第五章

字符串

字符串是**不可变的**（immutable），一旦创建，字符串中的字符无法直接修改。Python 字符串支持**多种操作和方法**，如字符串的分割（`split`）、替换（`replace`）、查找（`find`）和大小写转换（`upper`，`lower`）等。这些操作在处理文本数据时非常有效，能够简化对大规模文本的分析和处理。

5.1 创建字符串

重要性：★★★★★； **难易度**：★

在 Python 中，字符串是一种不可变的字符序列，创建字符串的基本语法较为简单，主要通过以下几种方式实现：

1. 单引号或双引号创建字符串

使用单引号（`' '`）或双引号（`" "`）可以创建字符串。例如：

```
string1 = 'Hello, World'
string2 = "Hello, Python"
print(string1)  # 输出：Hello, World
print(string2)  # 输出：Hello, Python
```

Python 允许在字符串中使用单双引号，只要引号的形式匹配。

2. 多行字符串

使用三重引号（`''' '''` 或 `""" """`）可以创建多行字符串。适用于较长的文本或需要保留格式的文本内容。例如：

```
message = """This is a multi-line
string example."""
print(message)
```

输出将保留原始的换行格式。

3. 转义字符与原始字符串

如果字符串中包含引号或其他特殊字符，可以使用反斜杠（\）进行转义。例如：

```
string3 = "He said, \"Python is awesome!\""
print(string3)  # 输出：He said, "Python is awesome!"
```

另外，使用前缀 `r` 可以创建原始字符串，不对反斜杠进行转义：

```
raw_string = r"C:\new_folder\test"
print(raw_string)  # 输出：C:\new_folder\test
```

4. 字符串的不可变性一旦字符串被创建，字符串中的字符无法被修改。这意味着尝试修改字符串中的某个字符会导致错误：

```
s = "hello"
s[0] = 'H'  # 报错：TypeError: 'str' object does not support item assignment
```

但可以通过创建新的字符串来变更其内容：

```
new_s = 'H' + s[1:]
print(new_s)  # 输出：Hello
```

5. f-string 格式化 Python 提供了 f-string 格式化方式，允许在字符串中嵌入变量或表达式。例如：

```
name = "Alice"
age = 25
print(f"My name is {name} and I am {age} years old.")
```

f-string 不仅语法简洁，还支持嵌入复杂的表达式。

案例：使用字符串输出客户信息

在营销数据分析的背景下，Python 的元组和字符串可以用于高效处理和存储数据。元组是一种不可变的数据结构，常用于保存不需要修改的多组数据，例如客户的基本信息。其不可变性使其在大数据环境下具备内存效率和处理速度的优势。在数据处理过程中，尤其是当处理涉及营销客户信息的场景时，可以使用元组存储多个客户的姓名、年龄等信息，并将这些数据转换为字符串进行进一步的输出或分析。以下代码演示了如何创建包含客户信息的元组，并将这些元组转化为字符串：

```
# 创建客户数据的元组
customers = (("John Doe", 28, "Male"), ("Jane Smith", 34, "Female"))

# 将元组数据转化为字符串
for customer in customers:
    customer_info = f"Name: {customer[0]}, Age: {customer[1]}, Gender: {customer[2]}"
    print(customer_info)
```

该代码中，首先创建了一个包含客户姓名、年龄和性别的元组，随后利用 `f-string` 将元组中的数据格式化为字符串，并输出格式化后的客户信息。此方法能够快速将元组中的结构化数据转化为可读的文本信息，在营销数据分析中可以用于生成客户报告或日志。

5.2 字符串的基本操作

重要性：★★★★★； **难易度：**★★★

字符串的基本操作包括通过索引访问字符、使用切片提取子字符串、计算长度、进行成员资格检查、连接与重复字符串操作，以及需注意字符串的不可变性，无法直接修改其中的元素。

5.2.1 索引

字符串可以通过**索引**来访问其各个字符。索引是从 0 开始的，即字符串中的第一个字符索引为 `0`，第二个字符索引为 `1`，以此类推。此外，Python 也支持**负索引**，负索引允许从字符串的末尾开始计数。例如，最后一个字符的索引为 `-1`，倒数第二个字符的索引为 `-2`。

假设有一个字符串 `s = "Python"`，可以通过索引访问各个字符，如下所示：

```
s = "Python"
print(s[0])   # 输出：P
print(s[1])   # 输出：y
print(s[-1])  # 输出：n （使用负索引）
print(s[-2])  # 输出：o （倒数第二个字符）
```

上述代码展示了如何使用正索引和负索引访问字符串中的字符。此外，如果尝试访问超出字符串长度的索引（例如 `s[10]`），会引发 `IndexError` 错误。

5.2.2 切片

在 Python 中，**字符串切片**是一种从字符串中提取子字符串的方式，允许访问字符串的某一部分。切片操作通过使用 `[start:end:step]` 的语法来实现，其中：

- **start**：切片的起始索引（包含在结果中），如果省略，则默认为 0。
- **end**：切片的结束索引（不包含在结果中），如果省略，则默认到字符串的末尾。
- **step**：切片时的步长，表示每次跳过的字符数，默认为 1。如果为负数，则表示从右向左提取。

```
s = "ABCDEFGHI"
print(s[2:7])   # 输出：CDEFG
print(s[:5])    # 输出：ABCDE （从索引0开始，步长为1）
print(s[4:])    # 输出：EFGHI （从索引4开始直到结束）
print(s[::2])   # 输出：ACEGI （每隔一个字符提取）
```

负索引允许从字符串的末尾开始计数。例如：

```
print(s[-4:-1])   # 输出：FGHI （从倒数第4个字符开始提取到倒数第2个）
print(s[::-1])    # 输出：IHGFEDCBA （反转字符串）
```

通过指定不同的 `start`、`end` 和 `step` 参数，切片操作可以灵活地提取字符串中的不同部分，甚至可以实现复杂的字符串操作，如反转字符串或跳跃式提取。

案例:使用字符串切片抽取客户反馈信息

在客户反馈数据分析中,**字符串切片**可以帮助有效提取特定信息或部分文本。例如,客户的反馈通常以文本形式提供,可能包含与产品或服务相关的不同关键信息。在进行分析时,可以使用字符串切片来获取这些数据的片段,帮助提取关键信息。

假设有一条客户反馈数据

`feedback = "Product: ABC, Rating: 5, Comment: Excellent service!"`

可以通过字符串切片操作提取其中的具体部分。以下是代码示例:

```
feedback = "Product: ABC, Rating: 5, Comment: Excellent service!"

# 提取产品名称
product = feedback[9:12]  # 输出: ABC

# 提取评分
rating = feedback[22:23]  # 输出: 5

# 提取评论
comment = feedback[33:]   # 输出: Excellent service!
```

在这个示例中,字符串切片操作被用于从反馈中提取产品名称、评分和评论。通过指定起始和结束索引,可以灵活地提取所需的数据片段,这在数据清理和预处理中尤为重要。

5.2.3 字符串长度计算

计算字符串的长度可以通过内置函数 `len()` 完成。该函数接受一个字符串作为参数,并返回其中字符的总数,包括空格和标点符号。这一功能在处理文本数据时非常重要,尤其是在需要对文本长度进行验证的场景中。

```
1 # 定义一个字符串
2 feedback = "Customer service was excellent!"
3
4 # 计算字符串的长度
5 length = len(feedback)
6
7 # 输出结果
8 print(length)  # 输出: 31
```

