基础篇

第 1 章
Python 语法基础

1.1 引言

随着信息技术的发展，新技术正在不断地改变着各行各业的工作方式，尤其是医疗行业。以人工智能、大数据为代表的新一轮科技革命和产业变革，为传统的医疗模式带来了新的机遇和挑战。2025 年春节期间 DeepSeek 的出现，将人工智能推向了一个新的高度。所以新时代下，医务工作者必须具有掌握和使用新技术的能力，从而更好地为病患提供治疗方案。

随着云计算、物联网、移动互联网等新兴科技的兴起，各行各业积累的数据量呈现指数级增长，“大数据”时代已真正到来，越来越多的企业开始提供大数据解决方案及大数据分析工具，而数据又是一种非常实用的资源，被誉为“未来的能源”。随着社会信息化水平的进一步提升，海量的数据将通过大数据分析工具进行处理，高质量的大数据分析工具和高质量的人工智能大数据解决方案将会成为人们关注的焦点。

本书将带领大家把编程由陌生变成熟悉，由熟悉变成简单，到最后只需要会问问题，即可写出代码，实现真正的零代码编写。但是请注意，零代码并非零基础，基于这一点，我们选择了易学好用的 Python 作为学习医疗大数据处理分析技术的入门语言，主要基于以下几个方面的原因：

1. 易于学习和使用

Python 具有简洁的语法和丰富的库，这使得它成为易于学习和使用的编程语言。对于医疗领域的研究人员和数据分析师来说，能够快速上手并应用 Python 进行数据处理和分析是至关重要的。Python 的易学易用性降低了学习门槛，而且当前流行的 AI 工具也都能与 Python 语言进行很好的融合。

2. 强大的数据处理能力

Python 拥有众多强大的数据处理库，如 Pandas、NumPy 等。这些库在处理大规模数据集时表现出色，能够高效地进行数据清洗、转换、集成等操作。

3. 丰富的数据分析与挖掘工具

Python 在数据挖掘和机器学习领域具有强大的竞争力。通过使用 Scikit-learn、

TensorFlow 等库，Python 可以轻松地完成各种数据挖掘和机器学习任务，如分类、回归、聚类和深度学习等。这些工具在医疗大数据处理分析中具有重要应用，可以帮助研究人员发现数据中的隐藏模式和规律，为医疗决策提供支持。

4. 强大的数据可视化能力

Python 拥有丰富的数据可视化库，如 Matplotlib、Seaborn 等。这些库可以将数据处理结果以直观的方式呈现出来，帮助研究人员更好地理解数据和发现数据中的模式。

5. 广泛的应用场景

Python 在医疗领域的应用场景非常广泛，它不仅可以用于医疗大数据的处理和分析，还可以用于医学图像处理、生物信息学数据分析、药物研发等多个方面。这种广泛的应用场景使得 Python 成为医疗领域不可或缺的工具之一。

综上所述，学习医疗大数据处理分析技术适合用 Python，因为它具有易于学习和使用、强大的数据处理能力、丰富的数据分析与挖掘工具、强大的数据可视化能力、广泛的应用场景及强大的社区支持等优势，这些优势使得 Python 成为医疗领域数据处理和分析的首选编程语言。

1.2　Python 编辑器

在处理和分析数据时，常常需要安装一些软件，这些软件称为编辑器。Python 的编辑器比较多，如 Anaconda、PyCharm 等，当然还有国产 AI 零代码编写工具 Trae。零代码并非零基础，所以尽管 Trae 可以不用逐行编写代码，但读者必须知道代码运行逻辑，有些时候还需要知道如何调试代码，所以在使用 Trae 之前，需要学习 Python 编程的基础知识，并了解相关编辑器的操作。

很多编辑器在处理和分析数据、解决某个问题时，并不需要用户自己去写很多的代码，因为已有 “好事者”写好了诸多方便好用的库和模块，只需要导入相应的库和模块调用即可。Anaconda 就是这样一款编辑软件，它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 900MB。现在 Spyder 和 Jupy 已成为数据分析的标准环境，尤其 Jupy 更是数据分析和交流使用得较多的工具之一。

Anaconda 主要用于数据科学和机器学习领域，它包含了大量常用的数据科学包和库，如 NumPy、Pandas、Scikit-learn 等，并且提供了包管理工具 Conda，方便用户安装、更新和管理这些包，所以本书采用 Anaconda 编辑器。Anaconda 有两种编写代码环境，除了常用的 Spyder，还有方便进行交互式编程和数据分析用的 Jupyter Notebook（以下简称 Jupy）。

Anaconda 官方下载网址为 https://www.anaconda.com。下载时请按照计算机的配置情况下载适配（如 Windows、Mac 或 Linux）版本，如图 1-1 所示。Anaconda 更新较快，为了使用稳定的第三方库，本书使用的是目前 Anaconda 官方 Windows 系统 64 位 Python 3.12 版本，下载后直接双击安装，可自选安装位置。安装完成后，在“开始”菜单中可以看到如图 1-2 所示的目录。注意在自选安装位置时，尽可能地避免使用中文路径，以防止出现不可预见的意外错误。

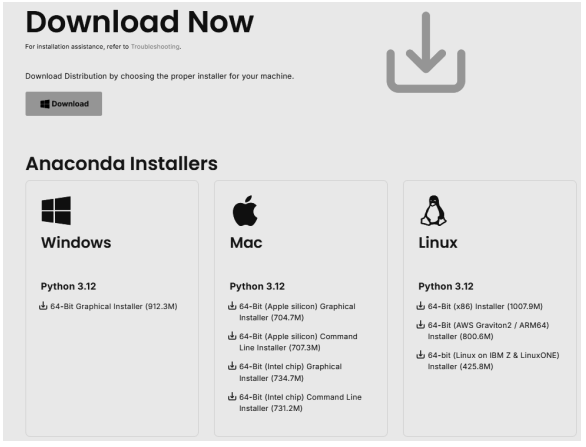


图 1-1　Anaconda 下载界面

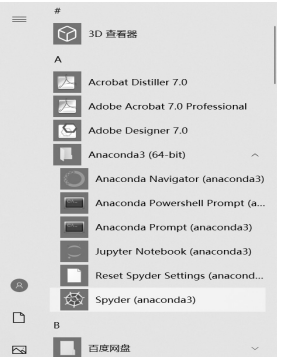


图 1-2　Anaconda 菜单

Anaconda 目录菜单中增加了 Anaconda Prompt 选项，单击它打开命令窗口，输入 conda install 命令，安装第三方库，也可以使用 pip install 命令来安装第三方库。例如，安装 jieba 分词模块的命令为 : **pip install jieba**，如图 1-3 所示。

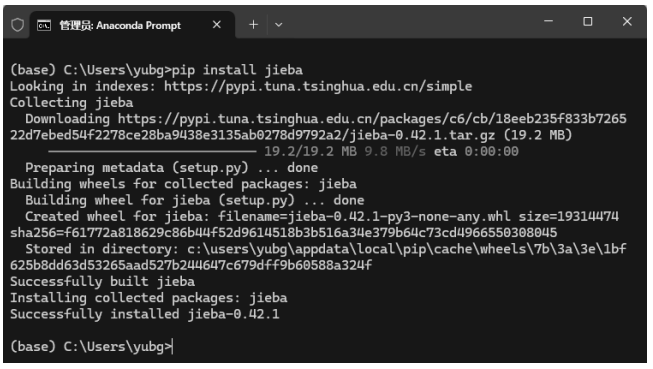


图 1-3　安装第三方库或模块

有时第三方库比较大，下载比较慢，可以增加清华镜像下载安装。

```
pip install jieba -i https://pypi.tuna.tsinghua.edu.cn/simple
```

在图 1-2 中选择 Anaconda Spyder 选项，打开 Spyder，其界面如图 1-4 所示，不同的版本界面略有差异。

在代码编辑区编辑代码，执行代码时选定代码行，按【F9】键或者单击工具栏上的 按钮即可。

Anaconda 是 Python 的一个开源发行版本，主要面向科学计算。本书将采用 Anaconda 下的 Spyder 和 Jupyter Notebook，偶尔会使用 Python 原生编辑器。一般情况下，个人写代码时用 Spyder 比较方便，在进行教学或者演讲交流时，用 Jupy 或许更胜一筹，毕竟它可以在演讲过程中进行代码交互，最后还可以将演讲过程，以及代码和运行结果导出保存为 .html 或 .pdf 格式。

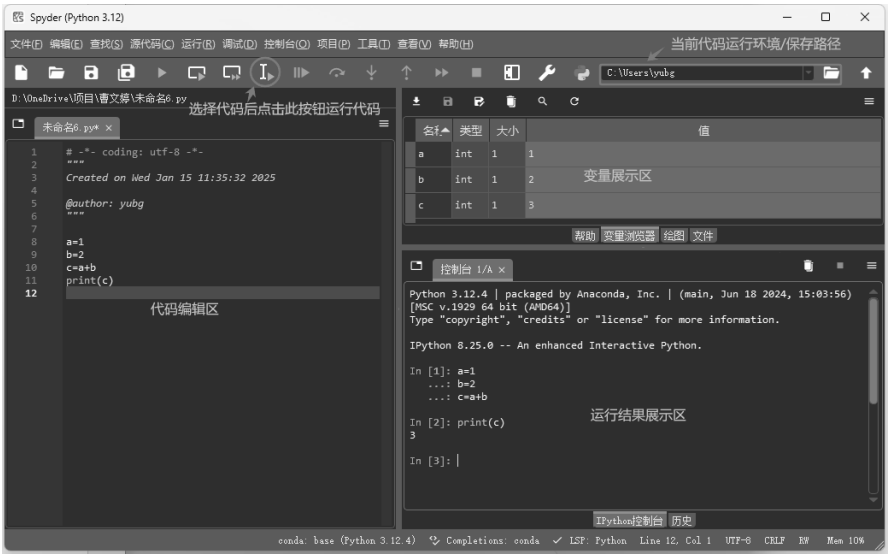


图 1-4　Spyder 界面

1.3　语法规范

在计算机语言中，都有一套代码运行规则。比如在 Python 语言中，一句代码占一行，行尾一般不用什么符号来表示一行结束；再如用“#”表示注释符，意思是在一行中遇到“#”时，该行“#”后的内容都是对该行代码的说明和解释，计算机不会执行“#”后的内容，即注释内容是给程序员看的，不是让机器运行的，这就类似于人们看书时，在书眉上做的批注，不属于书的正式内容。

1. 代码注释方法

（1） 通常在一行中，“#”后的语句不再被执行，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】# 号注释行，三引号注释段落。

```
# -* - coding: utf-8 -*-
"""
Created on Sun Mar 13 21:20:06 2024
@author: yubg
"""
for i in range(5):          # 半角状态冒号不能省略，下一行注意缩进
    print(i)                # 注意上一行末尾有冒号，该行要缩进 4 个空格
```

本例中的双三引号（"""）之间（第 3、4 行）被引用的也是注释，与“#”注释不同的是，三引号可以对段落进行注释，即对多行进行注释，而“#”仅注释一行内容。

Python 代码中所涉及的符号，如括号（()[]{}）、引号（单引号、双引号、三引号）和标

点符号（逗号、分号、冒号）等，都需要在英文半角状态下输入。

（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。单引号和双引号成对使用，单引号和双引号没有本质的差别。

2. 用缩进表示分层

Python 不像其他语言用括号来表示语句块或逻辑层次关系，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】键来表示缩进 4 个空格，但不要混合使用【Tab】键和空格来进行缩进，这会使得程序在跨平台时不能正常运行，官方推荐的做法是使用 4 个空格。

一般来说，行尾遇到冒号“:”，就表示下一行缩进的开始，如【例 1-1】中第 6 行“for i in range(5):”行尾有冒号，下一行的“print(i)”就需要缩进 4 个空格。

