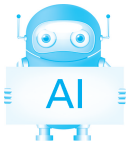
第 5 章



让数据变得可用：数据处理与文件操作

本章学习目标

（1）理解列表、字典、集合三种核心数据结构的特点与适用场景。

（2）掌握列表的创建、索引、增删改查、排序等基本操作及列表推导式。

（3）理解字典的键值映射原理，掌握字典的基本操作方法。

（4）掌握文本文件、CSV 文件与 JSON 文件的读写方法。

（5）理解数据质量对人工智能的影响，掌握脏数据的识别与修复方法。

（6）掌握数据清洗的基本流程，能够完成完整的数据清洗任务。

（7）了解 NumPy 向量化运算与 Pandas 数据处理的基本方法，能够为人工智能应用准备规范的数据集。

5.1 数据结构：列表、字典与集合

数据结构帮助我们组织数据，文件帮助我们保存和交换数据，数据清洗帮助我们提高数据质量，NumPy 与 Pandas 帮助我们更高效地为 AI 准备数据。

数据处理是人工智能应用的核心基础。无论是训练机器学习模型，还是构建智能应用，都需要先将数据组织成程序能够处理的结构。Python 提供了列表、字典和集合三种核心数据结构，本章以电影评论数据为背景，介绍它们的特点与操作方法，并讲解文本文件、CSV 文件与 JSON 文件的读写方法，最终完成从原始脏数据到可训练数据集的完整清洗流程。

数据结构的选择直接影响程序的效率与可读性。需要频繁按位置访问元素时，应选择列表，需要按名称快速查找时，应选择字典，需要去重时，应选择集合。在实际的人工智能项目中，原始数据往往以多种结构混合存在——一条评论可能同时包含文本、评分、标签、用户信息等多种类型，灵活组合使用不同的数据结构才能有效地组织与处理这些数据。

5.1.1 列表：有序的可变数据容器

1. 为什么需要数据容器

列表最适合表示按顺序排列的一组数据，例如评分序列、文件名列表、时间序列数据等。假设需要处理 5000 条电影评论的评分数据，如果为每条评分定义一个单独的变量，将需要 5000 个变量名，这显然不可行。数据容器类型的价值在于：可以用一个变量存储一组数

据，从而方便地进行批量访问、统计和修改。现实世界中的数据天然是成组的，例如一批评论的评分、一部电影多日的票房、一条评论的所有标签等，都适合用容器来组织。列表是Python中最常用的有序数据容器，可以存储任意类型的数据，且支持随时增删元素。以下程序演示了如何用列表存储电影评论的评分数据，并执行索引访问与切片操作。

```
#创建列表:5条评论的评分(1至10分)
ratings = [10, 9, 8, 10, 7]
#索引访问:从0开始,-1表示末尾,切片左闭右开
print(ratings[0], ratings[-1], ratings[1:3])
```

以上程序的运行结果如下：

```
10 7 [9, 8]
```

列表的索引规则可由图5-1直观表示。索引从0开始计数，正向索引依次为0、1、2、…，负向索引从末尾倒数，−1表示最后一个元素。切片ratings[1:3]遵循左闭右开原则，包含索引1对应的元素，不包含索引3对应的元素。

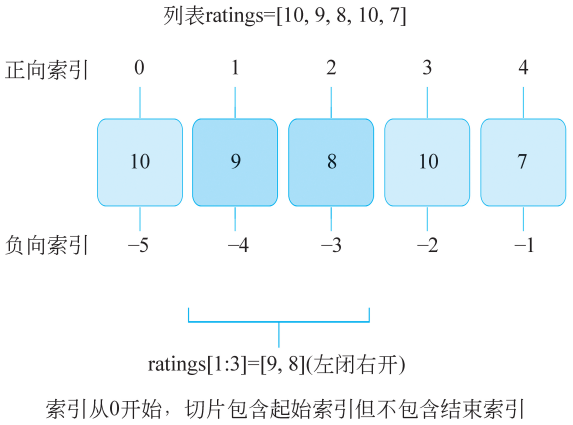


图5-1　列表索引与切片示意图

2. 列表的基本操作

列表支持修改元素、在末尾或指定位置添加元素、删除元素等操作。列表的常用方法如表5-1所示。

表5-1　列表的常用方法

方　法	作　用	示　例
lst[i]=x	修改第 i 个元素	ratings[2]=9
lst.append(x)	在末尾追加一个元素	ratings.append(10)
lst.insert(i, x)	在索引 i 处插入元素	ratings.insert(0, 8)
lst.remove(x)	删除第一个等于 x 的元素	ratings.remove(7)
del lst[i]	删除索引 i 处的元素	del ratings[0]