在这个示例中,`len()` 函数返回字符串中的字符总数。在客户反馈分析中,这种操作可以用于计算反馈内容的长度,从而对文本进行分类或筛选。

5.2.4 成员资格检查

字符串的成员资格检查可以通过 `in` 和 `not in` 运算符完成。它们用于检测子字符串是否存在于给定的字符串中,并返回布尔值 `True` 或 `False`。这些操作在处理文本数据时非常有用,可以快速判断某些关键字是否出现在文本中。

1. 使用 `in` 运算符检查子字符串是否存在：

```
feedback = "The product quality is excellent."
result = "excellent" in feedback
print(result)  # 输出：True
```

2. 使用 `not in` 运算符检查子字符串是否不存在：

```
result = "poor" not in feedback
print(result)  # 输出：True
```

5.2.5 连接与重复

字符串连接和**重复**是两个重要的操作，常用于文本数据的处理。以下将介绍它们的基本语法，并结合代码示例展示其应用。

1. 字符串连接

字符串连接可以通过使用 `+` 操作符，将多个字符串组合成一个新的字符串。例如：

```
greeting = "Hello"
name = "Alice"
result = greeting + " " + name
print(result)  # 输出：Hello Alice
```

在这个例子中，`+` 操作符将两个字符串连接起来，生成新的字符串 `"Hello Alice"`。

另一种高效的方式是使用 `join()` 方法，将一个可迭代对象（如列表）中的元素连接为一个字符串，尤其是在处理大量字符串时更为节省内存：

```
words = ["Python", "is", "fun"]
result = " ".join(words)
print(result)  # 输出：Python is fun
```

2. 字符串重复字符串的重复可以使用 `*` 操作符实现，将一个字符串按指定次数重复。例如：

```
repeat_str = "ha " * 3
print(repeat_str)  # 输出：ha ha ha
```

这种操作常用于生成格式化的分隔符或模式，例如：

```
line = "=" * 10
print(line)  # 输出：==========
```

5.3 字符串的常用方法

重要性：★★★★★； **难易度：**★★

5.3.1 拆分与合并

字符串的拆分与合并是文本处理中的常见操作。可以使用内置的 `split()` 和 `join()` 方法实现这些功能，分别将字符串分割为列表或将列表中的元素组合为字符串。

1. 字符串拆分：`split()`

`split()` 方法用于按照指定的分隔符将字符串拆分成子字符串列表。默认情况下，`split()` 会按照空格分隔字符串。如果需要，可以通过传递参数指定不同的分隔符。

```
sentence = "Python is fun to learn"
words = sentence.split()  # 按空格拆分
print(words)  # 输出：['Python', 'is', 'fun', 'to', 'learn']

# 使用指定分隔符拆分
sentence = "name,email,phone"
fields = sentence.split(',')
print(fields)  # 输出：['name', 'email', 'phone']
```

2. 字符串合并：`join()`

`join()` 方法用于将一个可迭代对象（如列表或元组）中的元素通过指定的分隔符连接成一个字符串。`join()` 方法调用时应在分隔符字符串上调用，并传入需要合并的字符串列表。

```
words = ['Python', 'is', 'fun']
sentence = ' '.join(words)  # 使用空格连接
print(sentence)  # 输出：Python is fun

# 使用自定义分隔符
fields = ['name', 'email', 'phone']
csv_format = ','.join(fields)
print(csv_format)  # 输出：name,email,phone
```

这些操作在处理结构化文本数据（如 CSV 文件）或构建文本报告时非常有用。

案例:使用字符串拆分与合并方法处理财务文本数据

在财务文本分析中,字符串的拆分与合并方法非常实用,尤其是处理包含多项财务数据的文本时。通过 `split()` 和 `join()` 方法,可以轻松拆分和组合数据字段,以便进一步分析和处理。

假设有一行财务数据,以逗号分隔各个字段(例如公司名称、收入、利润等)。可以使用 `split()` 方法将其拆分为独立的字段:

```
financial_data = "Company ABC, 5000000, 1000000, 2024"
fields = financial_data.split(", ")
print(fields)
# 输出: ['Company ABC', '5000000', '1000000', '2024']
```

在这个示例中,`split()` 方法根据逗号和空格将财务数据字符串分割为不同的部分。

当需要将处理过的财务数据重新组合为单个字符串以进行报告生成或存储时,可以使用 `join()` 方法:

```
fields = ['Company ABC', '5000000', '1000000', '2024']
result = ", ".join(fields)
print(result)
# 输出: Company ABC, 5000000, 1000000, 2024
```

在此示例中,`join()` 方法通过逗号和空格将列表中的数据字段合并为一个字符串,便于生成汇总报告。

5.3.2 查找和替换

字符串的查找和替换是文本处理中的重要操作,尤其是在商业文本分析中。Python 提供了内置的 `find()` 和 `replace()` 方法,分别用于查找子字符串的位置以及替换指定的子字符串。以下是它们的基本语法和示例代码:

1. 字符串查找:`find()` 方法

`find()` 方法用于在字符串中查找子字符串的索引位置。如果找到匹配的子字符串,返回其起始索引;否则返回 `-1`。可以指定可选的起始和结束索引,限制查找的范围。

```
1 text = "Revenue for the year is estimated at $5 million."
2 position = text.find("estimated")
3 print(position)  # 输出: 24
```

在该示例中,`find()` 方法返回子字符串 `"estimated"` 在字符串中的位置。

2. 字符串查找:`index()` 方法

字符串的 `index()` 方法用于查找子字符串在主字符串中的位置。其基本语法为:

```
str.index(sub[, start[, end]])
```

- `sub`:要搜索的子字符串。
- `start`:可选,搜索的起始位置。

- `end`：可选，搜索的结束位置。

如果找到该子字符串，`index()` 返回其在主字符串中的最低索引；若未找到，则抛出 `ValueError` 异常。以下是几个示例代码：

```
sentence = "Hello, world!"
position = sentence.index("world")
print(position)  # 输出：7
```

```
# 使用起始参数
phrase = "Python is great. Python is versatile."
position = phrase.index("Python", 10)
print(position)  # 输出：21
```

```
# 子字符串未找到
try:
    position = sentence.index("Java")
except ValueError:
    print("Substring not found.")  # 输出：Substring not found.
```

`index()` 方法在处理字符串搜索时非常有效，尤其是在确定子字符串存在的情况下。

2. 字符串替换：`replace()` 方法

`replace()` 方法用于将字符串中的某个子字符串替换为另一个子字符串。它的基本语法是：

```
str.replace(old, new, count)
```

其中 `old` 是要替换的子字符串，`new` 是替换后的字符串，`count` 是可选参数，表示替换的次数。如果不指定 `count`，将替换所有出现的子字符串。

```
report = "The profit margin was low. The profit margin needs improvement."
new_report = report.replace("profit margin", "revenue")
print(new_report)
# 输出：The revenue was low. The revenue needs improvement.
```