3. 语句断行

一般来说，Python 语言中一条语句占一行，在每条语句的结尾处不需要使用分号“;”。但在 Python 中也可以使用分号，表示将两条简单的语句写在一行。但如果一条语句较长需要分几行来写，可以用“\”来进行续行，即续行符，注意在续行符“\”之后不能出现其他任何字符，包括空格。

```
In [1]: s = "生如蝼蚁，当有鸿鹄之志。命如纸薄，应有不屈之心。\
   ...: 大丈夫生于天地间，岂能郁郁久居人下。当以梦为马，不负韶华。\
   ...: 乾坤未定，你我皆是黑马！"
   ...: print(s)
生如蝼蚁，当有鸿鹄之志。命如纸薄，应有不屈之心。 大丈夫生于天地间，岂能郁郁久居人下。当以梦为马，不负韶华。 乾坤未定，你我皆是黑马！
```

上面的字符串变量 s 本来是一行写完的，但是为了便于浏览，采用了分行来写，使用了续行符“\”，效果等同于一行写完。

一般来说，在一对括号中间或三引号之间断行时，系统均能自动识别断行。括号内（包括圆括号、方括号和花括号）断行后的第二行一般空 4 个空格，但有时为了层次清晰，一般采用“逻辑”对齐，在 Spyder 下会自动对齐。在括号内断行时不需要再添加续行符。

```
1   import pandas as pd
2   import os
3   path = os.path.join(os.path.expanduser("~"),"desktop") #桌面路径
4   fname = r"d:\OneDrive\统计信息中心\第七次调查\第七次卫生\
5           调查准备\code\全部村镇.xlsx"  #   全部村镇.xlsx
6   data = pd.read_excel(fname)
7   print(data.iloc[0:30],
8         "\n%s共有镇数："%(os.path.splitext(fname)[0]),
9         len(data))
10  print(data.iloc[30:],
11        "\n%s共有镇数："%(os.path.splitext(fname)[0]),
12        len(data))
```

上面代码中第 7 行、第 10 行的 print 代码行比较长，在 print 后的 () 内可以换行，分成两行或者多行来写，其下一行会自动与 data 逻辑对齐。print 代码行及其下两行等价以下两行代码：

```
print(data.iloc[0:30],"\n%s 共有镇数："%(os.path.splitext(fname)[0]),len(data))
print(data.iloc[30:],"\n%s 共有镇数："%(os.path.splitext(fname)[0]),len(data))
```

4. print() 函数的作用

print() 函数运行后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。

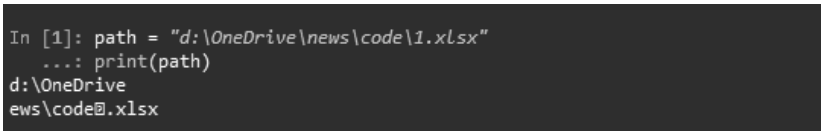
print() 是一个常用函数，其功能就是输出括号中的字符串或变量的值。print() 可以有多

个输出，以逗号分隔。如上面第 7 行、第 10 行代码中的 print() 输出。

当在循环输出中要将多个结果打印在一行并以逗号分隔时，可以在 print() 中添加参数 end=',' ，如【例 1-1】中的 print 行可以改为 print(i, end=';')。

5. 特殊符号的输出

反斜线“\”在行尾时表示续行符，在特殊字符组合中表示转义符。如“\n”的组合表示回车换行，“\t”表示 tab 键制表符。例如：



在上面的代码中，path 想表示的是一个完整的文件路径，但其中包含“\n”和“\1”，分别表示换行和特殊符号，所以在输出时就不再是完整的路径，而是发生了转义。所以路径中的反斜线“\”需要处理，在“\”前面需要再加一个转义符，即“\\”才表示输出一个“\”符号，如下：

```
In [2]: path = "d:\\OneDrive\\news\\code\\1.xlsx"
  ...: print(path)
d:\OneDrive\news\code\1.xlsx
```

当然也可以不用这方式，直接在路径字符串之前加一个“r”或“R”即可，它表示该字符串不要转义，而是按原样输出。

```
In [3]: path = r"d:\OneDrive\news\code\1.xlsx"
  ...: print(path)
d:\OneDrive\news\code\1.xlsx
```

6. 变量

变量类似于数学中的变量的概念，可以给它赋值，如 a = 3。变量的名称只能由数字、字母和下画线构成，数字不能用在开头，字母区分大小写，以下画线开头的变量有特殊含义。变量名不能含有空格和其他标点符号，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，这些需要命名的对象称为标识符。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，变量或函数名的名称一般使用小写的字母或单词，类的名称一般首字母要大写。

1.4　数据类型

Python 是一种高级编程语言，它支持多种数据类型，这些数据类型使得 Python 非常灵活和强大。Python 的数据类型大致可以分为下列几大类：数值类型、字符串类型（String）、列表（List）、元组（Tuple）、集合（Set）、字典（Dictionary）及布尔类型（Boolean）。

1. 数值类型

整型（int）：用于表示整数，如 1、2、100、-3 等。

浮点型（float）：用于表示浮点数，即带小数点的数字，如 3.14、1.0、-1.2 等。

2. 字符串类型（String）

用于表示文本数据，由字符序列组成，可以是字母、数字、空格、标点符号等。字符串在 Python 中用引号（单引号、双引号、单三引号或双三引号）引起来，如 "hello"、'world'、""" 你好 """。

字符串中的每个字符排列是有顺序的，从左向右，这个顺序称为索引（index）。Python 的索引是从 0 开始的自然数，如字符串“python”中的字母 p 的序号是 0，即索引为 0；字母 n 的序号为 5，即索引为 5。有时为了方便，也可以使用逆序（从右向左，即 -1，-2，…），所以字符串的最后一位索引可以使用 -1 表示。

按照索引可以提取字符串中的字符，在变量或字符串后带上方括号，括号内写索引。如提取字符串 s = “python” 中的字符 p，可以使用 s[0] 表示；提取字符 n，可以使用 s[5] 表示。由于 n 是字符串的最后一位，所以也可以使用 s[-1] 来提取。

```
In [1]: s = "python"

In [2]: print(s[0])
p

In [3]: print(s[5])
n

In [4]: print(s[-1])
n
```

提取字符串中的一个片段称为切片，提取切片需要用到开始和结束的位置索引，但是要注意，Python 的切片范围是取不到右侧值的，即切片 [s1：s2] 类似于数学上的取整区间 [s1：s2)，是左闭右开的，即取不到 s2 位置，如果想取到 s2 位置，则切片需要向后移动一个位置，即 [s1,s2 + 1]。

```
In [5]: print(s[1:4]) # 提取索引号为 1、2、3 位置上字符，所以提取不到 o 字母
yth
```

3. 列表（List）

列表是 Python 中最常用的数据结构之一，它是一个有序的数据集合，可以包含不同类型的元素，并且支持索引和切片操作，也支持添加、删除等操作。列表用方括号 [] 表示，如 L = [1, 2, 'a', 'b']。

向列表中增加元素，一般使用 append() 函数，默认追加在列表的末尾，如向列表 L 追加一个元素 0，可以写作：L.append(0)。

```
In [6]: L = [1, 2, 'a', 'b']

In [7]: L.append(0)

In [8]: L                    # 在 Spyder 或 Jupy 中可直接输出变量，等同于 print() 功能
```

```
Out[8]: [1, 2, 'a', 'b', 0]
```

要删除一个元素，可以使用 pop(index) 函数，参数 index 是指拟删除的元素的索引号，默认删除最后一个元素。

```
In [9]: L.pop()                          # 删除 L 最后一个元素，并将被删除的元素抛出（输出）
Out[9]: 0

In [10]: L
Out[10]: [1, 2, 'a', 'b']
```

4. 元组（Tuple）

元组与列表类似，与列表不同的是，元组一旦创建就不能被修改（即不可变），且用圆括号 () 表示，如 T = (1, 2, 'a', 'b')。元组没有删除和修改功能。

列表和元组之间可以相互转化，例如，将元组 T 转化为列表：list(T)；将列表 L 转化为元组：tuple(L)。

```
In [11]: T = (1, 2, 'a', 'b')

In [12]: list(T)
Out[12]: [1, 2, 'a', 'b']

In [13]: tuple(L)
Out[13]: (1, 2, 'a', 'b', 0)
```

列表和元组中元素的索引用法都与字符串的使用方法一致。

```
In [14]: print(L[:2])                    # 从 0 开始的切片 0 可以省略
[1, 2]

In [15]: print(T[2:])                    # 切片至最后一个字符，最后一个字符的索引可以省略
('a', 'b')
```

5. 集合（Set）

集合是一个无序的、不包含重复元素的数据集合。集合主要用于数学上的集合运算，如并集、交集、差集等。集合用花括号 {} 表示（注意，空集合不能用 {} 表示，因为这会与字典的语法冲突，应该使用 set() 来创建空集合），如集合 a = {1, 2, 3}, 集合 b = {0,2,3}。求集合的并集用符号“a|b”，求交集用“a&b”符号。集合的一个重要功能就是过滤重复值。

```
In [16]: a = {1, 2, 3}
   ...: b = {0, 2, 3}

In [17]: a & b                           # 求交集
Out[17]: {2, 3}

In [18]: a | b                           # 求并集
Out[18]: {0, 1, 2, 3}

In [19]: set([1,2,3,1,2])                # 过滤列表 [1,2,3,1,2] 中的重复值
Out[19]: {1, 2, 3}
```

6. 字典（Dictionary）

字典是 Python 中另一个非常重要的数据结构，它用于存储键（key）值（value）对。比

如手机存储电话号码，既要存储姓名，还要存储号码，这就需要存储类似于“姓名：号码”这样的一个冒号对，冒号前面的部分称为键（key），后面的部分称为值（value），所以叫作键值对。

字典是可变且无序的。字典也用花括号 {} 表示，其每个元素就是一个键值对，元素之间用逗号分隔，如 d={'name': 'Alice', 'age': 25} 有两个元素，分别是 'name': 'Alice' 和 'age': 25。

字典若增加一个元素（即键值对）或者修改一个键值对的值，都是直接给键名赋值，如给字典 d 增加一个元素“'sex': 'female'”, 则可写成 d['sex']= 'female'。若对已有的键值对修改键值，则直接给键名赋值即可，如把 'age' 的值修改为 30，则为 d['age']= 30。

```
In [20]: d={'name': 'Alice', 'age': 25}
  ...: d
Out[21]: {'name': 'Alice', 'age': 25}

In [22]: d["sex"] = "female"            # 增加一个元素
  ...: d
Out[23]: {'name': 'Alice', 'age': 25, 'sex': 'female'}

In [24]: d["age"] = 30                  # 修改一个键的值
  ...: d
Out[24]: {'name': 'Alice', 'age': 30, 'sex': 'female'}
```

7. 布尔类型（Boolean）

布尔类型只有两个值：True 和 False，分别用于表示真（1）和假（0）。布尔类型常用于条件判断、循环控制等场景。

上面介绍的几种类型除了数值型和布尔型，都可以使用 len() 函数查看变量的长度，即所包含的字符串、元素的个数，并且变量的数据类型都可以使用 type() 查看。

```
In [25]: type(d)
Out[25]: dict
```

Python 的这些数据类型提供了丰富的数据结构，使得 Python 在处理各种复杂的数据时非常灵活和强大，各种类型的其他用法在后续章节中将陆续介绍。

1.5 运算符

在 Python 语言中，符号“=”表示赋值，而双等于号“==”表示判断其两侧的值是否相等。

运算符加、减、乘、除、乘方的符号分别为：+、−、*、/、**。

```
比较符号为：大于     >
            小于     <
            ==       相等
            !=       不等于
//      取商的整数部分
%       取余数
```

```
In [1]: 2 ** 3            # 2 的三次方
Out[1]: 8

In [2]: 8 // 3            # 取商
Out[2]: 2

In [3]: 3 == 2            # 判断 3 和 2 是否相等
Out[3]: False

In [4]: 3 != 2            # 3 不等于 2
Out[4]: True
```

1.6　流程控制

Python 语言的流程控制是编程中非常基础且重要的概念，它允许程序根据条件或循环执行不同的代码块。Python 中的流程控制主要包括 3 种类型：顺序结构、选择结构（条件判断）和循环结构。

顺序结构是最简单的程序结构，程序按照代码的顺序从上到下依次执行。一般代码都是按照顺序结构逐行执行的，但是遇到选择结构（也称条件判断）和循环结构时，将按照条件来执行。