续表

方　法	作　用	示　例
lst.sort()	原地排序(升序)	ratings.sort()
lst.sort(reverse=True)	原地降序排序	ratings.sort(reverse=True)
len(lst)	返回列表长度	len(ratings)

以下程序演示了列表的修改、追加、插入与删除操作。

```
ratings = [10, 9, 8, 10, 7]

#修改元素
ratings[2] = 9

#追加与插入
ratings.append(10)
ratings.insert(0, 8)
print(ratings)

#删除元素
ratings.remove(7)
del ratings[0]
print(ratings)

#常用统计
print(f"长度:{len(ratings)},最大值:{max(ratings)},最小值:{min(ratings)}")
print(f"平均分:{sum(ratings) / len(ratings)}")
ratings.sort(reverse=True)
print(ratings)
```

以上程序的运行结果如下:

```
[8, 10, 9, 9, 10, 7, 10]
[10, 9, 9, 10, 10]
长度:5,最大值:10,最小值:9
平均分:9.6
[10, 10, 10, 9, 9]
```

注意: 列表的索引从 0 开始,不是 1。对于包含 5 个元素的列表,ratings[0]是第一个元素,ratings[5]会导致索引越界错误。切片操作 ratings[1:3]遵循左闭右开原则,包含索引 1 但不包含索引 3。

3. 列表推导式

列表推导式是 Python 中一种简洁的列表创建语法,可以在一行代码中完成遍历、筛选和转换操作。以下程序对比了传统循环写法与列表推导式,从评分列表中筛选出高分评论(评分大于或等于 9 分),并演示带条件转换的推导式。

```
ratings = [10, 9, 8, 10, 7, 9, 6, 10, 8, 9]

#传统写法
high_ratings = []
```

```
for r in ratings:
    if r >= 9:
        high_ratings.append(r)
print(high_ratings)

#列表推导式：筛选高分
high_ratings = [r for r in ratings if r >= 9]
print(high_ratings)

#带转换的推导式:评分转为情感标签
tags = ["好评" if r >= 9 else "中评" if r >= 6 else "差评" for r in ratings]
print(tags)
```

以上程序的运行结果如下：

```
[10, 9, 10, 9, 10, 9]
[10, 9, 10, 9, 10, 9]
['好评', '好评', '中评', '好评', '中评', '好评', '中评', '好评', '中评', '好评']
```

列表推导式的基本语法为[表达式 for 变量 in 可迭代对象 if 条件]，其含义是将每个满足条件的元素经过表达式变换后收集成新列表。当筛选逻辑较复杂时，仍建议使用传统的循环写法，以保证代码的可读性。

除了列表推导式，Python 还支持字典推导式与集合推导式，它们的语法结构类似，分别用花括号{}表示。以下程序演示了这两种推导式的用法，将评分列表转换为评分到情感标签的字典，以及提取所有不重复的电影名。

```
#字典推导式:评分映射到情感标签
ratings = [10, 8, 5, 9, 3]
tag_dict = {r: "好评" if r >= 7 else "中评" if r >= 4 else "差评" for r in ratings}
print(f"评分标签字典:{tag_dict}")

#集合推导式:提取不重复电影名
movies = ["肖申克的救赎", "盗梦空间", "肖申克的救赎", "星际穿越", "盗梦空间"]
unique_movies = {m for m in movies}
print(f"不重复电影:{unique_movies}")
```

以上程序的运行结果如下：

```
评分标签字典:{10: '好评', 8: '好评', 5: '中评', 9: '好评', 3: '差评'}
不重复电影:{'星际穿越', '盗梦空间', '肖申克的救赎'}
```