在此示例中，`replace()` 方法将所有出现的 `"profit margin"` 替换为 `"revenue"`，生成了一个新的字符串。

案例：使用字符串替换方法批量修改商业文本

在商业文本分析中，字符串的查找与替换方法是重要的工具，用于文本处理与信息提取。Python 提供了内置的 `find()` 和 `replace()` 方法，分别用于查找子字符串的位置和替换文本中的特定子字符串。这些方法在分析商业合同、交易记录或报告时十分有效。例如，在贸易报告中查找关键字：

```
report = "The total revenue for 2023 was $10 million."
position = report.find("revenue")
print(position)  # 输出：10
```

例如，在财务报告中将 `"revenue"` 替换为 `"income"`：

```
report = "The total revenue for 2023 was $10 million."
new_report = report.replace("revenue", "income")
print(new_report)
# 输出：The total income for 2023 was $10 million.
```

这些方法能够帮助快速处理和修改大量文本数据，在文本分析和报告生成中极为有效。

5.3.3 大小写转换

字符串的大小写转换可以通过以下几种常用的内置方法完成，包括 `upper()`、`lower()`、`capitalize()` 和 `swapcase()`，这些方法在处理文本数据时非常有用，尤其是在标准化、数据清洗和文本分析的场景中。

1. 字符串转换为大写：`upper()` 方法

`upper()` 方法将字符串中的所有字母转换为大写。示例：

```
text = "python is fun"
upper_text = text.upper()
print(upper_text)  # 输出：PYTHON IS FUN
```

2. 字符串转换为小写：`lower()` 方法

`lower()` 方法用于将字符串中的所有字母转换为小写。示例：

```
text = "Hello, WORLD!"
lower_text = text.lower()
print(lower_text)  # 输出：hello, world!
```

3. 首字母大写：`capitalize()` 方法

`capitalize()` 方法将字符串的第一个字母转换为大写，其他字母转换为小写，适用于标题或句子的首字母格式化。示例：

```
text = "python programming"
capitalized_text = text.capitalize()
print(capitalized_text)  # 输出：Python programming
```

4. 大小写互换：`swapcase()` 方法

`swapcase()` 方法将字符串中的大写字母转换为小写，小写字母转换为大写。示例：

```
text = "PyThOn PrOgRaMmInG"
```

```
swapped_text = text.swapcase()
print(swapped_text)  # 输出：pYtHoN pRoGrAmMiNg
```

案例：电子商务文本规范化处理

在电子商务文本分析中，处理用户评论、产品描述或订单信息时，字符串的大小写转换是常见的操作。这些转换有助于对文本进行标准化，从而简化数据分析和清洗过程。Python 提供了多种内置方法来完成大小写转换，常见的有 `upper()`、`lower()`、`capitalize()` 和 `swapcase()`。下面结合代码展示这些方法在电子商务文本分析中的应用。

假设有一段用户评论，转换为大写可以帮助进行不区分大小写的匹配或比较：

```
review = "great product! highly recommended."
uppercase_review = review.upper()
print(uppercase_review)
# 输出：GREAT PRODUCT! HIGHLY RECOMMENDED.
```

在处理订单或用户信息时，可以使用 `lower()` 将所有字母转换为小写以实现一致性，例如：

```
email = "John.Doe@Example.com"
normalized_email = email.lower()
print(normalized_email)
# 输出：john.doe@example.com
```

在处理产品标题或描述时，`capitalize()` 可以用于将首字母大写以确保展示的规范性：

```
product_description = "excellent quality and fast shipping."
capitalized_description = product_description.capitalize()
print(capitalized_description)
# 输出：Excellent quality and fast shipping.
```

当需要将大写字母转为小写、反之亦然时，可以使用 `swapcase()`。例如，在分析用户输入时：

```
comment = "Amazing DEAL! don't miss OUT!"
swapped_comment = comment.swapcase()
print(swapped_comment)
# 输出：aMAZING deal! DON'T MISS out!
```

5.3.4 去除空白字符

去除字符串中的空白符可以使用三种常见的方法：`strip()`、`lstrip()` 和 `rstrip()`。这些方法分别用于去除字符串两端或特定一端的空白符或其他字符。以下是这些方法的基本语法以及相应的代码示例。

1. 去除两端空白符：`strip()`

`strip()` 方法用于去除字符串开头和结尾的所有空白符（包括空格、换行符、制表符等）。示例如下：

```
text = "   Python is great!   "
trimmed_text = text.strip()
print(trimmed_text)
```

```
# 输出: "Python is great!"
```

此方法不会影响字符串中间的空白符，只会去除两端的空白符。

2. 去除左侧空白符：`lstrip()`

`lstrip()` 方法用于去除字符串左侧的空白符，右侧保持不变：

```
text = "   Python is great!   "
left_trimmed_text = text.lstrip()
print(left_trimmed_text)
# 输出: "Python is great!   "
```

3. 去除右侧空白符：`rstrip()`

`rstrip()` 方法去除字符串右侧的空白符，左侧保持不变：

```
text = "   Python is great!   "
right_trimmed_text = text.rstrip()
print(right_trimmed_text)
# 输出: "   Python is great!"
```

这些方法非常适合在处理用户输入或清理文本数据时使用，以确保数据的一致性和整洁性。通过结合这些方法，可以有效地去除不需要的字符，尤其是在数据预处理阶段。

案例：移除客户评论中的空白字符

在商业文本分析中，去除空白字符是数据清理的重要步骤，能够确保分析的准确性。Python 提供了多种方法来去除字符串中的空白字符，常见的方法包括 `strip()`、`lstrip()`、`rstrip()`、`replace()`、以及使用正则表达式。

以下代码示例展示了如何去除字符串中的空白字符：

```
# 示例字符串
comment = "   This is a customer review with extra spaces.   "

# 使用 strip() 方法去除前后空白字符
clean_comment = comment.strip()

# 输出结果
print(clean_comment)
```

在这个例子中，`strip()` 方法用于去除字符串前后的空白字符。对于需要去除所有空白字符的情况，可以使用 `replace()` 方法，如下所示：

```
# 使用 replace() 方法去除所有空白字符
cleaned_comment = comment.replace(" ", "")
print(cleaned_comment)
```

此方法将所有空格替换为空字符，从而实现了彻底清除空白字符的目的。

5.3.5 计数

字符串的 `count()` 方法用于计算指定子字符串在目标字符串中出现的次数。该方法非常适合用于文本处理和字符串分析任务，尤其是在需要统计某个字符或子字符串出现频率时。

```
string.count(substring, start=..., end=...)
```

- `substring`：必选参数，表示需要计数的子字符串。
- `start`（可选）：指定搜索的起始索引，默认为字符串的开头。
- `end`（可选）：指定搜索的结束索引，默认为字符串的末尾。

该方法返回一个整数，表示子字符串在指定范围内出现的次数。如果未找到子字符串，则返回 0。

示例 1：计数字符串中某字符的出现次数

```
message = 'python is popular programming language'
print(message.count('p'))  # 输出：4
```

在上述代码中，`'p'` 在字符串中总共出现了 4 次。

示例 2：使用 `start` 和 `end` 参数

```
string = "Python is awesome, isn't it?"
substring = "i"
count = string.count(substring, 8, 25)
print("The count is:", count)  # 输出：1
```

