pandas import DataFrame
    ...:
    ...: d1={'a':'[1,2,3]','b':'[0, 1,2]'}
    ...: a1=pandas.DataFrame.from_dict(d1,
            orient='index') #将字典转化为DataFrame，且key列做成了index
    ...: a1.index.name = 'key'     #将index的列名改成key
    ...: b1=a1.reset_index()       #重新增加index，并将原index做成了key列
    ...: b1.columns=['key','value'] #对列重新命名为key和value
    ...: b1
Out[34]:
key value
0 a [1,2,3]
1 b [0, 1,2]
```

(2) 将字典里的每一个元素作为一列 (同长)。

```
In [35]:
    ...: d2={'a':[1,2,3],'b':[4,5,6]} #字典的value必须长度相等
    ...: a2= DataFrame(d2)
    ...: a2
Out[35]:
a b
0 1 4
1 2 5
2 3 6
```

(3) 将字典里的每一个元素作为一列 (不同长)。

```
In [36]:
    ...: d = {'one' : pandas.Series([1, 2, 3]),
              'two' : pandas.Series([1, 2, 3, 4])} #字典的value长度可以不相等
    ...: df = pandas.DataFrame(d)
    ...: df
Out[36]:
one two
0 1.0 1
1 2.0 2
2 3.0 3
3 NaN 4
```

字典数据抽取也可以做如下处理：

```
In [37]: import pandas
    ...: from pandas import Series
    ...: import numpy as np
    ...: from pandas import DataFrame
    ...:
    ...: d = dict( A = np.array([1,2]), B = np.array([1,2,3,4]) )
    ...: DataFrame(dict([(k,Series(v)) for k,v in d.items()]))
Out[37]:
A  B
0 1.0 1
1 2.0 2
2 NaN 3
3 NaN 4
```

字典数据抽取还可以做如下处理：

```
In [38]: import numpy as np
    ...: import pandas as pd
    ...:
    ...: my_dict = dict( A = np.array([1,2]), B = np.array([1,2,3,4]) )
    ...: df = pd.DataFrame.from_dict(my_dict, orient='index').T
    ...: df
Out[38]:
A    B
0 1.0 1.0
1 2.0 2.0
2 NaN 3.0
3 NaN 4.0
```

4.4.3 排名索引

1. 排名排序

Series 的 sort_index(ascending=True) 方法可以对 index 进行排序操作，ascending 参数用于控制升序或降序，默认为升序。

在 DataFrame 上，sort_index(axis=0, ascending=True) 方法多了一个轴向的选择参数。

```
In [1]: from pandas import DataFrame
   ...: df0={'Ohio':[0,6,3],'Texas':[7,4,1],'California':[2,8,5]}
   ...: df=DataFrame(df0,index=['a','c','b'])
   ...: df
Out[1]:
Ohio Texas California
a 0 7 2
c 6 4 8
b 3 1 5

In [2]: df.sort_index()  #按照索引号进行排序
Out[2]:
Ohio Texas California
a 0 7 2
b 3 1 5
c 6 4 8

In [3]: df.sort_index(axis=1) #按列名进行排序
Out[3]:
California Ohio Texas
a 2 0 7
c 8 6 4
b 5 3 1
```

sort_values() 按某一列或多列进行排序。

```
In [4]: df.sort_values(by=['California','Texas']) #按某一列或多列进行排序
Out[4]:
Ohio Texas California
a 0 7 2
b 3 1 5
c 6 4 8
```

排名 (Series.rank(method='average', ascending=True)) 与排序的不同之处在于，它会把对象的 values 替换成名次 (从 1 ～ n)，对于平级项可以通过方法里的 method 参数来处理，method 参数有四个可选项：average、min、max、first。部分参数举例如下：

```
In[51]: from pandas import Series
   ...: ser=Series([3,2,0,3],index=list('abcd'))
   ...: ser
Out[5]:
a 3
b 2
c 0
d 3
dtype: int64

In [6]: ser.rank()
Out[6]:
a 3.5
b 2.0
c 1.0
d 3.5
dtype: float64
```

```
In [7]: ser.rank(method='min')
Out[7]:
a 3.0
b 2.0
c 1.0
d 3.0
dtype: float64