4. 嵌套列表

列表中的元素本身也可以是列表，从而构成二维结构，适合表示表格数据。以下程序用嵌套列表存储多部电影的评分记录，并演示按行遍历与按元素访问。

```
#嵌套列表:每行一部电影的评分记录
movie_ratings = [
    ["肖申克的救赎", 10, 9, 10],
```

```
    ["盗梦空间", 9, 8, 9],
    ["星际穿越", 10, 9, 10],
]

#按行遍历
for row in movie_ratings:
    print(row)

#访问单个元素:第 2 部电影(索引 1)的第 2 个评分(索引 2)
print(f"第二部电影的第二个评分:{movie_ratings[1][2]}")

#计算每部电影的平均评分
for row in movie_ratings:
    name = row[0]
    scores = row[1:]
    avg = sum(scores) / len(scores)
    print(f"《{name}》平均评分:{avg:.1f}")
```

以上程序的运行结果如下:

```
['肖申克的救赎', 10, 9, 10]
['盗梦空间', 9, 8, 9]
['星际穿越', 10, 9, 10]
第二部电影的第一个评分:8
《肖申克的救赎》平均评分:9.7
《盗梦空间》平均评分: 8.7
《星际穿越》平均评分: 9.7
```

嵌套列表的访问使用双重索引,如 movie_ratings[1][2]表示先取第 2 行(索引 1),再取该行的第 3 个元素(索引 2)。这种结构在数据量较小时足够使用,当数据量较大时,建议使用 5.4 节介绍的 Pandas DataFrame。

5. 列表与字符串的转换

在文本处理中,经常需要在列表与字符串之间进行转换。Python 的 split()方法将字符串按分隔符切分为列表,join()方法将列表合并为字符串。这两个方法是自然语言处理预处理的基础操作。以下程序演示了列表与字符串的互转过程。

```
#字符串转列表:按空格分割评论
comment = "这部电影 太好看 了 强烈 推荐"
words = comment.split()
print(f"按空白字符切分结果:{words}")

#列表转字符串:用逗号连接
tags = ["经典", "剧情", "励志"]
tag_str = ",".join(tags)
print(f"标签字符串:{tag_str}")
#按分隔符分割

date_str = "2025-03-15"
parts = date_str.split("-")
print(f"日期各部分:{parts}")
```

以上程序的运行结果如下：

```
按空白字符切分结果:['这部电影', '太好看', '了', '强烈', '推荐']
标签字符串:经典,剧情,励志
日期各部分:['2025', '03', '15']
```

这里并不是自然语言处理中的中文分词，而只是按空白符拆分字符串。

split()不带参数时默认按任意空白字符（空格、制表符、换行符）分割，并自动去除多余空白。带参数时则严格按指定分隔符分割。在处理用户输入的文本时，不带参数的 split() 更为常用。

6. 元组：不可变的有序容器

元组用圆括号()定义，与列表的区别在于元组一旦创建就不能修改。它适合存储固定不变的数据，例如电影的标题与上映年份、地理坐标等。以下程序演示了元组的基本用法。

```
#电影信息元组:(标题,年份,评分)
movie = ("肖申克的救赎", 1994, 9.7)

#解包:一次取出多个值
title, year, score = movie
print(f"电影:{title},年份:{year},评分:{score}")

#元组不可修改,以下语句会报错
# movie[2] = 9.8
```

以上程序的运行结果如下：

```
电影:肖申克的救赎,年份: 1994,评分: 9.7
```

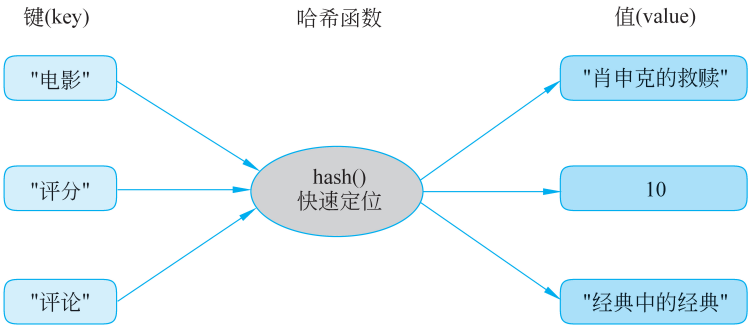
当函数需要返回多个值时，Python 默认以元组形式返回。元组的不可变性也使其可以作为字典的键，而列表则不能。