1.6.1　选择结构

选择结构允许程序根据条件表达式的真假来决定执行哪一部分的代码。在 Python 中，如果是二分支，则使用 if/else 结构即可；如果是多分支，则在 if 和 else 之间增加 elif 语句来实现多分支选择结构，elif 按照分支的需要而增加。

例如，下面的代码实现了新冠病毒检测分类，采用荧光定量 PCR 的界限值通常为 40，即 Ct 值超过 40 则判定为阴性，反之则为阳性。

```
In [1]: x = 37
  ...: if x > 40:
  ...:     print("测试结果为阴性。")
  ...: else:
  ...:     print("测试结果为阳性。")
测试结果为阳性。
```

乳腺癌的评估是很重要的，其预后有 4 种结果，可采用 if/elif/else 结构，除了 if 和 else，还有两种结果，所以需要增加两个 elif 语句。

```
In [2]: x = "HER-2 阳性"
  ...: y = ["Luminal A 型","HER-2 阳性","Luminal B 型","三阴"]
  ...: if x == y[0]:
  ...:     print(f"结果为：{y[0]}")
  ...: elif x == y[1]:
```

```
  ...:     print(f"结果为：{y[1]}")
  ...: elif x == y[2]:
  ...:     print(f"结果为：{y[2]}")
  ...: else:
  ...:     print(f"结果为：{y[3]}")
结果为：HER-2阳性
```

print() 函数代码内的语句 f"结果为：{y[1]}" 为格式化输出的一种表达方式，称为 f 格式输出，其中 {} 表示占位符，表示此处显示花括号内的变量值，即此处将显示 y[1] 的值。

1.6.2 循环结构

循环结构允许程序重复执行某段代码，直到满足特定的条件为止。Python 中主要有两种循环结构：for 循环和 while 循环。

1. for 循环

for 循环用于遍历任何序列（如列表、元组、字典、集合或字符串）或其他可迭代对象，即每次从序列中取出一个元素，并执行一次其下方的代码块，直到所有的元素都被取完。

下面的例子是将元组 fruits 中的元素挨个添加到空列表 fr 中。

```
In [3]: fruits = ("apple", "banana", "cherry")
  ...: fr = []
  ...: for i in fruits:
  ...:     fr.append(i)          # 将取出来的元素添加到 fr 中
  ...: print(fr)

['apple', 'banana', 'cherry']
```

2. while 循环

while 循环在给定条件为真时重复执行其下方的代码块，直到条件为假则终止执行其下的代码块。所以一般 while 循环都需要添加控制语句，以满足终止条件，否则将形成“死循环”。

下面的例子是先给出一个数 n，再循环计算从 1 加到 n 的总和，最后输出该总和。

```
In [4]: n = 5
  ...:                      # 使用 while 循环计算从 1 加到 n 的总和
  ...: i = 1
  ...: sum = 0
  ...: while i <= n:
  ...:     sum += i
  ...:     i += 1           # 控制语句，以满足终止条件
  ...:                      # 输出结果
  ...: print(f"从 1 加到 {n} 的总和是：{sum}")

从 1 加到 5 的总和是：15
```

在此例子中，当 i 的值小于 n 的值 5 的时候，即运行其下方的两行代码，其中 sun += i 等同于 sun = sum + i，即每执行一次就将 sum 和 i 的值相加；下面的 i += 1 等同于 i = i + 1，即每执行一次就自动增加 1，该行是条件终止控制语句。

i += 1 比 i = i + 1 要便于计算机内部计算，所以在循环代码中自身加 1 时一般采用 i += 1，当自身减 1 时用 i −= 1，同理，自身乘以 j 时用 i *= j。

1.7　常用函数

在 Python 程序代码中，除了 len()、print()、type() 等常见的最基本的函数，还有几个常用的函数，如 range、map、zip 及自定义函数等。

1. range() 函数

range() 函数是产生一个整数序列的函数，如 1，2，3，4…。该函数有 3 个参数 start、stop、step，即起止位置及步长，range(start,stop,step)。例如 range(3,7,1) 会产生一个包含 3、4、5、6 共 4 个元素的序列，也就是说 range() 函数和字符串的切片有相似之处，都是左闭右开，即右边的 stop 是取不到的。当 step 是 1 时可以省略不写，所以 range(3,7,1) 也可以写作 range(3,7)。

```
In [1]: a = range(3,7,1)

In [2]: print(a)
range(3, 7)

In [3]: list(a)
Out[3]: [3, 4, 5, 6]

In [4]: list(range(3))
Out[4]: [0, 1, 2]
```

从上面的 In[2] 代码行可以看出，range(3,7,1) 和 range(3,7) 是一致的，而且 range() 函数的结果是一个容器，不是一个列表，也不是一个元组，但可以使用 list() 或 tuple() 函数去调用它。

```
In [5]: tuple(a)
Out[5]: (3, 4, 5, 6)
```

range() 函数常与 for 循环一起使用。

2. map() 函数

map() 函数是一个内置函数，用于将指定的函数应用于可迭代对象中的每个元素，并返回一个迭代器。

```
map(func, iter)
```

func：应用到每一个元素上的函数。

Iter： 一个序列（如列表、元组等）。

例如将列表 [1,2.0,3.0,4,0] 中的每个元素转为整型 int。

```
In [6]: b = [1,2.0,3.0,4,0]

In [7]: m = map(int, b)   # 是 int，不是 int()

In [8]: print(m)
<map object at 0x0000021B19DFC250>

In [9]: list(m)           # 可用 list() 或 tuple() 调用
Out[9]: [1, 2, 3, 4, 0]
```

3. zip() 函数

zip() 函数也是一个内置函数，它用于将多个可迭代对象中的对应元素打包在一起，返回一个迭代器。这个迭代器生成的元组包含从每个传递的可迭代对象中获取的元素。

例如，可以将列表 ["a","b","c"] 和 [1,2,3,4] 对应组合起来做成列表或元组，如 [('a', 1), ('b', 2), ('c', 3)]。

```
In [10]: c = ["a","b","c"]

In [11]: d = [1,2,3,4]

In [12]: z = zip(c,d)

In [13]: print(z)
<zip object at 0x0000021B19A97880>

In [14]: list(z)          # 用 list() 或者 tuple() 函数调用
Out[14]: [('a', 1), ('b', 2), ('c', 3)]
```

有时候需要将打包的两个序列解开，也就是 [('a', 1), ('b', 2), ('c', 3)] 解开为 ["a","b","c"] 和 [1,2,3,4]，只需将打包的结果放在 zip(*) 中的星号之后即可。

```
In [15]: list(zip(*[('a', 1), ('b', 2), ('c', 3)]))
Out[15]: [('a', 'b', 'c'), (1, 2, 3)]

In [16]: cc , dd = _      # _ 表示上一步的结果

In [17]: cc
Out[17]: ('a', 'b', 'c')

In [18]: dd
Out[18]: (1, 2, 3)
```

In[15] 代码行是 zip() 打包的解包过程。

4. def 自定义函数

def 是用于自定义函数，其格式如下：

```
def name(par):
    block
    return
```

其中的 def 类似于声明，这里是在进行自定义函数，name 是要定义的函数的名称，par 是函数中需要赋值（传递值）的参数，block 是一个代码块，也就是要实现该函数功能的代码，return 是返回该函数最后需要的结果。比如，现在要自定义一个函数名为 cj 的函数，它的功能是将传递进去的数值进行平方后减 1，并将结果返回。运行 cj(5) 后返回结果为 24。

```
In [19]: def cj(x):
  ...:     c = x * x - 1
  ...:     return c
In [20]: cj(5)
Out[20]: 24
```

自定义函数在使用时，必须先运行该函数。

5. lambda 匿名函数

lambda 函数用于需要处理一个小功能的地方，而且这个功能只使用一次，所以不需要定义一个完整的函数，故不必为它命名。格式如下：

```
lambda arg: expression
```

这里 arg 是传入 lambda 函数的参数，而 expression 是功能表 $x^2+y^2x^2+y^2$。例如定义一个 x^2+y^2：

```
In [21]: f = lambda x,y: x**2 + y**2

In [22]: f(2,3)
Out[22]: 13
```

如果改用 def 来自定义函数 f，则如下：

```
In [28]: def f(x,y):
  ...:     c = x**2 + y**2
  ...:     return c
In [29]: f(2,3)
Out[29]: 13
```

通过对比匿名函数和自定义函数可以发现，对于小功能、使用一次的函数，使用匿名函数要简单得多。

在 Python 中，有很多常用的功能都已经被各领域的学者写成了函数，有的“工程”较大的函数则被写成了模块、包或库，那么，这些已经写好的现成函数、模块、包或库该怎么使用呢？

首先，对于已有的第三方的函数、模块、包和库（为了方便，本书统一称之为库），需要下载安装或者在线安装，在线安装用 pip install XXX 方式。安装完成后在使用时，需要先导入该库，如使用 math 模块时，需要在使用前先执行 import math。

```
In [1]: a = [1.2e+18, 1, -1.2e+18]
  ...: sum(a)
Out[1]: 0.0
```

在这个 sum() 求和函数中，结果本应该为 1，但是由于计算机浮点数的问题，导致精确度不高，使得其结果为 0，这在金融或医学中是非常致命的问题。所以我们的第三方开放库 math 已经解决了这个问题，在 math 这个文件中，就已经包含了类似于 sum() 函数功能的 fsum() 函数。

```
In [2]: import math

In [3]: math.fsum(a)
Out[3]: 1.0
```

这里首先导入了 math 文件 import math，在这个文件下有一个 fsum() 函数，所以在调用时就写成 math.fsum()。当然，有的文件名和函数名称很长，多的达到 20 多个字母，所以每次调用时很不方便，这时可以给该文件名或者函数名用 as 取个别名。

```
In [4]: import math as m
  ...: m.fsum(a)
Out[4]: 1.0
```

但是每次都要写一个“m.”有时候也很不方便，所以就有了 from 方式，表示从 math 中导入 fsum 函数，这样就不用每次都写“math.”了。

```
In [5]: from math import fsum
  ...: fsum(a)
Out[5]: 1.0
```

如果要使用该库下较多的函数，也可以一次性导入——直接使用 * 代替全体函数，如 from math import *，即从 math 中导入其下的所有函数。但是不建议这样使用，避免引起一些函数之间的冲突，从而导致程序无法正常运行。

```
In [6]: from math import *
  ...: fsum(a)
Out[6]: 1.0
```

为了更好地掌握 python 基础知识，尝试理解下面几个代码程序。

程序 1：定义一个整数变量和一个浮点数变量，然后将它们相加并输出结果。

```
# 代码框架：
integer_var = 10
float_var = 5.5
result = integer_var + float_var
print(f"结果是：{result}")
```

程序 2：编写一个程序，让用户输入一个数字，判断这个数字是正数、负数还是零，并给出相应的提示。

```
# 代码框架：
number = float(input("请输入一个数字："))
if number > 0:
     print("这是一个正数。")
elif number < 0:
     print("这是一个负数。")
else:
     print("这是零。")
```

程序 3：编写一个程序，打印从 1 到 10 的所有整数，但是当数字为 5 时，输出“Lucky Number!”。

```
# 代码框架：
for i in range(1, 11):
     if i == 5:
          print("Lucky Number!")
     else:
          print(i)
```

程序 4：定义一个函数，接收一个字符串，返回该字符串的反转字符串。

```
# 代码框架：
def reverse_string(s):
     return s[::-1]

original_string = "Hello, World!"
reversed_string = reverse_string(original_string)
print(reversed_string)
```

第 2 章
NumPy 和 Pandas

NumPy 和 Pandas 是 Python 中非常重要的两个库，特别是在数据科学领域。尽管 NumPy 和 Pandas 都有各自的特点和适用场景，但它们通常是协同工作的。通常情况下，如果只需要进行简单的数学运算或者处理数值型数组，NumPy 可能是更好的选择。然而，当涉及复杂的数据处理、分析和转换任务时，Pandas 提供的功能就显得更为强大。实际上，在许多数据科学项目中，NumPy 和 Pandas 都会被一起使用，NumPy 用来处理数值计算，而 Pandas 用来管理和分析数据。Pandas 的数据结构可以很容易地转换成 NumPy 数组，以便利用 NumPy 提供的高效计算能力。

在使用 NumPy 和 Pandas 时，一般先导入，并分别取别名为 np 和 pd，如下：

```
import numpy as np
import pandas as pd
```