In [8]: ser.rank(method='max')
Out[8]:
a 4.0
b 2.0
c 1.0
d 4.0
dtype: float64

In [9]: ser.rank(method='first')
Out[9]:
a 3.0
b 2.0
c 1.0
d 4.0
dtype: float64
```

注意：在 ser[0] 和 ser[3] 这对平级项上，不同 method 参数表现出不同的名次。DataFrame 的 .rank(axis=0, method='average', ascending=True) 方法多了 axis 参数，可选择按行或按列分别进行排名。

2. 重新索引

Series 对象的重新索引通过其 reindex(index=None,**kwargs) 方法实现。 **kwargs 中常用的参数有两个：method=None 和 fill_value=np.NaN。

```
In [1]: from pandas import Series
   ...: ser = Series([4.5,7.2,-5.3,3.6],index=['d','b','a','c'])
   ...: A = ['a','b','c','d','e']
   ...: ser.reindex(A)
Out[1]:
a -5.3
b 7.2
c 3.6
d 4.5
e NaN
dtype: float64

In [2]: ser = ser.reindex(A,fill_value=0)
   ...: ser
Out[2]:
a -5.3
b 7.2
c 3.6
d 4.5
e 0.0
dtype: float64
```

```
In [3]: ser.reindex(A,method='ffill')
Out[3]:
a -5.3
b 7.2
c 3.6
d 4.5
e 0.0
dtype: float64

In [4]: ser.reindex(A,fill_value=0,method='ffill')
Out[4]:
a -5.3
b 7.2
c 3.6
d 4.5
e 0.0
dtype: float64
```

reindex() 方法会返回一个新的对象，其 index 严格遵循给出的参数顺序，method: {'backfill', 'bfill', 'pad', 'ffill', None} 参数用于指定插值 (填充) 方式，当没有给出时，默认用 fill_value 填充，值为 NaN；而 ffill = pad 和 bfill = back fill 分别指插值时向前或向后取值。

DataFrame 的重新索引方法为 reindex(index=None,columns=None,**kwargs)，仅比 Series 多了一个可选的 columns 参数，用于给列索引。用法与上例 Series 类似，只不过插值方法 method 参数只能应用于行，即轴 axis = 0。

```
In [1]: import numpy as np
   ...: import pandas as pd
   ...: from pandas import DataFrame
   ...: df1=pd.DataFrame(np.arange(9).reshape(3,3),
index=['a','c','d'],
columns=['one','two','four'])
   ...: df1
Out[1]:
one two four
a 0 1 2
c 3 4 5
d 6 7 8

In [2]: df1.reindex(['a','b','c','d'])
Out[2]:
one two four
a 0.0 1.0 2.0
b NaN NaN NaN
c 3.0 4.0 5.0
d 6.0 7.0 8.0

In [3]: df1.reindex(index=['a','b','c','d'],columns=['one','two','three',
          'four'])
Out[3]:
one two three four
a 0.0 1.0 NaN 2.0
b NaN NaN NaN NaN
c 3.0 4.0 NaN 5.0
```

```
d 6.0 7.0 NaN 8.0

In [4]: df1.reindex(index=['a','b','c','d'],
          columns=['one','two','three','four'],
          fill_value=100)
Out[4]:
one two three four
a 0 1 100 2
b 100 100 100 100
c 3 4 100 5
d 6 7 100 8
```

4.4.4 数据合并

1. 记录合并

记录合并是指两个结构相同的数据框合并成一个数据框，也就是在一个数据框中追加另一个数据框的所有数据，其语法格式如下。

```
concat([dataFrame1, dataFrame2,…])
```

其中，DataFrame1 为数据框。

返回值：DataFrame。

【例 4-22】合并两个数据框，即合并记录。

```
In [1]: import pandas
   ...: from pandas import DataFrame
   ...: from pandas import read_excel
   ...:
   ...: df1 = read_excel(r'D:\book\rz2.xlsx')
   ...: df1
Out[1]:
    YHM       TCSJ ...            IP              DLSJ
0 S1402048 1.892225e+10 ... 221.205.98.55 2014-11-04 08:44:46
1 S1411023 1.352226e+10 ... 183.184.226.205 2014-11-04 08:45:06
2 S1402048 1.342226e+10 ... 221.205.98.55 2014-11-04 08:46:39
3 20031509 1.882226e+10 ... 222.31.51.200 2014-11-04 08:47:41
4 S1405010 1.892225e+10 ... 120.207.64.3 2014-11-04 08:49:03
5 20140007 NaN ... 222.31.51.200 2014-11-04 08:50:06
6 S1404095 1.382225e+10 ... 222.31.59.220 2014-11-04 08:50:02
7 S1402048 1.332225e+10 ... 221.205.98.55 2014-11-04 08:49:18
8 S1405011 1.892226e+10 ... 183.184.230.38 2014-11-04 08:14:55
9 S1402048 1.332225e+10 ... 221.205.98.55 2014-11-04 08:49:18
10 S1405011 1.892226e+10 ... 183.184.230.38 2014-11-04 08:14:55