5.1.2　字典：键值映射的数据容器

1. 键值对：用名字访问数据

列表通过位置索引访问元素，但当数据具有明确含义时，用名字访问比用数字索引更直观。例如，描述一条电影评论时，“评分”“评论”“评论者”这些字段名比“第 0 个”“第 1 个”更容易理解。字典正是为这种需求设计的：它通过键值对存储数据，每个键唯一对应一个值。字典的底层实现是哈希表。其基本原理是：对键进行哈希运算，得到一个整数，再映射到内存中的存储位置，从而实现接近常数时间的快速查找。这也是字典查找速度远快于列表的原因，即列表查找需要逐个比较，而字典查找通过哈希直接定位。图 5-2 展示了字典的键值映射关系。每个键经过哈希函数计算后指向一个存储槽位，槽位中存放对应的值。由于键的唯一性，相同的键不会重复存储。当数据中的每个字段都有明确的含义时，字典通常比列表更适合表达和维护。

以下程序演示了字典的创建与访问过程。



每个键经哈希运算后直接定位到值的存储位置
查找效率为$O(1)$，远快于列表的逐个比较

图 5-2 字典键值映射示意图

```
#创建字典:一条电影评论
review = {
    "电影": "肖申克的救赎",
    "评分": 10,
    "评论": "经典中的经典",
    "评论者": "影迷小王"
}

#通过键访问值
print(review["电影"], review["评分"])

#使用 get 方法访问不存在的键,避免报错
print(review.get("年份", "未知"))
```

以上程序的运行结果如下：

```
肖申克的救赎 10
未知
```

2. 字典的基本操作

字典支持修改、添加、删除元素等操作。字典的常用方法如表 5-2 所示。

表 5-2 字典的常用方法

方　法	作　用	示　例
d[key]	取值(键不存在时报错)	review["评分"]
d.get(key, default)	取值(键不存在时返回默认值)	review.get("年份", "未知")
d[key]=value	新增或修改键值对	review["年份"]=1994
del d[key]	删除键值对	del review["评论者"]
key in d	判断键是否存在	"评分" in review
d.keys()	返回所有键	review.keys()
d.values()	返回所有值	review.values()
d.items()	返回所有键值对	review.items()

以下程序演示了字典的遍历与修改操作。

```
review = {"电影": "肖申克的救赎", "评分": 10, "评论": "经典"}

#新增与修改
review["年份"] = 1994
review["评分"] = 9.7

#遍历键值对
for key, value in review.items():
    print(f"{key}:{value}")
```

以上程序的运行结果如下：

```
电影:肖申克的救赎
评分:9.7
评论:经典
年份:1994
```

3. 字典的典型应用：统计词频

字典非常适合统计类任务。以下程序统计一段评论中每个词出现的次数，这是自然语言处理中常见的预处理步骤。

```
comment = "电影 太 好看 电影 值得 推荐 电影"
words = comment.split()

word_count = {}
for word in words:
    word_count[word] = word_count.get(word, 0) + 1

print(word_count)
```

以上程序的运行结果如下：

```
{'电影': 3, '太': 1, '好看': 1, '值得': 1, '推荐': 1}
```

Python 还提供了 collections.Counter 类，可以更简洁地完成词频统计：from collections import Counter；Counter(words)，结果与上述程序一致。

此外，字典还支持嵌套结构，即字典的值本身也可以是字典，适合表示具有层次关系的数据。以下程序用嵌套字典存储多位评论者的详细信息，并演示嵌套访问与修改。

```
#嵌套字典:多位评论者的信息
reviews_db = {
    "小王": {"电影": "肖申克的救赎", "评分": 10, "评论": "经典"},
    "小红": {"电影": "盗梦空间", "评分": 9, "评论": "烧脑"},
}

#访问嵌套数据
```

```
print(reviews_db["小王"]["电影"])

#新增一位评论者
reviews_db["小明"] = {"电影": "星际穿越", "评分": 10, "评论": "震撼"}

#修改嵌套数据
reviews_db["小红"]["评分"] = 8
print(reviews_db)
```