在这个示例中，计数从索引 8 开始，到索引 25 结束，因此只找到 1 次 `'i'` 字符。

`count()` 方法经常用于文本数据分析中。例如，在处理自然语言数据时，该方法可以快速统计特定词汇或字符的频率，帮助分析文本内容或执行模式匹配任务。

5.4 其他常用的字符串方法

除上一节介绍的字符串常用方法外，字符串还有许多实用的常用方法，参见表5.1，可以结合 `help` 函数自行学习其他常用的字符串方法的用法。

字符串有多个以 `is` 开头的方法，这些方法用于对字符串内容进行各种类型的验证，返回布尔值（`True` 或 `False`）。表5.2列出了常见的字符串内容类型验证方法及其含义和用法：

案例:字符串常用方法综合应用

在商业文本分析中,字符串处理方法的综合应用至关重要。以下是一个示例代码,展示了如何使用 Python 的字符串方法进行文本处理,包括拆分、清洗和连接字符串,以便进行后续的文本分析。

```
# 示例文本
text = """Product,Sales,Profit
Widget A,100,20
Widget B,150,30
Widget C,200,50"""

# 拆分文本为行
lines = text.splitlines()
data = []

# 处理每一行
for line in lines:
    # 拆分每一行的字段
    fields = line.split(',')
    # 清洗字段，去除空白
    cleaned_fields = [field.strip() for field in fields]
    data.append(cleaned_fields)

# 打印处理后的数据
for entry in data:
print(f"Product: {entry[0]}, Sales: {entry[1]}, Profit: {entry[2]}")
```

此代码首先使用 `splitlines()` 方法将文本拆分为多行,然后利用 `split()` 方法分割每行中的字段。接着,通过列表推导式清洗字段,去除多余的空白,最终将处理后的数据存储在一个列表中,便于后续分析。

5.5 字符串格式化

重要性:★★★★★; **难易度:**★★

字符串格式化是一项重要的技能,特别是在处理动态文本输出时,如生成报告、用户提示或数据展示。Python 提供了多种格式化字符串的方法,包括旧式的百分号格式化 (%), `str.format()` 方法,以及较新的 F 字符串格式化 (`f-strings`)。

5.5.1 字符串格式化基本用法

1. 百分号格式化

这是 Python 最早的字符串格式化方式,通过使用% 符号替换占位符。例如:

```
1 name = "Alice"
2 age = 30
3 print("Hello, my name is %s and I am %d years old." % (name, age))
```

在这个例子中，%s 表示字符串占位符，%d 表示整数占位符。该方法虽然简洁，但可读性和灵活性较低，已逐渐被 `str.format()` 和 `f-strings` 所取代。

2. `str.format()` 方法

`str.format()` 引入了更加灵活的字符串格式化方式。使用大括号 `{}` 作为占位符，支持位置参数和关键字参数。例如：

```
name = "Bob"
score = 95.5
message = "Student: {} | Score: {:.2f}".format(name, score)
print(message)
```

这里，`{}` 占位符被 `name` 替换，而 `{:.2f}` 将 `score` 格式化为保留两位小数的浮点数。此外，`str.format()` 还支持通过位置参数和关键字参数进行更复杂的字符串格式化。

3. F 字符串格式化（`f-strings`）

Python 3.6 引入了 F 字符串格式化，这是目前推荐的格式化方式。它允许在字符串中直接嵌入变量和表达式，使代码更加简洁明了。例如：

```
name = "Eve"
gpa = 3.8
message = f"Student: {name} | GPA: {gpa:.2f}"
print(message)
```

该示例中，变量 `name` 和 `gpa` 直接嵌入到字符串中，并且可以通过 `{gpa:.2f}` 将 `gpa` 格式化为两位小数的浮点数。F 字符串不仅支持变量插值，还能嵌入复杂的表达式。

4. 综合应用示例

以下代码展示了如何在报告生成场景中使用 F 字符串和 `str.format()` 进行字符串格式化：

```
from datetime import datetime

# 使用str.format()格式化日期
current_date = datetime.now()
formatted_date = "Report generated on: {:%Y-%m-%d}".format(current_date)

# 使用F字符串生成报告内容
product = "Widget A"
sales = 100
profit = 20.567
report = f"Product: {product} | Sales: {sales} | Profit: ${profit:.2f}"

print(formatted_date)
print(report)
```

此示例首先使用 `str.format()` 格式化当前日期，然后使用 F 字符串将产品、销售额和利润嵌入报告中，并将利润格式化为两位小数。

5.5.2 str.format() 方法的位置参数和关键字参数

`str.format()` 方法可以通过位置参数和关键字参数来进行字符串格式化，灵活控制字符串的内容替换。

1. 位置参数

使用位置参数时，根据参数在 `format()` 方法中的顺序将值插入到字符串的占位符中，参数的顺序由大括号中的数字索引来决定。例如：

```
message = "Hello, {0}. You are {1} years old.".format("Alice", 25)
print(message)
```

输出结果为：

```
Hello, Alice. You are 25 years old.
```

在这个例子中，`{0}` 和 `{1}` 分别表示 `"Alice"` 和 `25` 两个位置参数。

2. 关键字参数

关键字参数允许通过名称引用参数值，这样使代码更加清晰。例如：

```
message = "Hello, {name}. You are {age} years old.".format(name="Bob", age=30)
print(message)
```

输出结果为：

```
Hello, Bob. You are 30 years old.
```

通过使用关键字参数 `name` 和 `age`，可以指定各自的值，使得格式化更加直观。

3. 混合使用位置参数和关键字参数可以混合使用位置参数和关键字参数，但要注意，位置参数必须在关键字参数之前。例如：

```
message = "Hello, {0}. Your balance is {balance}.".format("David", balance=230.23)
print(message)
```

输出结果为：

```
Hello, David. Your balance is 230.23.
```

这种方法结合了两种参数的优势，提供了更大的灵活性。

注意：当混合使用位置参数和关键字参数时，位置参数必须位于前面，否则会引发语法错误。

5.6 字符串的高级格式化设置

重要性：★★； **难易度**：★★★★

5.6.1 格式化迷你语言

在 Python 中，字符串的高级格式化功能为处理复杂的文本输出提供了强大的工具，尤其是在打印表格或整齐的输出时非常重要。主要方法包括 `str.format()` 和 F 字符串（f-strings），它们都支持 Python 的“格式化迷你语言”（formatting mini-language），允许对字符串进行精确控制，例如对齐、填充、宽度设定和精度设置。

Python 的格式化迷你语言是一套强大的工具，允许开发者在格式化字符串时精确控制输出。无论是 `str.format()` 还是 F 字符串（`f-strings`），都支持这种迷你语言，可以指定输出的宽度、对齐方式、数值格式等。

格式化迷你语言的通用格式为：

```
":[fill][align][sign][#][0][width][,][.precision][type]".format(value)
```

- `fill`：指定用于填充空白的字符，默认是空格。
- `align`：控制对齐方式，`<` 表示左对齐，`>` 表示右对齐，`^` 表示居中对齐。
- `sign`：用于数值的符号处理，`+` 表示始终显示正负号，`-` 表示仅对负数显示符号，空格则在正数前加空格。
- `width`：指定输出字段的最小宽度。
- `precision`：用于控制浮点数的小数位数或字符串的最大长度。
- `type`：定义数据类型，例如整数（`d`）、浮点数（`f`）、二进制（`b`）、十六进制（`x`）等。