[11 rows x 5 columns]

In [2]: df2 = read_excel(r'D:\book\rz3.xlsx')
   ...: df2
Out[2]:
    YHM      TCSJ           YWXT          IP              DLSJ
0 S1402011 18603514812 1.225790e+17 221.205.98.55 2014-11-04 08:44:46
1 S1411022 13103515003 1.225790e+17 183.184.226.205 2014-11-04 08:45:06
```

```
2 S1402033 13203559930 NaN 221.205.98.55 2014-11-04 08:46:39

In [3]: df=pandas.concat([df1,df2])
   ...: df
Out[3]:
    YHM       TCSJ ...            IP               DLSJ
0 S1402048 1.892225e+10 ... 221.205.98.55 2014-11-04 08:44:46
1 S1411023 1.352226e+10 ... 183.184.226.205 2014-11-04 08:45:06
2 S1402048 1.342226e+10 ... 221.205.98.55 2014-11-04 08:46:39
3 20031509 1.882226e+10 ... 222.31.51.200 2014-11-04 08:47:41
4 S1405010 1.892225e+10 ... 120.207.64.3 2014-11-04 08:49:03
5 20140007 NaN ... 222.31.51.200 2014-11-04 08:50:06
6 S1404095 1.382225e+10 ... 222.31.59.220 2014-11-04 08:50:02
7 S1402048 1.332225e+10 ... 221.205.98.55 2014-11-04 08:49:18
8 S1405011 1.892226e+10 ... 183.184.230.38 2014-11-04 08:14:55
9 S1402048 1.332225e+10 ... 221.205.98.55 2014-11-04 08:49:18
10 S1405011 1.892226e+10 ... 183.184.230.38 2014-11-04 08:14:55
0 S1402011 1.860351e+10 ... 221.205.98.55 2014-11-04 08:44:46
1 S1411022 1.310352e+10 ... 183.184.226.205 2014-11-04 08:45:06
2 S1402033 1.320356e+10 ... 221.205.98.55 2014-11-04 08:46:39

[14 rows x 5 columns]
```

两个文件的数据记录都合并到一起，实现了数据记录的“叠加”或者记录顺延。但是索引号没有顺延，而是保留了各自的原索引，增加一个 ignore_index 参数即可实现索引顺延，即：

```
pandas.concat([df1,df2],ignore_index=True)
```

2. 字段合并

字段合并是指将同一个数据框中的不同的列进行合并，形成新的列，其语法格式如下。

```
X = x1+x2+…
```

参数说明如下。

x1：数据列 1。

x2：数据列 2。

返回值：Series，合并后的系列与合并前各列的长度一致。

【例 4-23】多个字段合并成一个新的字段。

```
In [1]: import pandas
   ...: from pandas import DataFrame
   ...: from pandas import read_csv
   ...: df = read_csv(r'D:\book\rz4.csv',sep=" ",names=['band','area',
             'num'])
   ...: df
Out[1]:
band area num
0 189 2225 4812
1 135 2225 5003
2 134 2225 9938
```

```
3 188 2225 6753
4 189 2225 3721
5 134 2225 9313
6 138 2225 4373
7 133 2225 2452
8 189 2225 7681

In [2]: df = df.astype(str)
   ...: tel=df['band']+df['area']+df['num']
   ...: tel
Out[2]:
0 18922254812
1 13522255003
2 13422259938
3 18822256753
4 18922253721
5 13422259313
6 13822254373
7 13322252452
8 18922257681
dtype: object
```

3. 字段匹配

字段匹配是指不同结构的数据框 (两个或以上的数据框)，按照一定的条件进行合并，即追加列，类似 Excel 中的 vlookup 函数，其语法格式如下。