NumPy 和 Pandas 的别名为 np 和 pd 是约定俗成的，当然也可以用其他的字母或单词来表示别名，但不建议这么做。

2.1 NumPy

NumPy 是一个开源的 Python 科学计算库，它提供了大量的功能来处理向量、矩阵及 n 维数组（ndarray）。NumPy 的设计是为了处理大量的数值数据，因此它非常适合进行数学和逻辑运算，如矩阵乘法、傅里叶变换、统计运算等。

数组与 Python 内置的列表 List 相比，NumPy 数组的计算速度更快，占用内存更少，非常适合处理大量的数据。

2.1.1 数组的创建

数组的创建方法比较随意，可以是列表、元组，也可以直接用函数生成。

用一个列表或元组通过 array 函数创建一维数组的代码如下：

```
In [1]: import numpy as np

In [2]: lis = [1, 2, 3, 4, 5]
   ...: arr1 = np.array(lis)
   ...: arr1
Out[2]: array([1, 2, 3, 4, 5])

In [3]: tup = tuple(lis)
   ...: np.array(tup)
Out[3]: array([1, 2, 3, 4, 5])
```

同理，二元的列表和元组可以创建二维数组。

```
In [4]: lis2 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
   ...: arr2 = np.array(lis2)
   ...: arr2
Out[4]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

也可以用数组函数创建数组，如使用 numpy.zeros() 函数创建一个全零数组。

```
In [5]: arr3 = np.zeros((3, 3))    # 全零数组
   ...: arr3
Out[5]:
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])

In [6]: arr4 = np.full((3,3),2)    # 创建一个 3×3 的全为 2 的数组
   ...: arr4
Out[6]:
array([[2, 2, 2],
       [2, 2, 2],
       [2, 2, 2]])
```

在 NumPy 中有多种创建 ndarray 数组的函数，常用的几个函数如表 2-1 所示。

数组的大小、形状可以使用 shape 来查看，返回的是数组的维度大小，即一个表示各维度长度的元组。数组也可以使用 len()，返回第一维度的长度，即行数。要想查看数组共有多少个数值，可使用 size。注意这里的 shape 和 size 都不需要带括号。

```
In [7]: arr4.shape
Out[7]: (3, 3)
```

表明 arr4 是 3 行 3 列的数组，有 9 个数值。

```
In [8]: len(arr4)
Out[8]: 3

In [9]: arr4.size
Out[9]: 9
```

表 2-1　创建数组的常用函数

函　　数	功　　能	参 数 说 明
np.array(object,dtype)	从列表或元组创建数据	object: 列表或元组 dtype: 数据类型（可选项）
np.arange(start,stop,step)	创建一个一维数组	start: 起始值，可选项，默认为 0 stop: 终止值（不包含） step: 步长，可选项，默认为 1
np.random.rand(shape)	随机产生一个元素值为 [0,1) 的随机数的数组	shape: 数组形状
np.random.randn(shape)	随机产生一个元素值服从正态分布的随机数的数组	shape: 数组形状
np.random.randint(start,stop,shape)	随机产生一个元素值离散均匀分布的整数数组	start: 起始值（包含） stop: 终止值（不包含） shape: 数组形状
np.random.uniform(start,stop,shape)	随机产生一个元素值服从均匀分布的浮点数数组	start: 起始值（包含） stop: 终止值（不包含） shape: 数组形状
np.ones(shape,dtype)	创建全 1 数组	shape: 数组形状 dtype: 数据类型（可选项）
np.zeros(shape,dtype)	创建全 0 数组	shape: 数组形状 dtype: 数据类型（可选项）
np.full(shape,val)	创建全 val 的数组	shape: 数组形状 val: 数组元素值
np.eye(shape)	创建对角线元素值为 1 的单位矩阵	shape: 数组形状
np.linspace(start,stop, n)	创建一个一维等差数列数组	start: 序列的起始值 stop: 序列的终止值 n: 数组元素个数
np.logspace(start, stop, n)	创建一个一维等比数列数组	start: 序列的起始值 stop: 序列的终止值 n: 数组元素个数

表 2-1 中表示数组形状的 shape 参数若只有 1 个数值，则为一维数组；若给出 2 个数值，则为二维数组，如 (3,4)，表示 3 行 4 列的数组；若给出 3 个数值，则为三维数组，以此类推。

函数 np.arange(start,stop,step) 类似于 range() 函数，只是 np.arange() 的步长可以是小数，如 0.5、1.1 等。

```
In [10]: aa = np.arange(1,11,0.2) # 在 1 到 11 之间生成一个步长为 0.2 的数组

In [11]: aa
```

```
Out[11]:
array([ 1. , 1.2, 1.4, 1.6, 1.8, 2. , 2.2, 2.4, 2.6, 2.8, 3. ,
3.2, 3.4, 3.6, 3.8, 4. , 4.2, 4.4, 4.6, 4.8, 5. , 5.2,
5.4, 5.6, 5.8, 6. , 6.2, 6.4, 6.6, 6.8, 7. , 7.2, 7.4,
7.6, 7.8, 8. , 8.2, 8.4, 8.6, 8.8, 9. , 9.2, 9.4, 9.6,
9.8, 10. , 10.2, 10.4, 10.6, 10.8])
```

np.random 可按要求生成随机数组，如产生一个从 0 到 100 之间的 2×3 的数组，或者产生一个 0 到 1 之间服从正态分布的 10 个浮点数，可以使用 np.random.rand()、np.random.randn()、np.random.randint() 创建随机数组，它们之间的区别如下：

np.random.rand()：用于生成指定形状的 [0, 1) 的均匀分布的随机数。它接收多个参数来指定返回数组的形状，每个参数对应于生成数组的一个维度。例如，np.random.rand(3, 2) 将生成一个形状为 (3, 2) 的二维数组。

np.random.randn()：用于生成指定形状的标准正态分布（平均值为 0，标准差为 1）的随机数。与 np.random.rand() 类似，它也接收多个参数来指定返回数组的形状。例如，np.random.randn(3, 2) 将生成一个形状为 (3, 2) 的二维数组。

np.random.randint()：用于生成指定范围内的随机整数，默认为 int。它接收 3 个参数，分别为最小值、最大值和返回数组的形状。例如，np.random.randint(0, 10, (3, 2)) 将生成一个形状为 (3, 2) 的二维数组，其中的元素是 0 到 9（不包括 10）之间的随机整数。

```
In [12]: np.random.rand()        # 随机产生 [0, 1) 的 1 个均匀分布的随机数
Out[12]: 0.9636627605010293

In [13]: np.random.rand(2)       # 随机产生 [0, 1) 的 2 个均匀分布的随机数
Out[13]: array([0.38344152, 0.79172504])

In [14]: np.random.rand(2,3)     # 随机产生 [0, 1) 的 2×3 形状均匀分布的随机数
Out[14]:
array([[0.52889492, 0.56804456, 0.92559664],
       [0.07103606, 0.0871293 , 0.0202184 ]])
```

np.random.rand() 用于生成 [0, 1) 的均匀分布的随机数。

```
In [15]: np.random.randn()
Out[15]: 0.48431215412066475

In [16]: np.random.randn(3)
Out[16]: array([ 0.57914048, -0.18158257,  1.41020463])

In [17]: np.random.randn(3,2)
Out[17]:
array([[-0.37447169,  0.27519832],
       [-0.96075461,  0.37692697],
       [ 0.03343893,  0.68056724]])
```

np.random.randn() 用于生成标准正态分布的随机数。

```
In [18]: np.random.randint(0, 10, (3, 2))
Out[18]:
array([[0, 0],
       [4, 5],
       [5, 6]])
```

np.random.randint() 用于生成指定范围内的随机整数。其参数可以是 1 个或 2 个。

np.random.randint(10) 生成一个 0 到 10 之间（不含 10）的随机整数。

np.random.randint(5, 10) 生成一个 5 到 10 之间（不含 10）的随机整数。

```
In [19]: np.random.randint(10)          # 生成一个 0 到 9 之间的随机整数
Out[19]: 8

In [20]: np.random.randint(0, 10, 5)
Out[20]: array([4, 1, 4, 9, 8])
```

需要注意的是，这些函数生成的随机数都具有随机性，但也可以通过设置种子数来保证可重复性，即每次运行代码时生成的随机数都是相同的，如 np.random.seed(0)。

```
In [21]: np.random.seed(0)
    ...: np.random.rand()
Out[21]: 0.5488135039273248

In [22]: np.random.seed(0)
    ...: np.random.rand()
Out[22]: 0.5488135039273248
```

上面运行两次的随机种子数均为 0，其结果也均一致。所以从某种意义上说，用这种方式生成的是伪随机数，因为真正的随机数是不可重复的。

2.1.2　数组的操作

创建好数组后，可根据需求改变数组的基础形态，如改变形状、转置、展平、数组的元素类型转换等操作。表 2-2 列出了常用的数组变换方法。

表 2-2　常用的数组变换方法

函　数	说　明
ndarray.reshape(m,n)	将原数组变为 m 行 n 列，操作不影响原数组
ndarray.resize(m,n)	直接将原数组形状修改为 m 行 n 列
ndarray.flatten()	不改变原数组，返回原数组展平成一维数组的副本
ndarray.transpose()	数组转置，将数组的行变成列；也可以用 .T 来完成转置

使用 reshape() 方法改变数组形状，不会修改原始数组，而是生成了一个新的数组，而使用 resize() 方法的效果与 reshape() 相同，但会直接在原始数组上进行操作。对于 reshape() 和 resize() 方法，若将参数 m 和 n 其中的一个设置为 -1，则表示数组的维度通过数组元素的个数自动计算。

```
In [1]: import numpy as np
   ...: narr1 = np.arange(12)
   ...: narr1
Out[1]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

```
In [2]: narr1.reshape(3,4)                # 将一维 narr1 修改为 3 行 4 列的数组
Out[2]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [3]: narr1                             # narr1 本身没有被改变
Out[3]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

In [4]: narr1.reshape(6,-1)               # 自动计算列数
Out[4]:
array([[ 0,  1],
       [ 2,  3],
       [ 4,  5],
       [ 6,  7],
       [ 8,  9],
       [10, 11]])
```

虽然数组要求所有元素的数据类型必须相同，但在需要时也可以通过 astype() 方法对数组中元素的数据类型进行转换。需要注意的是，如果将浮点数转换为整数，则小数部分会被截断。

例如，创建一个元素全为字符串的数组，先将其转换浮点数，再转换为整数。

```
In [5]: narr2 = np.full((3,3),"1.1")
   ...: narr2
Out[5]:
array([['1.1', '1.1', '1.1'],
       ['1.1', '1.1', '1.1'],
       ['1.1', '1.1', '1.1']], dtype='<U3')

In [6]: nar = narr2.astype(float)         # 将数组 narr2 的数据类型转换为 float
   ...: nar
Out[6]:
array([[1.1, 1.1, 1.1],
       [1.1, 1.1, 1.1],
       [1.1, 1.1, 1.1]])

In [7]: nar.astype(int)                   # 将数组 nar 的数据类型转换为整数
Out[7]:
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]])
```

在 NumPy 中，可以使用 sort() 函数和方法对数组进行按行或按列排序，还可以使用 argsort() 函数获得数组元素排序后的索引位置。基本使用方法如下：

np.sort(a,axis)：对数组 a 按行或者列排序，生成一个新的数组。当 axis=1 时，按行排序；当 axis=0 时，按列排序。

a.sort(axis): 对数组 a 使用 sort() 方法进行排序，因 sort() 方法是直接作用在数组对象上的，所以会改变原始数组。

np.argsort(a)：返回对数组 a 的元素进行排序后的索引位置。

```
In [8]: narr1 = np.array([[1,4,3,2],[11,10,9,12],[7,6,5,8]])
   ...: narr1
Out[8]:
array([[ 1,  4,  3,  2],
```

```
       [11, 10,  9, 12],
       [ 7,  6,  5,  8]])

In [9]: np.sort(narr1,axis = 1)          # 对数组按行排序，原数组不发生改变
Out[9]:
array([[ 1,  2,  3,  4],
       [ 9, 10, 11, 12],
       [ 5,  6,  7,  8]])

In [10]: narr1                           # 原数组没有改变
Out[10]:
array([[ 1,  4,  3,  2],
       [11, 10,  9, 12],
       [ 7,  6,  5,  8]])

In [11]: narr1.sort(axis = 0)            # 对数组按列进行排序，直接在原数组上修改

In [12]: narr1                           # 原数组被改变
Out[12]:
array([[ 1,  4,  3,  2],
       [ 7,  6,  5,  8],
       [11, 10,  9, 12]])

In [13]: np.argsort(narr1)               # 返回对数组 narr1 中的元素进行排序后的索引位置
Out[13]:
array([[0, 3, 2, 1],
       [2, 1, 0, 3],
       [2, 1, 0, 3]], dtype=int64)
```