1. 对齐和填充示例

通过设置 `fill` 和 `align` 可以灵活控制字符串的对齐和填充字符：

```
1 text = "Hello"
2 print(f"{text:<10}")   # 左对齐，宽度10
3 print(f"{text:^10}")   # 居中对齐，宽度10
4 print(f"{text:*>10}")  # 右对齐，宽度10，用'*'填充
```

2. 数值符号处理

`sign` 参数用于控制数字的符号显示。其基本语法包括三种选项：`+`、`-` 和 ` `（空格）。使用 `+` 时，无论数字为正或负，都会在前面加上正号；使用 `-` 时，仅在负数前加上负号，这是默认行为；而使用空格时，正数前会加一个空格以便与负数对齐。以下代码示例展示了这些用法：

```
1 print("{:+} {:+}".format(58, -58))  # 输出：+58 -58
2 print("{:-} {:-}".format(58, -58))  # 输出：58 -58
3 print("{: } {: }".format(58, -58))  # 输出：  58 -58
```

以上示例展示了如何使用 `sign` 参数进行数字格式化，从而使输出更加清晰与规范。

3. 参数 # 和 0

`#` 和 `0` 这两个参数用于控制数字的格式和输出样式。

参数用于指示在数字格式化时添加前缀。例如，当格式化为二进制、八进制或十六进制时，# 将会在结果前添加相应的前缀（如 `0b`、`0o`、`0x`）。例如：

```
print('{:#b}'.format(255))   # 输出：0b11111111
print('{:#o}'.format(255))   # 输出：0o377
print('{:#x}'.format(255))   # 输出：0xff
```

`0` 参数用于在数字前进行零填充，以达到指定的宽度。当使用 `0` 时，如果数字的位数不足以满足给定的宽度，将会在左侧补零。例如：

```
print('{:05}'.format(42))     # 输出：00042
print('{:02x}'.format(255))   # 输出：ff
print('{:#010b}'.format(255)) # 输出：0b000011111111
```

在上述示例中，使用 `:05` 表示输出的总宽度为 5 位，不足的部分用 0 填充；而 `:##010b` 则表示输出宽度为 10 位，并在二进制格式前添加 `0b` 前缀。

4. width 参数

`width` 参数用于定义字段的最小宽度。它通过指定整数值控制输出时每个字段的最小字符数，确保格式统一和对齐。`width` 的设置可以结合对齐方式和填充字符一起使用。

`width` 参数的格式如为 `"{:width}".format(value)`

这里的 `width` 为一个整数，表示最小字段宽度。例如，以下代码将输出带有最小宽度为 10 个字符的字符串：

```
print("{:10}".format("Hello"))
```

输出结果为：

```
Hello
```

`width` 参数通常与对齐符号一起使用。使用 `<`、`>`、`^` 符号分别表示左对齐、右对齐和居中对齐。

```
print("{:<10}".format("Left"))
print("{:>10}".format("Right"))
print("{:^10}".format("Center"))
```

还可以指定填充字符，默认情况下为空格。通过在对齐符号之前添加填充字符，可以填充剩余的空白。

```
print("{:*<10}".format("Fill"))
print("{:~^10}".format("Test"))
```

输出结果为：

```
Fill******
~~~Test~~~
```

`width` 参数在格式化数字时同样有效。例如，将数字格式化为至少 5 个字符宽，并右对齐：

```
print("{:5d}".format(42))
```

5. , 参数

逗号参数（`,`）用于对数字进行分组，以便增强可读性，尤其是在处理大数时非常有用。其作用是为数值添加千位分隔符。

```
number = 1234567890
print("{:,}".format(number))
```

输出结果为：

```
1,234,567,890
```

此示例中，逗号作为千位分隔符，使得输出更容易阅读。此功能不仅适用于整数，还可以与浮点数结合使用：

```
number = 1234567.89
print("{:,.2f}".format(number))
```

输出结果为：

```
1,234,567.89
```

在此例中，`:.2f` 控制保留两位小数，而逗号参数确保了千位分隔符的正确显示。此功能在会计和财务报表中非常有用，因为大数通常需要以这种方式展示。

6. precision 参数

`precision` 参数用于控制浮点数或字符串的精度。其基本形式为在格式说明符中加入点号后跟一个数字，如 `{:.2f}`。该数字表示需要显示的小数位数或字符串的最大字符数。

浮点数的精度控制：`precision` 常用于限制浮点数的小数位数。例如，以下代码将一个浮点数截取到小数点后两位：

```
pi = 3.141592653589793
print("Pi to two decimal places: {:.2f}".format(pi))
```

此处，`:.2f` 将浮点数 `pi` 格式化为两位小数。

字符串的精度控制：在处理字符串时，`precision` 参数用于限制最大字符数。例如：

```
text = "Python"
print("{:.3s}".format(text))
```

输出为：

```
Pyt
```

这里，`{:.3s}` 限制了字符串的长度为 3 个字符。

综合应用：可以将 `precision` 与其他格式化选项结合使用，如宽度、对齐等。例如：

```
num = 123.456789
print("{:8.3f}".format(num))
```

此代码不仅将浮点数限制为三位小数，还将结果对齐到总宽度为 8 的字段。

7. type 参数

`type` 参数用于指定如何格式化不同类型的数据，如整数、浮点数、字符串等。常见的类型代码包括：

整数格式化：`d`：将数字格式化为十进制整数。例如：

```
print("{:d}".format(123))  # 输出：123
```

浮点数格式化：`f`：将数字格式化为固定小数点形式，默认保留六位小数。例如：

```
print("{:.2f}".format(123.456789))  # 输出：123.46
```

科学计数法：`e` 或 `E`：将数字格式化为科学计数法，小写 `e` 或大写 `E` 表示指数。例如：

```
print("{:e}".format(1234567))  # 输出：1.234567e+06
```

进制格式化

`b`：将整数格式化为二进制。

`o`：将整数格式化为八进制。

`x` 或 `X`：将整数格式化为十六进制，`x` 为小写，`X` 为大写。例如：

```
print("{:x}".format(255))  # 输出：ff
```

字符串格式化：`s`：将数据格式化为字符串。例如：

```
print("{:s}".format("Hello"))  # 输出：Hello
```

百分比：%：将数字乘以 100 并显示为百分数。例如：

```
print("{:.2%}".format(0.25))  # 输出：25.00%
```

通过组合使用 `type` 参数和其他格式化选项（如宽度、精度），可以灵活地控制输出格式，适用于不同的商业应用场景，如财务数据的显示和报告生成。

5.6.2 输出字面上的% 和 {} 占位符

在 Python 字符串格式化中，输出字面上的占位符% 和大括号 {} 需要使用特定的转义方法。以下示例展示了如何实现这一功能。

```
# 使用%格式化输出字面上的%
value = 50
percent_string = "The success rate is %d%%." % value
print(percent_string)

# 使用{}格式化输出字面上的{}
name = "Alice"
braces_string = "Hello, {{name}}! Your score is {score:.1f}.".format(score=95.5)
print(braces_string)
```

输出结果：

```
The success rate is 50%.
Hello, {name}! Your score is 95.5.
```

解释：

1. **输出字面上的%**：在使用% 格式化时，双百分号%% 表示输出一个字面上的百分号。这里%d%% 中的第一个% 用于格式化整数，第二个% 用于显示字面值。

2. **输出字面上的 {}**：在使用 `format()` 方法时，双大括号 {{ }} 用于输出字面上的大括号。例如，{{name}} 将输出为 {name}，而不会被视为格式化占位符。

案例：格式化输出财务数据

在商业和财务分析中，格式化输出能够让数据更美观且易于理解。Python 提供了多种字符串格式化方法，可以用于生成简洁且有结构的财务报表或其他分析结果。下面的代码展示了如何使用 `str.format()` 方法进行格式化输出，用于生成一份财务报告，显示年度收入、支出和利润等关键数据。