```
merge(x,y,left_on,right_on)
```

参数说明如下。

x：第一个数据框。

y：第二个数据框。

left_on：第一个数据框用于匹配的列。

right_on：第二个数据框用于匹配的列。

返回值：DataFrame。

【例 4-24】按指定唯一字段匹配增加列。

```
In [1]: import pandas
   ...: from pandas import DataFrame
   ...: from pandas import read_excel
   ...: df1 = read_excel(r'D:\book\rz2.xlsx',sheet_name='Sheet3')
   ...: df1
Out[1]:
id band num
0 1 130 123
1 2 131 124
2 4 133 125
3 5 134 126

In [2]: df2 = read_excel(r'D:\book\rz2.xlsx',sheet_name='Sheet4')
   ...: df2
Out[2]:
```

```
id band area
0 1 130 351
1 2 131 352
2 3 132 353
3 4 133 354
4 5 134 355
5 5 135 356

In [3]: pandas.merge(df1,df2,left_on='id',right_on='id')
Out[3]:
id band_x num band_y area
0 1 130 123 130 351
1 2 131 124 131 352
2 4 133 125 133 354
3 5 134 126 134 355
4 5 134 126 135 356
```

这里只匹配了有相同序号的行，如 df1 中没有 id=3，在结果中也没有 id=3；在 df2 中有两个 id=5，在结果中也有两个 id=5，但是只匹配第一个 id=5。

4.4.5 数据计算

1. 简单计算

简单计算是指对各字段进行加、减、乘、除四则运算，其计算出的结果作为新的字段，如图 4-5 所示。

id	num	price
1	123	159
2	124	753
3	125	456
4	126	852

id	num	price	result
1	123	159	19557
2	124	753	93372
3	125	456	57000
4	126	852	107352

图 4-5 字段之间的运算结果作为新的字段

```
In [1]: from pandas import read_csv
   ...: df = read_csv(r'D:\book\rz2.csv',sep=',')
   ...: df
Out[1]:
id band num price
0 1 130 123 159
1 2 131 124 753
2 3 132 125 456
3 4 133 126 852

In [2]: result=df.price*df.num
   ...: result
Out[2]:
0 19557
1 93372
2 57000
3 107352
dtype: int64
```

```
In [3]: df['result']=result
   ...: df
Out[3]:
id band num price result
0 1 130 123 159 19557
1 2 131 124 753 93372
2 3 132 125 456 57000
3 4 133 126 852 107352
```

2. 数据标准化

数据标准化是指将数据按照比例缩放，使之落入特定的区间，一般使用 0-1 标准化，其语法格式如下。

```
X*=(x-min)/(max-min)
```

```
In [4]: from pandas import read_csv
   ...: df = read_csv(r'D:\book\rz2.csv',sep=',')
   ...: df
Out[4]:
id band num price
0 1 130 123 159
1 2 131 124 753
2 3 132 125 456
3 4 133 126 852

In [5]: scale=(df.price-df.price.min())/(df.price.max()-df.price.min())
   ...: scale
Out[5]:
0 0.000000
1 0.857143
2 0.428571
3 1.000000
Name: price, dtype: float64
```

4.4.6 数据分组

数据分组就是根据数据分析对象的特征，按照一定的数据指标，把数据划分为不同的区间来进行研究，以揭示其内在的联系和规律性。简单来说，就是新增一列，将原来的数据按照其性质归入新的类别中，其语法格式如下。

```
cut(series,bins,right=True,labels=NULL)
```

参数说明如下。

series：需要分组的数据。

bins：分组的依据数据。

right：分组的时候右边是否闭合。

abels：分组的自定义标签，可以不自定义。

现有数据如图 4-6 所示，下面将数据进行分组。

序号	品牌	数据	价格
1	130	123	159
2	131	124	753
3	132	125	456
4	133	126	852

序号	品牌	数据	价格	类别
1	130	123	159	价格 500 以下
2	131	124	753	价格 500 以上
3	132	125	456	价格 500 以下
4	133	126	852	价格 500 以上

图 4-6　数据分组

```
In [1]: import pandas
   ...: from pandas import read_csv
   ...:
   ...: df = read_csv(r'D:\book\rz2.csv',sep=',')
   ...: df
Out[1]:
id band num price
0 1 130 123 159
1 2 131 124 753
2 3 132 125 456
3 4 133 126 852