2.1.3　条件筛选

数组中的元素可以通过索引和切片进行访问或修改，其操作同列表。

对数组元素进行筛选可以通过条件表达式、where() 和 extract() 函数实现。

1. 条件表达式筛选

例如，创建一个范围在 32 到 56 之间的随机整数数组，表示 12 岁儿童的体重（kg）。分别筛选出其中体重低于 35kg 和体重在 40 ～ 50kg 的数据。

```
In [1]: import numpy as np
   ...: np.random.seed(2025)
   ...: w = np.random.randint(33,56,(3,5))      # 创建随机数据
   ...: w
Out[1]:
array([[51, 45, 36, 52, 45],
       [33, 55, 38, 43, 47],
       [34, 41, 38, 38, 48]])
```

获取体重小于 35kg 的数据。

```
In [2]: w < 35                                  # 获得体重低于 35kg 的布尔数组
Out[2]:
array([[False, False, False, False, False],
       [ True, False, False, False, False],
       [ True, False, False, False, False]])
```

返回的是逻辑值，从一堆逻辑之中很难发现要找的数据，但是逻辑值可以作为索引使用。

```
In [3]: w[w<35]                                    # 筛选出体重低于 35kg 的数组元素
Out[3]: array([33, 34])
```

逻辑值作为索引时仅返回真值。

```
In [4]: cond = (w>40) & (w<50)                     # 获得体重在 40~50kg 的布尔数组
   ...: cond
Out[4]:
array([[False,  True, False, False,  True],
       [False, False, False,  True,  True],
       [False,  True, False, False,  True]])

In [5]: w[cond]
Out[5]: array([45, 45, 43, 47, 41, 48])
```

2. where() 函数筛选

在 NumPy 中使用 where() 函数返回数组中满足给定条件的元素的索引，基本格式为：

```
np.where(condition)
```

condition 为筛选条件，返回结果以元组的形式给出，原数组有多少维，输出的元组中就包含多少个数组，分别对应符合条件元素的各维度索引。

例如，使用 where() 函数筛选上述儿童体重在 40 ～ 50kg 的数据。

```
In [6]: idx = np.where((w>40)&(w<50))              # 获得体重在 40~50kg 的元素的索引
   ...: idx
Out[6]:
(array([0, 0, 1, 1, 2, 2], dtype=int64),
 array([1, 4, 3, 4, 1, 4], dtype=int64))
```

注意，这里返回的结果元组中的第一个元素 array([0, 0, 1, 1, 2, 2], dtype=int64) 是满足条件的行索引，第二个元素 array([1, 4, 3, 4, 1, 4], dtype=int64) 是满足条件的列索引，如果元组 idx[0][0] 是 0，idx[1][0] 是 1，组合起来就是（0，1）索引位置，该位置上的元素为 44。

根据数组元素的索引，筛选出体重在 40 ～ 50kg 的数组元素。

```
In [7]: w[idx]
Out[7]: array([45, 45, 43, 47, 41, 48])
```

3. extract() 函数筛选

按某条件查找，返回元素，格式为：extract(condition, arr)。

上面筛选出体重低于 35kg 和体重在 40 ～ 50kg 的数据，也可以一步筛选到位，如下：

```
In [8]: np.extract(((w>40) & (w<50))| (w<35), w)
Out[8]: array([45, 45, 33, 43, 47, 34, 41, 48])
```

“&”符号便是“且”的关系，是交集的关系；“|”符号是“或”的关系，是并集的关系。

2.2 Pandas

Pandas 是一个基于 NumPy 的库，它为数据处理和分析提供了更加灵活和强大的数据结构。Pandas 主要有两个核心的数据结构：Series 和 DataFrame。Series 是一种一维数组，它可以容纳任何数据类型（如整数、字符串、浮点数、Python 对象等），并且提供“显性”的轴索引（或标签）index。DataFrame 是一个二维的表格型数据结构，可以看作是由多个 Series 组成的，其中每一列可以有不同的数据类型。DataFrame 支持按标签或位置索引数据，并且提供了大量的方法来进行数据清洗、处理和分析，如排序、选择、过滤、分组、合并等。Pandas 的一个重要特点是不同列可以有不同的数据类型，并且提供了标签化索引的能力，使得数据检索更为方便。

2.2.1 Series 的创建与访问

Series 是一维数组序列，也称为序列，存储一行或一列数据。它由一组数据和相应的数据索引 index 组成。Series 的索引不局限于整数，还可以自定义为字符串，如 a、b、c、d，像这样不是默认的从开始的自然数索引，习惯称之为标签，再如 first、second、third 等。使用索引可以非常方便地在 Series 序列中取值。

Series 对象使用 pd.Series() 函数创建，使用方式如下：

```
pd.Series(data, index, dtype)
```

参数说明：

data：序列数据，可以是 list、dict 或 NumPy 中的一维 ndarray 数组。

index：序列索引（标签），可以用列表表示，默认为从 0 开始的自然数索引。

dtype：序列的数据类型，默认根据 data 中的数据自动设置。

例如，分别通过 list、dict 和一维 ndarray 数组创建 Series 对象。

```
In [1]: import numpy as np
   ...: import pandas as pd
   ...: # 直接给定列表创建序列 series1
   ...: series1 = pd.Series([45,12,56,24,35],['a','b','c','d','e'])
   ...: series1
Out[1]:
a    45
b    12
c    56
d    24
e    35
dtype: int64

In [2]: narr1 = np.array(['father','mother','brother','sister','son','daughter'])
   ...: cn = pd.Series(narr1)      # 通过数组 narr1 创建序列 cn
   ...: cn
Out[2]:
```

```
0      father
1      mother
2     brother
3      sister
4         son
5    daughter
dtype: object

In [3]: dict1 = {'orange':4.0,'pear':3.5,'apple':6,'grape':12.5}
   ...:                              # 通过字典 dict1 创建序列 price，索引为字典的键
   ...: price = pd.Series(dict1)
   ...: price
Out[3]:
orange     4.0
pear       3.5
apple      6.0
grape     12.5
dtype: float64
```

Series 对象的访问是通过对其所在的位置、对应的索引进行的，还可以进行切片和按条件筛选访问。

```
In [1]: import pandas as pd
   ...: L_1 = ['aa','bb','cc','dd','ee']
   ...: s1 = pd.Series(L_1)
   ...: s1
Out[1]:
0    aa
1    bb
2    cc
3    dd
4    ee
dtype: object

In [2]: s1[3]
Out[2]: 'dd'

In [3]: s2 = pd.Series(L_1,index=list("abcde"))
   ...: s2
Out[3]:
a    aa
b    bb
c    cc
d    dd
e    ee
dtype: object

In [4]: s2["d"]
Out[4]: 'dd'
```

访问的方式同列表的方式，其切片也一致。

```
In [5]: s1[:3]
Out[5]:
0    aa
1    bb
2    cc
dtype: object
```

```
In [6]: s1[::2]
Out[6]:
0    aa
2    cc
4    ee
dtype: object

In [7]: s1[::-1]
Out[7]:
4    ee
3    dd
2    cc
1    bb
0    aa
dtype: object
```

2.2.2　DataFrame 的创建与访问

DataFrame 称为数据框，是带索引（标签）的二维表格数据结构，存储多行和多列数据集合，由多个 Series 组成，是 Series 的容器，其数据结构如图 2-1 所示。

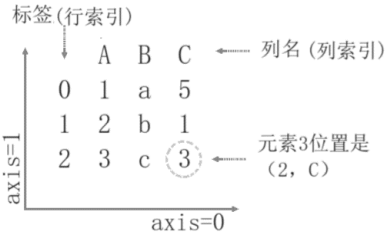


图 2-1　数据框结构示意图

使用 DataFrame() 函数来创建数据框对象，格式如下：

```
pd.DataFrame(data,index,columns,dtype)
```

参数说明：

data：数据内容，可以是二维数组、Series 序列、列表、字典、数据框对象等。

index：数据框的行索引，如果没有指定行索引，默认为从 0 开始的自然数索引。

columns：数据框的列名，即列索引，如果没有指定列索引，默认为从 0 开始的自然数索引。

dtype：指定数据框的数据类型，默认为 None，数据类型根据创建数据框对象的数据内容自动设置。

```
In [1]: import pandas as pd
   ...: dic = {'A': [1,2,3], 'B': list("abc"),"C":[5,1,3]}
   ...: df = pd.DataFrame(dic)       # 按字典创建数据框
   ...: df
```

```
Out[1]:
   A  B  C
0  1  a  5
1  2  b  1
2  3  c  3

In [2]: import numpy as np
   ...:
   ...: arr = np.random.randint(0,10,(3,4))
   ...: df1 = pd.DataFrame(arr,columns=list("ABCD")) # 由数组创建数据框
   ...: df1
Out[2]:
   A  B  C  D
0  0  9  4  6
1  7  2  1  4
2  8  7  8  5
```

查看数据框的行数（数据条数）跟 Series 一致，使用 len() 函数；查看数据框的列名用 df.columns；提取数据框的行索引用 df.index；获取数据框的形状（行数与列数）用 df.shape。

```
In [3]: len(df)
Out[3]: 3

In [4]: df.columns          # 提取列名
Out[4]: Index(['A', 'B', 'C'], dtype='object')

In [5]: df.index            # 提取行索引
Out[5]: RangeIndex(start=0, stop=3, step=1)

In [6]: df.index.tolist()   # 转化为列表
Out[6]: [0, 1, 2]

In [7]: df.shape
Out[7]: (3, 3)
```

iloc[] 和 loc[] 是 Pandas 中常用的两种索引方式，可以查看数据框中的某个元素或某个块元素，它们的主要区别在于索引时所用的对象不同。为了方便描述，这里做一个约定，当行索引 index 为自定义的类型时，如 a、b、c，或标签不是从 0 开始的，如 1，2，3…等，称之为“标签”；默认的从 0 开始的自然数索引，称之为索引号。

iloc[] 通过索引号（行号和列号）来进行索引，它使用整数位置来确定位置。这意味着给定的索引号是一个自然数列，即从 0 开始，如 0，1，2，…，15。

loc[] 通过行标签或列标签来进行索引，它使用标签名字来确定位置。这意味着给定的索引必须是行（列）的标签，可以是 a,b,c,…或 first，second，…等。

df.iloc[] 格式为：**df.iloc[m:n,p:q]**，其中：m、n 为行索引；p、q 为列索引。

df.loc[] 格式为：**df.loc[M:N,P:Q]**，其中：M:N 为行标签；P:Q 为列标签。

```
In [1]: import numpy as np
   ...: import pandas as pd
   ...: arr = np.random.randint(0,10,(3,4))
   ...: df1 = pd.DataFrame(arr,columns=list("ABCD"))
   ...: df1
Out[1]:
   A  B  C  D
```

```
0  3  4  9  6
1  2  9  4  4
2  8  5  8  0

In [2]: df1.iloc[:2,1:3]
Out[2]:
   B  C
0  4  9
1  9  4

In [3]: df1.loc[1:2,"B"]
Out[3]:
1    9
2    5
Name: B, dtype: int32

In [4]: df1.loc[:2,"A":"C"]              # 提取 A 到 C 列
Out[4]:
   A  B  C
0  3  4  9
1  2  9  4
2  8  5  8

In [5]: df1.loc[:2,["A","C"]]            # 按给定的列名（做成了列表）提取
Out[5]:
   A  C
0  3  9
1  2  4
2  8  8

In [6]: df1.iloc[1]                      # 提取行索引为 1 的行
Out[6]:
A    2
B    9
C    4
D    4
Name: 1, dtype: int32

In [7]: df1.loc[2]                       # 提取行标签为 2 的行
Out[7]:
A    8
B    5
C    8
D    0
Name: 2, dtype: int32
```