```
# 定义财务数据
financial_data = [
{"year": 2020, "revenue": 9876543.21, "expenses": 5432109.87, "profit": 4444433.34},
{"year": 2021, "revenue": 12345678.90, "expenses": 6543210.12, "profit": 5802468.78}
]

# 输出表头
print(f"{'Year':<10}{'Revenue':>15}{'Expenses':>15}{'Profit':>15}")
print("="*55)

# 输出每年的财务数据，带有千位分隔符和两位小数
for data in financial_data:
    print(f"{data['year']:<10}{data['revenue']:>15,.2f}{data['expenses']:>15,.2f}{data['profit
    ']:>15,.2f}")
```

输出结果：

```
Year             Revenue         Expenses          Profit
=======================================================
2020          9,876,543.21      5,432,109.87      4,444,433.34
2021         12,345,678.90      6,543,210.12      5,802,468.78
```

解释：

1. **千位分隔符**：使用 `:,` 为数字添加千位分隔符，使得大数更易读。
2. **小数精度控制**：通过 `:.2f` 格式化浮点数，确保输出保留两位小数，常用于展示货币数值。
3. **对齐方式**：使用 `<` 和 `>` 控制左右对齐，保证列整齐排列。

这种格式化方式尤其适用于财务数据的展示，可以生成结构化的报告，如财务报表和年度报告等，确保数据的准确性和易读性。

5.7 string 模块

Python 的 `string` 模块提供了一系列用于处理字符串的常量和函数。该模块包含常用的字符集合，如字母、数字、标点符号等，简化了字符串操作。此外，`string` 模块还提供了诸如 `capwords()`、`translate()` 等实用函数，能够实现字符转换、格式化等功能，特别适合在数据处理和文本清理中使用。

5.7.1 常用常量

`string` 模块中，常量提供了一些预定义的字符集合，用于简化字符串处理。以下是一些常用常量及其基本用法。

```
import string

# 输出所有小写字母
print("小写字母:", string.ascii_lowercase)

# 输出所有大写字母
print("大写字母:", string.ascii_uppercase)

# 输出所有字母（包含大写和小写）
print("所有字母:", string.ascii_letters)

# 输出数字字符
print("数字字符:", string.digits)

# 输出标点符号
print("标点符号:", string.punctuation)
```

输出结果：

```
小写字母: abcdefghijklmnopqrstuvwxyz
大写字母: ABCDEFGHIJKLMNOPQRSTUVWXYZ
所有字母: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
数字字符: 0123456789
标点符号: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

解释：

1. `string.ascii_lowercase`：包含所有小写字母，从 `a` 到 `z`。
2. `string.ascii_uppercase`：包含所有大写字母，从 `A` 到 `Z`。
3. `string.ascii_letters`：包含所有字母，包括大写和小写。
4. `string.digits`：包含数字字符，从 `0` 到 `9`。
5. `string.punctuation`：包含所有常见的标点符号，如 `!`、`@`、# 等。

这些常量广泛应用于数据验证、密码生成等场景。通过使用这些预定义的字符集合，可以提高代码的可读性和效率，避免手动输入和维护大段字符。

5.7.2 translate 函数

`translate()` 函数用于基于一个翻译表（translation table）替换或移除字符串中的字符。该翻译表可以通过 `str.maketrans()` 方法创建，`translate()` 函数结合此表高效地执行字符映射操作。这个功能常用于数据清理或字符串替换等场景。

```
# 导入string模块
import string

# 创建一个翻译表，替换字符并移除特定字符
translation_table = str.maketrans("abc", "123", "d")

# 应用translate函数
text = "abcdef"
translated_text = text.translate(translation_table)

# 输出结果
print("原始文本:", text)
print("翻译后的文本:", translated_text)
```

输出结果：

```
原始文本: abcdef
翻译后的文本: 123ef
```

解释：

1. `str.maketrans("abc", "123", "d")` 创建了一个映射，其中 `a`、`b`、`c` 分别被替换为 `1`、`2`、`3`，同时字符 `d` 被移除。

2. `translate()` 函数将翻译表应用于字符串 `"abcdef"`，结果是 `"123ef"`，其中 `a`、`b`、`c` 被替换，`d` 被移除。

这种方法特别适合进行大规模的字符替换、清理输入数据，或是在文本处理中快速完成多字符的转换或移除操作。

案例：应用 string 模块清洗商业文本数据

在商业文本数据分析中，数据清洗是一个关键步骤，尤其是当数据来自多个源并且格式各异时。`string` 模块在文本清洗中非常有用，能够帮助有效地处理和规范化原始数据。以下示例展示了如何使用 `string` 模块来执行文本清洗操作，如移除标点符号和转换大小写。

```
import string

# 定义待清洗的文本
text = "Hello, World! This is a text with numbers (123) and punctuation!!!"

# 1. 移除标点符号
cleaned_text = text.translate(str.maketrans('', '', string.punctuation))

# 2. 转换为小写
lowercase_text = cleaned_text.lower()

# 3. 去除多余空格
final_text = " ".join(lowercase_text.split())

# 输出清洗后的文本
print("原始文本:", text)
print("清洗后的文本:", final_text)
```

输出结果：

```
原始文本: Hello, World! This is a text with numbers (123) and punctuation!!!
清洗后的文本: hello world this is a text with numbers 123 and punctuation
```

解释：

1. **移除标点符号**：通过 `str.maketrans()` 创建一个映射，将 `string.punctuation` 中的所有标点符号替换为空字符。
2. **转换为小写**：使用 `lower()` 方法将文本转换为小写，确保数据一致性，特别适用于关键词分析等任务。
3. **去除多余空格**：通过 `split()` 和 `join()` 组合移除多余的空格，确保文本格式统一。

这种方法适合处理原始文本数据，确保其在分析和建模阶段的质量。文本清洗是商业分析中的基础步骤，能帮助提升模型性能和结果的准确性。

5.8 特殊字符

重要性：★★★★； **难易度：**★★

在 Python 字符串处理过程中，特殊字符（special characters）是指那些不能直接表示或具有特殊含义的字符。为了在字符串中正确使用这些字符，通常需要使用转义字符（escape character）来避免语法错误或实现特定功能。转义字符以反斜杠（\）为前缀，后跟一个特定字符，来表示一个特殊的含义。常见的 Python 特殊字符和用法如表5.3所示。

1. **换行符 \n**: 用于在字符串中插入一个换行。

```
print("Hello\nWorld")
```

输出:

```
Hello
World
```

2. **制表符 \t**: 用于插入一个水平制表符。

```
print("Hello\tWorld")
```

输出:

```
Hello    World
```

3. **单引号 `'` 和双引号 `"`**: 当字符串使用单引号或双引号时，如果需要在字符串中包含相同类型的引号，需要使用转义字符。

```
print('It\'s a beautiful day')
print("He said, \"Python is awesome!\"")
```