In [2]: bins=[min(df.price)-1,500,max(df.price)+1]
   ...: labels=["价格500以下","价格500以上"]
   ...: pandas.cut(df.price,bins)
Out[2]:
0 (158, 500]
1 (500, 853]
2 (158, 500]
3 (500, 853]
Name: price, dtype: category
Categories (2, interval[int64, right]): [(158, 500] < (500, 853]]

In [3]: pandas.cut(df.price,bins,right=False)
Out[3]:
0 [158, 500)
1 [500, 853)
2 [158, 500)
3 [500, 853)
Name: price, dtype: category
Categories (2, interval[int64, left]): [[158, 500) < [500, 853)]

In [4]: pa=pandas.cut(df.price,bins,right=False,labels=labels)
   ...: pa
Out[4]:
0 价格500以下
1 价格500以上
2 价格500以下
3 价格500以上
Name: price, dtype: category
Categories (2, object): ['价格500以下' < '价格500以上']

In [5]: df['label']=pandas.cut(df.price,bins,right=False,labels=labels)
   ...: df
Out[5]:
```

```
id band num price label
0 1 130 123 159 价格500以下
1 2 131 124 753 价格500以上
2 3 132 125 456 价格500以下
3 4 133 126 852 价格500以上
```

4.4.7 日期处理

1. 日期转换

日期转换是指将字符型的日期格式转换为日期格式数据的过程，其语法结构如下。

```
to_datetime(dateString,format)
```

format 格式如下。

%Y：年份。

%m：月份。

%d：日期。

%H：小时。

%M：分钟。

%S：秒。

【例 4-25】to_datetime(df. 日期时间列，format= '%Y/%m/%d')。

```
In [1]: from pandas import read_csv
   ...: from pandas import to_datetime
   ...: df = read_csv(r'D:\book\rz3.csv',sep=',',encoding='utf8')
   ...: df
Out[1]:
num price year month date
0 123 159 2016 1 2016/6/1
1 124 753 2016 2 2016/6/2
2 125 456 2016 3 2016/6/3
3 126 852 2016 4 2016/6/4
4 127 210 2016 5 2016/6/5
5 115 299 2016 6 2016/6/6
6 102 699 2016 7 2016/6/7
7 201 599 2016 8 2016/6/8
8 154 199 2016 9 2016/6/9
9 142 899 2016 10 2016/6/10

In [2]: df_dt = to_datetime(df.date,format="%Y/%m/%d")
   ...: df_dt
Out[2]:
0 2016-06-01
1 2016-06-02
2 2016-06-03
3 2016-06-04
4 2016-06-05
5 2016-06-06
6 2016-06-07
7 2016-06-08
```

```
8 2016-06-09
9 2016-06-10
Name: date, dtype: datetime64[ns]
```

注意：csv 的格式应为 UTF-8，否则会报错。另外，csv 里 date 格式是文本 (字符串)。

2. 日期格式化

日期格式化是指将日期型的数据按照给定的格式转化为字符型的数据，其语法结构如下。

```
apply(lambda x:处理逻辑)
datetime.strftime(x,format)
```

【例 4-26】日期型数据转化为字符型数据。

```
df_dt = to_datetime(df.注册时间, format= '%Y/%m/%d');
df_dt _str = df_dt.apply(df.注册时间, format= '%Y/%m/%d')。
```

```
In [1]: from pandas import read_csv
   ...: from pandas import to_datetime
   ...: from datetime import datetime
   ...:
   ...: df = read_csv(r'D:\book\rz3.csv',sep=',',encoding='utf8')
   ...: df_dt = to_datetime(df.date,format="%Y/%m/%d")
   ...:
   ...: df_dt_str=df_dt.apply(lambda x: datetime.strftime(x,"%Y/%m/%d"))
   ...: df_dt_str
Out[1]:
0 2016/06/01
1 2016/06/02
2 2016/06/03
3 2016/06/04
4 2016/06/05
5 2016/06/06
6 2016/06/07
7 2016/06/08
8 2016/06/09
9 2016/06/10
Name: date, dtype: object
```

注意：当希望将函数 f 应用到 DataFrame 对象的行或列时，可以使用 apply(f, axis=0, args=(), **kwds) 方法。当 axis=0 时表示按列运算，当 axis=1 时表示按行运算。例如：