以上程序的运行结果如下：

```
肖申克的救赎
{'小王': {'电影': '肖申克的救赎', '评分': 10, '评论': '经典'}, '小红': {'电影': '盗梦空间', '评分': 8, '评论': '烧脑'}, '小明': {'电影': '星际穿越', '评分': 10, '评论': '震撼'}}
```

嵌套字典的访问使用连续的方括号索引，如 reviews_db["小王"]["电影"]表示先取键"小王"对应的字典，再取该字典中键“电影”对应的值。JSON 文件读取后的数据通常就是这种嵌套字典结构，5.2 节将详细介绍。

5.1.3 集合：去重与集合运算

1. 集合的去重特性

集合在数据处理中最典型的用途包括：用户 ID 去重、找出两门课共同选修的学生、筛选同时看过两部电影的观众等。集合是一个无序的、元素唯一的数据容器。其最重要的特性是自动去重，即将一个含重复元素的列表转换为集合，重复元素只保留一份。这一特性在数据清洗中极为常用。集合的底层实现与字典类似，也是基于哈希表。正因为如此，集合的成员判断操作(x in s)效率为 $O(1)$，远快于列表的 $O(n)$。当需要频繁判断元素是否存在时，集合是比列表更优的选择。不过集合也有局限性，即它不支持索引访问(因为无序)，不能存储可变对象(如列表、字典)，且元素顺序不可预测。如果需要保持插入顺序的去重容器，可以使用 Python 3.7+中的字典(dict.fromkeys()方法)或 collections.OrderedDict。以下程序演示了集合的去重功能。

```
#评论者列表(含重复)
reviewers = ["小王", "小红", "小王", "小明", "小红", "小李"]

#转为集合去重
unique_reviewers = set(reviewers)
print(f"原始{len(reviewers)}人,去重后{len(unique_reviewers)}人")
print(unique_reviewers)

#转回列表,以便后续处理
reviewer_list = list(unique_reviewers)
print(reviewer_list)
```

以上程序的运行结果如下：

```
原始 6 人,去重后 4 人
{'小明', '小李', '小王', '小红'}
['小明', '小李', '小王', '小红']
```

2. 集合运算

集合支持数学中的交、并、差运算，这在分析多组数据的关系时非常方便。集合的常用运算如表 5-3 所示。

表 5-3　集合的常用运算

运　算	操 作 符	含　义	示　例
交集	&	两个集合共有的元素	a & b
并集	|	两个集合所有的元素	a | b
差集	−	在 a 中但不在 b 中的元素	$a-b$
对称差	^	只属于其中一个集合的元素	a ^ b
子集判断	<=	a 是否为 b 的子集	a <= b

代码如下。

```
#看过《肖申克的救赎》的观众
group_a = {"小王", "小红", "小明"}
#看过《盗梦空间》的观众
group_b = {"小红", "小明", "小李"}
print(f"两部电影都看过:{group_a & group_b}")
print(f"至少看过一部:{group_a | group_b}")
print(f"只看过《肖申克的救赎》:{group_a - group_b}")
print(f"只看过其中一部:{group_a ^ group_b}")
```

以下程序以两组看过不同电影的观众为例，演示集合运算。

以上程序的运行结果如下：

```
两部电影都看过:{'小明', '小红'}
至少看过一部:{'小明', '小李', '小王', '小红'}
只看过《肖申克的救赎》:{'小王'}
只看过其中一部:{'小王', '小李'}
```

集合是无序的，每次输出元素的顺序可能不同。如果需要保持顺序，应在去重后使用列表，并自行维护顺序，或使用 dict.fromkeys()方法。

5.1.4　三种数据结构的对比与选择

列表、字典、集合各有适用场景，选择合适的数据结构能显著提升程序效率。三者的对比如表 5-4 所示。当需要频繁判断某个元素是否存在时，应优先使用集合或字典，因为它们的查找效率为 $O(1)$，而列表为 $O(n)$。例如，判断一个评论者是否在黑名单中，用集合存储黑名单远快于用列表。

记住一句话：处理数据时，先问自己三个问题：要不要保留顺序？要不要按名字查找？要不要去重？答案分别对应列表、字典和集合。