输出:

```
It's a beautiful day
He said, "Python is awesome!"
```

4. **反斜杠 **: 用于表示一个实际的反斜杠，因为单个反斜杠在 Python 中是转义字符。

```
print("This is a backslash: \\")
```

输出:

```
This is a backslash: \
```

5. **回车符 \r 和退格符 \b**: \r 用于将光标移到行首，\b 则是退格符，删除前一个字符。

```
print("Hello\rWorld")  # 输出为 "Worldo"
print("Hello\b World")  # 输出为 "Hell World"
```

6. **原始字符串 `r` 或 `R`**: 在需要保留反斜杠的情况下，可以通过在字符串前加 `r` 或 `R`，使反斜杠不被解释为转义字符。

```
print(r"C:\new_folder\test.txt")
```

输出:

```
C:\new_folder\test.txt
```

```
# 使用转义字符打印带有引号的字符串
print("He said, \"Python is fun!\"")
# 输出: He said, "Python is fun!"

# 打印包含路径的字符串
print(r"C:\Users\username\Desktop")
# 输出: C:\Users\username\Desktop

# 使用换行符和制表符格式化输出
print("Name:\tJohn\nAge:\t25")
# 输出:
# Name:   John
# Age:    25
```

案例:处理客户反馈文本中的特殊字符(空白字符)

在数据分析过程中,Python 的转义字符(`escape characters`)常常用于处理文本数据,使数据更加整洁并便于进一步的分析。**转义字符通过反斜杠(\)引导,能够表示一些特殊符号或不可见字符,例如换行、制表符等**。在读取和清理数据时,通常需要处理文本中的特殊字符。例如,某些数据列可能包含不可见的换行符、制表符或其他影响分析的符号。以下代码展示了如何使用 Python 转义字符对读取的数据进行清理:

```
import pandas as pd

# 模拟从CSV读取的数据, 包含换行符和制表符
data = {'Comments': ['Great service!\nThank you', 'Price:\t$5,000', 'Contact: support@example.com\n'
    ]}
df = pd.DataFrame(data)

# 使用转义字符 \n 和 \t 进行文本处理
df['Cleaned_Comments'] = df['Comments'].str.replace('\n', ' ').str.replace('\t', ' ')

print(df)
```

在此代码中,`replace`方法用于替换文本中的换行符(\n)和制表符(\t),将其转换为空格。这有助于在数据分析前保持数据的一致性和整洁。转义字符在处理文本数据、输出格式化、或者处理文件路径时非常有用。例如,在 Windows 路径中,反斜杠(\)常作为文件分隔符,因此需要使用双反斜杠(\\)来表示。

5.9 正则表达式

重要性:★★★★; **难易度:**★★★★

正则表达式(Regular Expression,简称 RegEx)是一种用于匹配文本模式的特殊字符序列。通过定义特定的模式,正则表达式能够高效地查找、匹配和操作字符串。Python 的内置模块`re`提供了强大的正则表达式功能,用于执行模式匹配操作,例如搜索、替换和分割字符串。

5.9.1 基本语法

1. **普通字符**: 直接与字符串中的相同字符匹配。例如，`"abc"` 匹配字符串中的 `"abc"`。

2. **元字符**:

- `.`: 匹配除换行符以外的任意一个字符。如，`a.b` 匹配 `"a1b"`、`"acb"`，但不匹配 `"ab"`。

- `^`: 匹配字符串的开始部分。如，`^abc` 匹配以 `"abc"` 开头的字符串。

- `$`: 匹配字符串的结尾。如，`abc$` 匹配以 `"abc"` 结尾的字符串。

- `[]`: 字符集，匹配方括号中的任意一个字符。如，`[a-z]` 匹配任意小写字母。

- `|`: 或运算符，匹配符号两侧的任意一个模式。如，`a|b` 匹配 `"a"` 或 `"b"`。

- `()`:用于分组，允许将多个字符视为一个整体。如，`(abc|def)` 匹配 `"abc"` 或 `"def"`。

3. **量词**:

- `*`: 匹配前一个字符的零次或多次出现。如，`a*` 可以匹配 `"a"`、`aa"`、`aaa"`，或不包含 `"a"` 的字符串。

- `+`: 匹配前一个字符的一次或多次出现。如，`a+` 匹配至少一个 `"a"`。

- `?`: 匹配前一个字符的零次或一次出现。如，`a?` 匹配 `"a"` 或不包含 `"a"` 的字符串。

- `{n,m}`: 匹配前一个字符至少 n 次，至多 m 次的出现。例如，`a{2,3}` 匹配 `"aa"` 或 `"aaa"`，但不匹配单个 `"a"`。

4. **特殊字符**:

- `\d` : 匹配任何一个数字，相当于 `[0-9]`。

- `\w` : 匹配任何一个字母、数字或下划线，相当于 `[a-zA-Z0-9_]`。

- `\s` : 匹配任何一个空白字符，包括空格、制表符等。

- `\b` 匹配一个单词的开头或结尾，但不消耗任何字符，即它不会匹配实际的字符，而是检查当前位置是否处于单词和非单词字符之间的边界。单词字符包括字母、数字和下划线（相当于 `\w`，即 `[a-zA-Z0-9_]`），非单词字符包括空格、标点符号等（即不属于单词字符的内容）。`\bfoo` 匹配以 `foo` 开头的单词，如”foo” 在”foo bar” 中会被匹配。`foo\b` 匹配以 `foo` 结尾的单词，如”foo” 在”the foo” 中会被匹配。`\bfoo \b` 匹配完全独立的”foo”，即它必须是一个完整的单词，不是其他单词的一部分。

5. **常用函数**：

在 Python 的 `re` 模块中，几个常用的函数提供了强大的正则表达式支持，用于处理文本匹配、查找和替换。以下是一些常用函数的基本语法及其使用示例：

- `re.match()` 函数尝试从字符串的开头匹配一个模式。如果匹配成功，则返回一个匹配对象，否则返回 `None`。

```python
import re
pattern = r"\d+"  # 匹配一个或多个数字
result = re.match(pattern, "123abc")
print(result.group())  # 输出: 123
```

该函数只在字符串的开头尝试匹配，因此如果模式在字符串中部出现，它不会成功。

- `re.search()` 函数在整个字符串中搜索第一个匹配项，而不局限于开头。

```python
import re
pattern = r"\d+"  # 匹配一个或多个数字
result = re.search(pattern, "abc123xyz")
print(result.group())  # 输出：123
```

即使数字出现在字符串的中间，`re.search()` 仍能找到第一个匹配项。

- `re.findall()` 会返回所有与模式匹配的子串，结果为一个列表。它与 `re.search()` 不同，后者只返回第一个匹配项。

```python
import re
pattern = r"\d+"  # 匹配一个或多个数字
result = re.findall(pattern, "123abc456def")
print(result)  # 输出：['123', '456']
```

该函数非常适合用于提取多个匹配项的场景。

- `re.sub()` 用于替换字符串中匹配模式的部分，返回替换后的新字符串。

```python
import re
pattern = r"\d+"  # 匹配一个或多个数字
result = re.sub(pattern, "NUMBER", "123abc456def")
print(result)  # 输出：NUMBERabcNUMBERdef
```