```
In [1]: from pandas import DataFrame
   ...: df=DataFrame({'ohio':[1,3,6],'texas':[1,4,5],'california':[2,5,8]},
           index=['a','c','d'])
   ...: df
Out[1]:
ohio texas california
a 1 1 2
c 3 4 5
d 6 5 8

In [2]: f = lambda x:x.max()-x.min()
   ...: df.apply(f) #默认按列运算，同df.apply(f,axis=0)
```

```
Out[2]:
ohio 5
texas 4
california 6
dtype: int64

In [3]: df.apply(f,axis=1) #按行运算
Out[3]:
a 1
c 2
d 3
dtype: int64
```

3. 日期抽取

日期抽取是指从日期格式里面抽取出需要的部分属性，其语法格式如下。

```
Data_dt.dt.property
```

属性取值的相关含义如下。

second：1 ～ 60 秒，从 1 开始到 60。

minute：1 ～ 60 分，从 1 开始到 60。

hour：1 ～ 24 小时，从 1 开始到 24。

day：1 ～ 31 日，一个月中第几天，从 1 开始到 31。

month：1 ～ 12 月，从 1 开始到 12。

year：年份。

weekday：1 ～ 7，一周中的第几天，从 1 开始，最大为 7。

【例 4-27】对日期进行抽取。

```
In [1]: from pandas import read_csv;
   ...: from pandas import to_datetime;
   ...: df = read_csv(r'D:\book\rz3.csv',sep=',',encoding='utf8')
   ...: df
Out[1]:
num price year month date
0 123 159 2016 1 2016/6/1
1 124 753 2016 2 2016/6/2
2 125 456 2016 3 2016/6/3
3 126 852 2016 4 2016/6/4
4 127 210 2016 5 2016/6/5
5 115 299 2016 6 2016/6/6
6 102 699 2016 7 2016/6/7
7 201 599 2016 8 2016/6/8
8 154 199 2016 9 2016/6/9
9 142 899 2016 10 2016/6/10

In [2]: df_dt =to_datetime(df.date,format='%Y/%m/%d')
   ...: df_dt
Out[2]:
0 2016-06-01
1 2016-06-02
2 2016-06-03
```

```
3 2016-06-04
4 2016-06-05
5 2016-06-06
6 2016-06-07
7 2016-06-08
8 2016-06-09
9 2016-06-10
Name: date, dtype: datetime64[ns]

In [3]: df_dt.dt.year
Out[3]:
0 2016
1 2016
2 2016
3 2016
4 2016
5 2016
6 2016
7 2016
8 2016
9 2016
Name: date, dtype: int64

In [4]: df_dt.dt.day
Out[4]:
0 1
1 2
2 3
3 4
4 5
5 6
6 7
7 8
8 9
9 10
Name: date, dtype: int64
```

其他月、星期、秒、时如下：

```
df_dt.dt.month
df_dt.dt.weekday
df_dt.dt.second
df_dt.dt.hour
```

本章小结

本章主要介绍了利用 Numpy 库和 Pandas 库进行数据准备、数据处理等内容；数据的查重、缺失值的处理，以及正则表达式和时间的处理。在数据分析工作中，数据清洗占了很大的比重，如何快速地整理数据是本章的重点。

正则表达式的应用 .mp4

练　　习

班主任现有一个班级的如下两张表。

成绩表

学　号	C#	线　代	Python
16010203	78	88	96
16010210	87	58	83
16010205	84	65	82
16010213	86	72	67
16010215	67	76	85
16010208	76	43	69
16010209	56	68	92
16010204	89	缺考	86
16010211	81	81	75
16010212	73	77	69
16010206	65	80	84
16010214	90	73	91
16010207	91	64	86

信息表

姓　名	学　号	手机号码
张三	16010203	16699995521
李四	16010204	16699995522
王五	16010205	16699995523
赵六	16010206	16699995524
郑七	16010207	16699995525
钱八	16010208	16699995526
张千	16010209	16699995527
赵六	16010210	16699995528
李矛	16010211	16699995529
张白	16010212	16699995510
白九	16010213	16699995511
冀二	16010214	16699995512
余一	16010215	16699995513

请帮班主任做如下工作。

(1) 给成绩表加上姓名列。

(2) 给成绩表加上字段“总分”列，并求出总分。

(3) 计算各门课程的平均成绩以及标准差。