注意，上面代码中的 "A":"C" 和 ["A","C"] 提取的结果差异。

提取整行和整列的代码如下：

```
In [8]: df2 = pd.DataFrame(arr,columns=list("ABCD"),index=list("abc"))
   ...: df2
Out[8]:
   A  B  C  D
a  3  4  9  6
b  2  9  4  4
c  8  5  8  0
```

```
In [9]: df2.loc["c"]                    # 按编号提取 c 行
Out[9]:
A    8
B    5
C    8
D    0
Name: c, dtype: int32

In [10]: df2.iloc[1]                    # 按索引号提取 b 行
Out[10]:
A    2
B    9
C    4
D    4
Name: b, dtype: int32

In [11]: df2["B"]                       # 提取 B 列
Out[11]:
a    4
b    9
c    5
Name: B, dtype: int32
```

提取列时，直接在数据框变量后跟列名，如 df[" 列名 "]，也可以写成：df. 列名。

```
In [12]: df2.B
Out[12]:
a    4
b    9
c    5
Name: B, dtype: int32
```

有时，为了方便查阅数据框的开头部分和结尾部分，可以使用 head() 和 tail() 函数。

```
In [13]: df2.head(2)                    # 查看 df2 的前 2 行数据，默认是 5 行
Out[13]:
   A  B  C  D
a  3  4  9  6
b  2  9  4  4

In [14]: df2.tail(2)                    # 查看 df2 的末尾 2 行数据，默认是 5 行
Out[14]:
   A  B  C  D
b  2  9  4  4
c  8  5  8  0
```

head() 和 tail() 函数默认只查看 5 行数据。

2.2.3 条件筛选

选取 DataFrame 中的数据除了使用索引和切片，还可以根据条件筛选满足条件的数据。根据一定的条件，对数据进行提取，使用方式如下：

```
dataframe[condition]
```

其中：condition 是过滤条件，返回值是 DataFrame。

常用的 condition 类型如下：

比较运算：==、<、>、>=、<=、!=，如：df[df.column_A>1000)]。

范围运算：between(left,right)，如：df[df. column_A.between(100,1000)]。

空置运算：pandas.isnull(column) ，如：df[df.title.isnull()]。

字符匹配：str.contains(patten,na = False) ，如：df[df.title.str.contains(' 电台 ',na=False)]。

逻辑运算：&（与），|（或），~（取反）；如：df[(df. column_A >=100)&(df. column_A <=1000)]，与 df[df. column_A.between(100,1000)] 等价。

```
In [1]: import numpy as np
   ...: import pandas as pd
   ...: np.random.seed(20231005)                # 随机种子，保证每次取随机数相同
   ...: arr = np.random.randint(0,50,(4,3))     # 产生随机的数组
   ...: df3 = pd.DataFrame(arr,columns=list("ABC"),index=list("abcd"))
   ...: df3
Out[1]:
    A   B   C
a   4  46  41
b  48  13  38
c  38  47  29
d  46  35  38
In [2]: df3.B.between(10,40)             # 筛选出 B 列位于 10 到 40 之间的数据，返回逻辑值
Out[2]:
a    False
b     True
c    False
d     True
Name: B, dtype: bool
In [3]: df3[df3.B.between(10,40)]        # 筛选出符合条件的数据行
Out[3]:
    A   B   C
b  48  13  38
d  46  35  38
```

df3.B.between(10,40) 表示选出 B 列中位于 10 到 40 之间的数据值，返回的是逻辑值，逻辑真假均返回，见上面代码 Out[2]，故没有筛选出需要的数据。但逻辑值可以作为索引，所以可以将选出的结果作为索引在 df3 中进行提取，即 df3[df3.B.between(10,40)] 的作用就是利用逻辑值筛选出符合条件的数据行，见上面代码 Out[3]。

```
In [4]: df3[~df3.B.between(10,40)]       # 筛选出介于 10 到 40 之间以外的数据行
Out[4]:
    A   B   C
a   4  46  41
c  38  47  29

In [5]: df3[~(df3.C>40)]                 # 筛选出 C 列中 >40 的数据并取反
Out[5]:
    A   B   C
b  48  13  38
c  38  47  29
d  46  35  38
```

2.2.4 数据的增删改

数据的增删改是数据处理的基本操作，DataFrame 的增删改操作方法丰富，灵活多样。

1．增加行列数据

增加列的方法十分简单，直接在数据框对象后增加列名赋值列数据即可。

```
df[" 列名 "] = column_data
In [6]: df3["D"] = range(len(df3["B"]))# len(df3["B"]) 表示 B 列的元素个数

In [7]: df3
Out[7]:
    A   B   C  D
a   4  46  41  0
b  48  13  38  1
c  38  47  29  2
d  46  35  38  3
```

len(df3[“B”]) 表示提取 df3 中 B 列的行数（B 列的元素个数），range(len(df3[“B”])) 表示产生 B 列行数个数据，这样做成跟 B 列一样长度的数据，成为新的 D 列数据，即 df3[“D”] = range(len(df3[“B”]))。注意：若列名 D 存在，则 D 列原数据将被覆盖；若列名为 D 的列不存在，则表示新增列，并且新增作为最后一列。

增加一行数据可以使用 loc[] 的方式，例如增加标签为 4，数据为列表 [1,2,3,4,5]。

```
In [10]: w = [1,2,3,4,5]
    ...: df3.loc[4] = w                                    # 在原数据框上增加行

In [11]: df3
Out[11]:
    A  D2   B   C  D
a   4   0  46  41  0
b  48   1  13  38  1
c  38   2  47  29  2
d  46   3  35  38  3
4   1   2   3   4  5
```

合并两个数据框可以使用 concat() 函数。concat() 函数用于将多个数据结构（如 Series、DataFrame）水平连接或垂直堆叠在一起。其基本语法如下：

```
pandas.concat(objs, axis=0, join='inner', ignore_index=False, **kwargs)
```

参数说明：

objs：要连接的对象序列，可以是 Series 或 DataFrame。

axis：连接轴，默认为 0（垂直叠加）、1（水平连接）。

join：连接方式，默认为 'inner'，也可以是 'outer'。

ignore_index：是否重置索引，默认为 False；当设置为 True 时，返回的 DataFrame 的索引将重新排序。

```
In [9]: d ={'A':11, 'D2':12, 'B':13, 'C':14, 'D':15}
   ...: d0 = pd.DataFrame(d,index=[0])                  # 给出了索引值 index=[0]
   ...: pd.concat([df3,d0],ignore_index=True)           # 生成一个新的数据框
```

```
Out[9]:
    A  D2   B   C   D
0   4   0  46  41   0
1  48   1  13  38   1
2  38   2  47  29   2
3  46   3  35  38   3
4  11  12  13  14  15
```

请注意，用字典做成数据框时，当字典的键值不是列表或元组，而是数值型或者字符型时，需要给出 index。

2．删除数据

使用 drop() 函数可以删除 DataFrame 中的数据，使用方式如下：

```
df.drop(标签,axis,inplace)
```

参数说明：

标签：表示待删除的行索引名或列索引名。

axis：axis=0 表示删除行，axis=1 表示删除列，默认为 0，即删除行。

inplace：布尔值，表示操作是否对原数据生效，默认为 False，表示不修改原数据，返回的是一个原数据的复制；若设置 inplace = True，则直接修改原数据。

```
In [12]: df3.drop(4,inplace=True)                          # 在数据框上直接删除行标签为 4 的行

In [13]: df3
Out[13]:
    A  D2   B   C  D
a   4   0  46  41  0
b  48   1  13  38  1
c  38   2  47  29  2
d  46   3  35  38  3
In [14]: df3.drop("D2",axis=1)                             # 删除列 D2 并生成新的数据框
Out[14]:
    A   B   C  D
a   4  46  41  0
b  48  13  38  1
c  38  47  29  2
d  46  35  38  3

In [15]: df3.drop(["D2","D"],axis=1,inplace=True)          # 在原数据框上删除 D2 和 D 列

In [16]: df3
Out[16]:
    A   B   C
a   4  46  41
b  48  13  38
c  38  47  29
d  46  35  38
```

3．修改数据

对于 DataFrame 中的数据，可以根据需求修改某一个值、修改某一列、替换单个或多个值等。修改数据可以理解为赋值覆盖，所以可以使用 iloc[] 或者 loc[] 方法对已有的值直接赋值覆盖，以达到修改的目的。

若替换单个和多个值，也可以使用 replace() 函数，使用方式为：

```
df.replace(old_value,new_value,inplace)
```

参数说明：

old_value：需要替换的值。当需要同时替换多个值时，使用字典数据即可。

New_value：替换后的值。

inplace：布尔值，表示操作是否对原数据生效，默认为 False，表示不修改原数据，返回的是一个原数据的复制；若设置 inplace = True，则直接修改原数据。

```
In [17]: df3.loc["c","B"] = 0            # 将 df3 中 c 行 B 列交叉位置上的数据修改为 0

In [18]: df3                             # 原数据框被修改
Out[18]:
    A   B   C
a   4  46  41
b  48  13  38
c  38   0  29
d  46  35  38

In [19]: df3["A"] = [0,1,2,3]            # 修改 df3 的 A 列
    ...: df3
Out[19]:
   A   B   C
a  0  46  41
b  1  13  38
c  2   0  29
d  3  35  38

In [20]: df3.loc["d"] = [2,3,0]          # 修改 df3 的 d 行
    ...: df3
Out[20]:
   A   B   C
a  0  46  41
b  1  13  38
c  2   0  29
d  2   3   0

In [21]: df3.replace(0,99,inplace=True) # 将 df3 中所有的 0 修改为 99

In [22]: df3
Out[22]:
    A   B   C
a  99  46  41
b   1  13  38
c   2  99  29
d   2   3  99

In [23]: df3.replace({1:88,2:77})        # 将 df3 中所有的 1 换成 88，2 换成 77
Out[23]:
    A   B   C
a  99  46  41
b  88  13  38
c  77  99  29
d  77   3  99
```

2.2.5　排序

数据框的排序分为两类，一类是对索引的排序，另一类是针对某行某列值的排序。

标签索引排序：sort_index(axis,ascending,inplace)

行列的值排序：sort_values(by,axis,ascending,inplace)

参数说明：

axis：控制排序的轴方向，axis=0 按行排序，axis=1 按列排序，默认为 0。

ascending：默认为 True，升序；为 False 时则按降序排序。

by：按指定关键字排序，由行索引名或列索引名组成的列表。

```
In [24]: df3
Out[24]:
    A   B   C
a  99  46  41
b   1  13  38
c   2  99  29
d   2   3  99

In [25]: df3.sort_index(axis=0,ascending=False)          # 按行索引降序排
Out[25]:
    A   B   C
d   2   3  99
c   2  99  29
b   1  13  38
a  99  46  41

In [26]: df3.sort_values(by="B")                         # 按照 B 列升序排
Out[26]:
    A   B   C
d   2   3  99
b   1  13  38
a  99  46  41
c   2  99  29
```

2.2.6　索引重置

将行索引设置为默认从 0 开始的自然数索引，或者替换成数据框中的某列或其他列表、序列为索引。常用的有 reset_index() 和 set_index() 等方式。

reset_index() 用于替换数据框中的行索引为默认的从 0 开始的自然数索引，使用方式如下：

```
reset_index（drop,inplace）
```

参数说明：

drop：表示是否删除原索引，默认为 False，即保留原索引作为一个新的列存在。

inplace：表示是否在原数据框上操作，默认为 False，生成一个新的数据框。

```
In [27]: df3.reset_index()                          # 直接换成默认的索引，原索引保存为一列
```

```
Out[27]:
  index   A   B   C
0     a  99  46  41
1     b   1  13  38
2     c   2  99  29
3     d   2   3  99

In [28]: df3.reset_index(drop=True,inplace=False) # 启用了 drop 参数为 True
Out[28]:
    A   B   C
0  99  46  41
1   1  13  38
2   2  99  29
3   2   3  99
```

set_index() 用于将数据框中的某列替换为行索引。

```
set_index(keys, drop=True, append=False, inplace=False)
```

参数说明：

keys：列标签或列标签的列表，将被用作新索引。

drop：布尔值，默认为 True，即设置新索引后，原索引列将从数据框中删除。如果为 False，则原索引列将保留为数据框的一个普通列。

append：布尔值，默认为 False。如果为 True，则将新的列添加到现有索引中，形成多重索引。

inplace：布尔值，默认为 False。如果为 True，则修改原数据框，否则返回一个新的数据框。

```
In [29]: df3.set_index("B")                    # 将原数据框中的 B 列设置为新的索引
Out[29]:
     A   C
B
46  99  41
13   1  38
99   2  29
3    2  99
```

其实，也可以直接对 df3.index 赋值。

```
In [30]: df3.index = [1,2,3,4]

In [31]: df3
Out[31]:
    A   B   C
1  99  46  41
2   1  13  38
3   2  99  29
4   2   3  99
```

2.3　读存数据

对于一些二维的数据表格，如 CSV、Excel、TXT 等格式的数据，如何读取到 Pandas 中作为数据框格式使用呢？反过来，已经处理好的数据框格式的数据又如何进行保存呢？

2.3.1　读取数据

1. 读取 CSV 与 TXT 文件

CSV（Comma-Separated Values，逗号分隔值）是一种纯文本格式的文件，用来存储表格数据（数字和文本）。在 CSV 文件中，记录是由字段组成的，字段之间由某种字符或字符串分隔，最常见的是逗号或制表符（\t）。记录之间则以某种换行符分隔。CSV 文件可以由任意数目的记录组成，所有记录都有完全相同的字段序列。

在 Pandas 中使用 read_csv() 函数将 CSV 或 TXT 文件中的数据读取到数据框数据结构，其使用方式如下：

```
pd.read_csv(filename,sep,header,index_col,nrows,skiprows,encoding)
```

参数说明：

filename：表示要读取的文件名（或含有路径的文件名），类型可以是 .txt 或 .csv。

sep：指定数据的分隔符，默认为逗号“,”。

header：指定将哪一行作为列索引，默认为 infer，表示自动识别。当 header = 0 时，表示将文件的第一行作为数据框数据的列索引。

index_col：指定列作为行索引，index_col=0 表示将第一列作为行索引。

nrows：设置需要读取数据中的前 n 行数据。

skiprows：需要跳过的行数。

encoding：设置文本编码格式，默认值为“utf-8”，中文系统中 ANSI 类型的编码应设置为“gbk”。

例如，读取如图 2-2 所示的内容数据表。CSV 格式可以用记事本或者 Excel 打开显示内容。

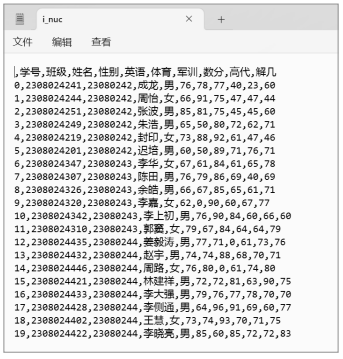


图 2-2　CSV 文件内容

```
In [1]: import pandas as pd
   ...: path_csv = r"d:\OneDrive\i_nuc.csv"
   ...: df = pd.read_csv(path_csv)
   ...: df.head()
Out[1]:
    Unnamed: 0       学号       班级    姓名  性别  英语  体育  军训  数分  高代  解几
0            0  2308024241  23080242    成龙   男    76    78    77    40    23    60
1            1  2308024244  23080242    周怡   女    66    91    75    47    47    44
2            2  2308024251  23080242    张波   男    85    81    75    45    45    60
3            3  2308024249  23080242    朱浩   男    65    50    80    72    62    71
4            4  2308024219  23080242    封印   女    73    88    92    61    47    46

In [2]: df = pd.read_csv(path_csv,index_col=0)
   ...: df.tail()
Out[2]:
       学号       班级    姓名  性别  英语  体育  军训  数分  高代  解几
15  2308024421  23080244   林建祥   男    72    72    81    63    90    75
16  2308024433  23080244   李大强   男    79    76    77    78    70    70
17  2308024428  23080244   李侧通   男    64    96    91    69    60    77
18  2308024402  23080244    王慧    女    73    74    93    70    71    75
19  2308024422  23080244   李晓亮   男    85    60    85    72    72    83
```

从上面的两种读取方式来看，默认的读取方式会将行索引做成一列，而参数 index_col=0 声明将第一列作为行索引后，则是正常显示的数据表。

读取如图 2-3 所示的 TXT 文本表格内容。

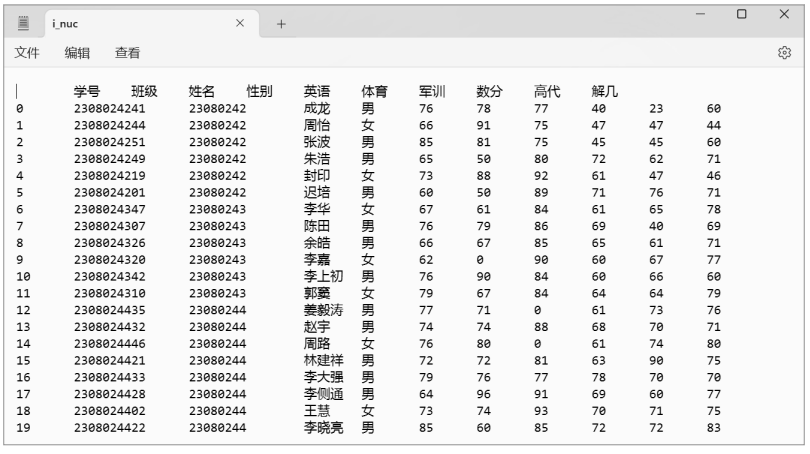


图 2-3　TXT 文本内容

```
In [3]: path_txt = r"d:\OneDrive\i_nuc.txt"
   ...: df = pd.read_csv(path_txt,sep="\t",index_col=0,encoding="gbk")

   ...: df.head()
Out[3]:
       学号       班级    姓名  性别  英语  体育  军训  数分  高代  解几
0  2308024241  23080242   成龙   男    76    78    77    40    23    60
1  2308024244  23080242   周怡   女    66    91    75    47    47    44
2  2308024251  23080242   张波   男    85    81    75    45    45    60
3  2308024249  23080242   朱浩   男    65    50    80    72    62    71
4  2308024219  23080242   封印   女    73    88    92    61    47    46
```

此外，还可以使用 pd.read_table() 函数打开 TXT 文本文件。

```
In [4]: pd.read_table(path_txt,index_col=0,encoding="gbk")
Out[4]:
         学号        班级     姓名 性别  英语  体育  军训  数分  高代  解几
0   2308024241  23080242   成龙  男  76  78  77  40  23  60
1   2308024244  23080242   周怡  女  66  91  75  47  47  44
2   2308024251  23080242   张波  男  85  81  75  45  45  60
3   2308024249  23080242   朱浩  男  65  50  80  72  62  71
4   2308024219  23080242   封印  女  73  88  92  61  47  46
5   2308024201  23080242   迟培  男  60  50  89  71  76  71
6   2308024347  23080243   李华  女  67  61  84  61  65  78
7   2308024307  23080243   陈田  男  76  79  86  69  40  69
8   2308024326  23080243   余皓  男  66  67  85  65  61  71
9   2308024320  23080243   李嘉  女  62   0  90  60  67  77
10  2308024342  23080243  李上初  男  76  90  84  60  66  60
11  2308024310  23080243   郭窦  女  79  67  84  64  64  79
12  2308024435  23080244  姜毅涛  男  77  71   0  61  73  76
13  2308024432  23080244   赵宇  男  74  74  88  68  70  71
14  2308024446  23080244   周路  女  76  80   0  61  74  80
15  2308024421  23080244  林建祥  男  72  72  81  63  90  75
16  2308024433  23080244  李大强  男  79  76  77  78  70  70
17  2308024428  23080244  李侧通  男  64  96  91  69  60  77
18  2308024402  23080244   王慧  女  73  74  93  70  71  75
19  2308024422  23080244  李晓亮  男  85  60  85  72  72  83
```

2. 读取 Excel 文件

Excel 是常见的存储和处理数据的软件，可以使用 read_excel() 函数从 Excel 文件中读取数据，其使用方式如下：

```
pd.read_excel (filename,sheet_name,header,shiprows,index_col)
```

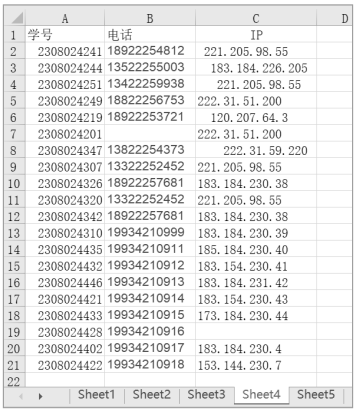
参数说明：

filename：表示要导入 Excel 文件的路径和文件名，支持 .xlsx 格式。

sheet_name：指定 Excel 文件的工作表名称，默认工作表名称为 Sheet1。

header、shiprows、index_col 这 3 个参数的使用方式与 read_csv() 函数相同。

例如，读取如图 2-4 所示的 Excel 文件中的 Sheet4 表格内的数据内容。



	A	B	C	D
1	学号	电话	IP	
2	2308024241	18922254812	221.205.98.55	
3	2308024244	13522255003	183.184.226.205	
4	2308024251	13422259938	221.205.98.55	
5	2308024249	18822256753	222.31.51.200	
6	2308024219	18922253721	120.207.64.3	
7	2308024201		222.31.51.200	
8	2308024347	13822254373	222.31.59.220	
9	2308024307	13322252452	221.205.98.55	
10	2308024326	18922257681	183.184.230.38	
11	2308024320	13322252452	221.205.98.55	
12	2308024342	18922257681	183.184.230.38	
13	2308024310	19934210999	183.184.230.39	
14	2308024435	19934210911	185.184.230.40	
15	2308024432	19934210912	183.154.230.41	
16	2308024446	19934210913	183.184.231.42	
17	2308024421	19934210914	183.154.230.43	
18	2308024433	19934210915	173.184.230.44	
19	2308024428	19934210916		
20	2308024402	19934210917	183.184.230.4	
21	2308024422	19934210918	153.144.230.7	
22				

Sheet1 | Sheet2 | Sheet3 | Sheet4 | Sheet5

图 2-4　Excel 文件表格内容

```
In [5]: path_xlsx = r"d:\OneDrive\i_nuc.xlsx"
   ...: df = pd.read_excel(path_xlsx,sheet_name="Sheet4")
   ...: df.head()
Out[5]:
        学号          电话                IP
0   2308024241  1.892225e+10      221.205.98.55
1   2308024244  1.352226e+10     183.184.226.205
2   2308024251  1.342226e+10      221.205.98.55
3   2308024249  1.882226e+10      222.31.51.200
4   2308024219  1.892225e+10      120.207.64.3
```

注意：当前的版本号支持 .xlsx 格式，对于早期的 Excel 的 .xls 格式，需要安装 xlrd 模块。

2.3.2 保存数据

1. 保存为 Excel 文件

使用 to_excel() 方法可将数据框对象数据写入 Excel 文件中，其使用方式如下：

```
df.to_excel (filename, sheet_name, columns,index)
```

参数说明：

filename：表示要导出为 Excel 文件的文件名（可含路径），支持 .xlsx 格式。

sheet_name：指定 Excel 文件的工作表名称，默认工作表名称为 Sheet1。

columns：指定写入文件的列，为列表类型，默认为 None，表示写入所有列。

index：表示是否将行索引写入文件，默认为 True，表示写入。

例如，2.3.1 节的 df 可以保存为 Excel，文件名为 1.xlsx。

```
In [6]: df.to_excel("c:/Users/yubg/Desktop/1.xlsx")            # 保存在桌面上
```

注意：必须写全文件扩展名，即 .xlsx。

2. 保存为 CSV/TXT 文件

使用 to_csv() 方法将数据框对象的数据导出到 TXT 和 CSV 格式的文件中，其使用方式如下：

```
df.to_csv(filename,sep,header,columns,index,encoding)
```

参数说明：

filename：表示要导出为 CSV 或 TXT 的文件名（可以含路径）。

sep：指定数据的分隔符，默认为逗号“,”。

header：表示是否写入列名，默认为 True，表示写入。

columns：指定写入文件的列，为列表类型，默认为 None，表示写入所有列。

index：表示是否将行索引写入文件，默认为 True，表示写入。

encoding：设置写入文件的编码格式。

例如，2.3.1 节的 df 可以保存为 CSV 和 TXT 文件。

```
In [7]: df.to_csv("c:/Users/yubg/Desktop/2.csv")

In [8]: df.to_csv("c:/Users/yubg/Desktop/3.txt")
```

用 Excel 和记事本打开 2.csv 文件，发现 Excel 打开后的表格中的中文是乱码，如图 2-5 所示，所以在保存的时候需要用 encoding 参数声明保存编码格式。

	A	B	C	D	E
1		瀛冫彿	鋈佃瘽	IP	
2	0	2308024241	18922254812	221.205.98.55	
3	1	2308024244	13522255003	183.184.226.205	
4	2	2308024251	13422259938	221.205.98.55	
5	3	2308024249	18822256753	222.31.51.200	
6	4	2308024219	18922253721	120.207.64.3	
7	5	2308024201		222.31.51.200	
8	6	2308024347	13822254373	222.31.59.220	
9	7	2308024307	13322252452	221.205.98.55	
10	8	2308024326	18922257681	183.184.230.38	
11	9	2308024320	13322252452	221.205.98.55	
12	10	2308024342	18922257681	183.184.230.38	
13	11	2308024310	19934210999	183.184.230.39	
14	12	2308024435	19934210911	185.184.230.40	
15	13	2308024432	19934210912	183.154.230.41	
16	14	2308024446	19934210913	183.184.231.42	
17	15	2308024421	19934210914	183.154.230.43	
18	16	2308024433	19934210915	173.184.230.44	
19	17	2308024428	19934210916		
20	18	2308024402	19934210917	183.184.230.4	
21	19	2308024422	19934210918	153.144.230.7	
22					

2

2

文件　编辑　查看

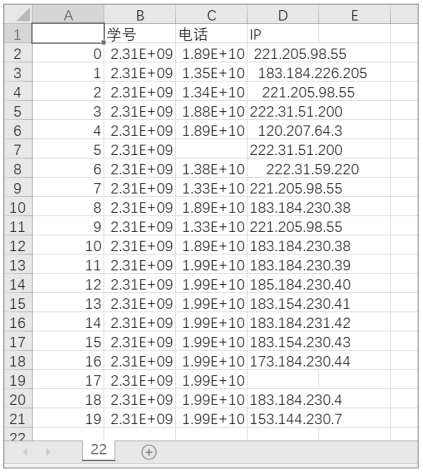
```
,学号,电话,IP
0,2308024241,18922254812.0, 221.205.98.55
1,2308024244,13522255003.0,  183.184.226.205
2,2308024251,13422259938.0,   221.205.98.55
3,2308024249,18822256753.0,222.31.51.200
4,2308024219,18922253721.0,  120.207.64.3
5,2308024201,,222.31.51.200
6,2308024347,13822254373.0,    222.31.59.220
7,2308024307,13322252452.0,221.205.98.55
8,2308024326,18922257681.0,183.184.230.38
9,2308024320,13322252452.0,221.205.98.55
10,2308024342,18922257681.0,183.184.230.38
11,2308024310,19934210999.0,183.184.230.39
12,2308024435,19934210911.0,185.184.230.40
13,2308024432,19934210912.0,183.154.230.41
14,2308024446,19934210913.0,183.184.231.42
15,2308024421,19934210914.0,183.154.230.43
16,2308024433,19934210915.0,173.184.230.44
17,2308024428,19934210916.0,
18,2308024402,19934210917.0,183.184.230.4
19,2308024422,19934210918.0,153.144.230.7
```

行 1，列 1　100%　Windows (CRLF)　UTF-8

图 2-5　用 Excel 和记事本分别打开 CSV 文件

增加保存编码格式为 gbk 后，再打开 22.csv 文件，其显示变为正常，如图 2-6 所示。

```
In [9]: df.to_csv("c:/Users/yubg/Desktop/22.csv",encoding="gbk)
```



	A	B	C	D	E
1		学号	电话	IP	
2	0	2.31E+09	1.89E+10	221.205.98.55	
3	1	2.31E+09	1.35E+10	183.184.226.205	
4	2	2.31E+09	1.34E+10	221.205.98.55	
5	3	2.31E+09	1.88E+10	222.31.51.200	
6	4	2.31E+09	1.89E+10	120.207.64.3	
7	5	2.31E+09		222.31.51.200	
8	6	2.31E+09	1.38E+10	222.31.59.220	
9	7	2.31E+09	1.33E+10	221.205.98.55	
10	8	2.31E+09	1.89E+10	183.184.230.38	
11	9	2.31E+09	1.33E+10	221.205.98.55	
12	10	2.31E+09	1.89E+10	183.184.230.38	
13	11	2.31E+09	1.99E+10	183.184.230.39	
14	12	2.31E+09	1.99E+10	185.184.230.40	
15	13	2.31E+09	1.99E+10	183.154.230.41	
16	14	2.31E+09	1.99E+10	183.184.231.42	
17	15	2.31E+09	1.99E+10	183.154.230.43	
18	16	2.31E+09	1.99E+10	173.184.230.44	
19	17	2.31E+09	1.99E+10		
20	18	2.31E+09	1.99E+10	183.184.230.4	
21	19	2.31E+09	1.99E+10	153.144.230.7	
22					

22

图 2-6　Excel 打开 gbk 编码格式

3. 保存变量

joblib 是一个为 Python 设计的模块，主要用于并行计算和存储大型 NumPy 数组。joblib

提供了一个高效的存储和加载机制，特别是针对大型的 NumPy 数组。它使用 .gz 压缩格式来存储数据。joblib 的并行处理和高效存储功能使其成为了数据科学和机器学习领域中一个非常有用的工具。使用时需要先导入。

```
from joblib import dump, load
import numpy as np

# 创建一个大的 NumPy 数组
big_array = np.random.rand(10000, 10000)

# 保存数组
dump(big_array, 'big_array.joblib')
```

当重新开机需要打开 big_array 数据时，运行下面的代码可加载数组。

```
from joblib import dump, load
loaded_array = load('big_array.joblib')
```

2.4 Pandas 其他操作

在对数据进行清洗时，常用的几个数据框操作如下：

astype()：将指定列的数据类型转换为另一种数据类型。

apply()：将指定的函数应用于指定行/列或者及其行/列中的每个元素。

map()：将指定的函数应用于行/列中的元素。

unique()：返回指定列中的唯一值。

nunique()：返回指定列中不同值的数量。

value_counts()：返回指定列中每个唯一值的计数。

isnull()：返回指示缺失值的布尔值。

例如，现有数据框中的“班级”列数据是数值型，需要将其班级号（班级列数据的后两位）提取出来，这就需要将该数值型转化为字符型，再从字符串中利用切片提取即可。

```
In [1]: import pandas as pd
   ...:
   ...: path_xlsx = r"d:\OneDrive\i_nuc.xlsx"
   ...: df = pd.read_excel(path_xlsx,sheet_name="Sheet3")
   ...: df.head()
Out[1]:
        学号        班级      姓名 性别 英语 体育 军训  数分  高代  解几
0   2308024241  23080242   成龙  男  76  7.8  77  40.0  23.0  60
1   2308024244  23080242   周怡  女  66  9.1  75  47.0  47.0  44
2   2308024251  23080242   张波  男  85  8.1  75  45.0  45.0  60
3   2308024249  23080242   朱浩  男  65    5  80  72.0  62.0  71
4   2308024219  23080242   封印  女  73  8.8  92  61.0  47.0  46

In [2]: df.班级.dtype       # 查看班级列数据类型
Out[2]: dtype('int64')
```

```
In [3]: bj = df.班级.astype("str").map(lambda x:x[-2:])

In [4]: bj
Out[4]:
0     42
1     42
2     42
3     42
4     42
5     42
6     43
7     43
8     43
9     43
10    42
11    43
12    43
13    44
14    44
15    44
16    44
17    44
18    44
19    44
20    44
Name: 班级, dtype: object
```

df.班级.astype("str") 的作用是将班级列拿出来，将其中的每个元素都转化为 str 字符型，再对该列使用 map() 对每个元素提取后两位，这里用到了匿名函数 lambda。

其实，这里也可以使用 apply()。apply() 函数的作用是将一个函数应用到数据框或 Series 的每个元素或行/列上。

```
In [5]: df.班级.apply(lambda x:str(x)[-2:])
Out[5]:
0     42
1     42
2     42
3     42
4     42
5     42
6     43
7     43
8     43
9     43
10    42
11    43
12    43
13    44
14    44
15    44
16    44
17    44
18    44
19    44
20    44
```

```
Name: 班级, dtype: object
```

已经将班级号提取出来做成了一个序列 bj。现在需要知道在这个序列中有几个班级，是哪几个班级，分别出现了几次。

```
In [6]: bj.unique()                # 找出不同班级
Out[6]: array(['42', '43', '44'], dtype=object)

In [7]: bj.nunique()               # 找出不同班级，给出不同班级数
Out[7]: 3

In [8]: bj.value_counts()          # 找出不同班级号出现的次数
Out[8]:
44    8
42    7
43    6
Name: 班级, dtype: int64
```

注意 unique() 和 nunique() 的区别。

为了统计成绩需要，除了知道有“缺考”和“作弊”，还需将数据为空的成绩找出来。

```
In [9]: df.isnull().tail(6)
Out[9]:
     学号    班级    姓名    性别    英语    体育    军训    数分    高代    解几
15  False  False  False  False  False  False  False  True   False  False
16  False  False  False  False  False  False  False  False  False  False
17  False  False  False  False  False  False  False  False  True   False
18  False  False  False  False  False  False  False  False  False  False
19  False  False  False  False  False  False  False  False  False  False
20  False  False  False  False  False  False  False  False  False  False
```

从上面密密麻麻的 False 中很难发现 True，最好能直接给出含有空值的行。这就需要使用 any() 函数。

在 Pandas 库中，any() 函数是 Series 方法之一，用于验证给定系列对象中是否存在任何非零值。此外，any() 函数也可以应用于数据框对象。当应用于数据框时，它将返回一个布尔值，指示数据框中是否存在任何非零值。

```
In [10]: df[df.isnull().any(axis=1)]
Out[10]:
         学号        班级     姓名  性别  英语  体育  军训   数分    高代   解几
15  2308024446  23080244   周路   女  76    8  77   NaN   74.0   80
17  2308024433  23080244  李大强   男  79  7.6  77   78.0  NaN    70
```

any() 返回的是每行或每列是否含有全零的值，当某行/列全部为 0 时，则返回 False。例如：

```
In [11]: data = pd.DataFrame(data={"A":[1,2,3],"B":[0,0,0],"C":[0,2,0]})
    ...: data
Out[11]:
   A  B  C
0  1  0  0
1  2  0  2
2  3  0  0

In [12]: data.any()       # B 列全部为 0
```

```
Out[12]:
A     True
B    False
C     True
dtype: bool
```

所以，对于上述 df 中的空值行，需要知道哪些行有空值，再把这些行的行号或者逻辑值作为索引提取出含有空值的数据行。df.isnull().any(axis=1) 就是判断哪些行有空值，返回的是一个逻辑值，所以 df[df.isnull().any(axis=1)] 就可以提取出含有空值的行。

下面需要将数据中的“班级”列名修改为“班级名”，使用 rename() 函数即可。

```
In [13]: df.rename(columns={"班级":"班级名"}).head()
Out[13]:
        学号       班级名   姓名 性别   英语  体育   军训   数分   高代   解几
0  2308024241  23080242  成龙   男   76   7.8   77  40.0  23.0   60
1  2308024244  23080242  周怡   女   66   9.1   75  47.0  47.0   44
2  2308024251  23080242  张波   男   85   8.1   75  45.0  45.0   60
3  2308024249  23080242  朱浩   男   65     5   80  72.0  62.0   71
4  2308024219  23080242  封印   女   73   8.8   92  61.0  47.0   46
```