`re.sub()` 可以将所有匹配项替换为指定的字符串，常用于文本清理。

- `re.split()` 根据模式分割字符串，返回一个子字符串列表。

```python
import re
pattern = r"\d+"  # 使用数字作为分隔符
result = re.split(pattern, "abc123def456ghi")
print(result)  # 输出：['abc', 'def', 'ghi']
```

它将匹配的部分作为分隔符，非常适合按规则分割字符串。

5.9.2 代码示例

以下示例展示了如何使用 Python 正则表达式进行简单的字符串匹配：

示例 1：提取字符串中的所有数字

使用 `re.findall()` 从字符串中提取所有数字。

```python
import re

# 定义匹配一个或多个数字的正则表达式
pattern = r"\d+"
text = "订单号是12345，金额为678.90元"

# 查找所有匹配的数字
matches = re.findall(pattern, text)
print(matches)  # 输出：['12345', '678', '90']
```

解释：`\d +` 匹配一个或多个连续的数字字符，`re.findall()` 函数会返回所有匹配的子字符串。

示例 2:匹配字符串的开头和结尾

使用 `re.search()` 检查字符串是否以"The" 开头并以"Spain" 结尾。

```
import re

# 定义模式，匹配以 "The" 开头并以 "Spain" 结尾的字符串
pattern = r"^The.*Spain$"
text = "The rain in Spain"

# 查找匹配项
match = re.search(pattern, text)

if match:
    print("匹配成功")
else:
    print("匹配失败")
```

解释: `^The` 表示字符串必须以"The" 开头，`.*` 表示任意数量的字符（除换行符外），`Spain\$` 表示字符串必须以"Spain" 结尾。

示例 3:替换字符串中的所有空白字符

使用 `re.sub()` 将字符串中的所有空白字符替换为其他内容。

```
import re

# 匹配所有空白字符（包括空格、制表符等）
pattern = r"\s+"
text = "abc 12   de 23 \n f45 6"

# 将所有空白字符替换为空字符串
new_text = re.sub(pattern, '', text)
print(new_text)  # 输出: 'abc12de23f456'
```

解释: `\s +` 匹配一个或多个空白字符，`re.sub()` 用于将所有匹配的部分替换为指定的字符串。

示例 4:分割字符串

使用 `re.split()` 依据正则表达式分割字符串。

```
import re

# 定义模式，匹配数字
pattern = r"\d+"
text = "订单号:12345，金额:678.90"

# 使用数字作为分隔符
result = re.split(pattern, text)
print(result)  # 输出: ['订单号:', '，金额:', '.']
```

解释: `re.split()` 使用正则表达式匹配的位置将字符串分割为多个部分。`\d +` 匹配一个或多个数字字符。

示例 5:验证电子邮件格式

使用 `re.match()` 来验证输入的字符串是否符合电子邮件格式。

```python
import re

# 定义匹配电子邮件地址的正则表达式
pattern = r"^[\w\.-]+@[\w\.-]+\.\w+$"
email = "[email protected]"

# 检查字符串是否匹配电子邮件格式
if re.match(pattern, email):
    print("有效的电子邮件地址")
else:
    print("无效的电子邮件地址")
```

解释: `^[\w \.-]+` 匹配电子邮件地址的用户名部分，`@` 匹配'@' 符号，`[\w \.-]+` 匹配域名部分，`\.\w +$` 匹配顶级域名。

示例 6:从字符串中提取年份

使用 `re.search()` 提取字符串中的四位数年份。

```python
import re

# 定义匹配四位数年份的正则表达式
pattern = r"\b\d{4}\b"
text = "这本书出版于1995年"

# 查找匹配的年份
match = re.search(pattern, text)
if match:
    print("找到年份:", match.group())
```

解释: `\b \d {4}\b` 匹配恰好为四位数的年份,其中 `\b` 确保匹配的数字是独立的词。

示例 7:匹配电话号码格式

使用 `re.findall()` 提取符合电话号码格式的内容。

```python
import re

# 匹配形如 (123) 456-7890 的电话号码
pattern = r"\(\d{3}\)\s\d{3}-\d{4}"
text = "我的号码是 (123) 456-7890"

# 查找所有匹配的电话号码
matches = re.findall(pattern, text)
print(matches)  # 输出: ['(123) 456-7890']
```

解释: `\(\d {3}\)\s \d {3}-\d {4}` 匹配标准的美国电话号码格式,其中 `\d {3}` 匹配三位数字，`\s` 匹配空格。

示例 8:替换文本中的单词

使用 `re.sub()` 替换文本中的指定单词。

```python
import re

```

```
# 替换 "bad" 为 "good"
pattern = r"\bbad\b"
text = "This is a bad example."
new_text = re.sub(pattern, "good", text)
print(new_text)  # 输出: "This is a good example."
```

解释: `\bbad \b` 匹配独立的单词"bad",使用 `re.sub()` 将其替换为"good"。

5.9.3 应用示例

在电子商务文本数据分析中,正则表达式(Regular Expressions, 简称 RegEx)是一种高效的文本处理工具,常用于提取、验证和清理数据。以下是一些简单的正则表达式应用示例代码,结合电子商务场景展示了其实际应用。

1. 提取电子邮件地址

在客户数据或市场营销数据处理中,可能需要从文本中提取电子邮件地址。以下代码使用正则表达式提取电子邮件:

```
import re

# 定义匹配电子邮件的正则表达式
pattern = r'[\w\.-]+@[\w\.-]+\.\w+'

# 需要处理的文本数据
text = "客户联系: [email protected], [email protected]"

# 使用findall提取所有匹配的电子邮件
emails = re.findall(pattern, text)
print(emails)  # 输出 ['[email protected]', '[email protected]']
```

该代码通过 `re.findall()` 函数匹配所有符合邮箱格式的字符串,非常适合用于处理大规模客户信息。

2. 提取产品价格

在电子商务网站爬取数据时,提取产品价格是常见需求。以下代码展示了如何使用正则表达式提取价格信息:

```
import re

# 匹配价格的正则表达式(形如$99.99)
pattern = r'\$\d+\.\d{2}'

# 示例文本,包含多个价格信息
text = "商品价格为$29.99,另一个商品售价为$49.99。"

# 使用findall提取价格
prices = re.findall(pattern, text)
print(prices)  # 输出 ['$29.99', '$49.99']
```

3. 订单号提取

从电子商务交易记录中提取订单号也是常见的应用场景。使用如下代码可以匹配并提取订单号：

```
import re

# 定义匹配订单号的正则表达式
pattern = r'Order\s+\d+'

# 示例交易记录
text = "订单详情：Order 12345 已成功提交。"

# 提取订单号
order_numbers = re.findall(pattern, text)
print(order_numbers)  # 输出 ['Order 12345']
```

4. 替换文本中的关键词

在处理客户评论或产品描述时，可能需要将某些敏感词替换为其它词。以下代码展示如何使用正则表达式替换文本中的某个关键词：

```
import re

# 替换"apple"为"orange"
pattern = r'\bapple\b'
text = "I like apple. Apple is healthy."
new_text = re.sub(pattern, 'orange', text, flags=re.IGNORECASE)
print(new_text)  # 输出 "I like orange. orange is healthy."
```

案例：社交媒体文本清理和信息提取

在社交媒体帖文分析中，正则表达式（regex）是一个强大的工具，能够高效地清理和提取非结构化的文本数据，特别是从帖文、评论等文本中提取关键信息。以下是一个结合社交媒体数据分析的代码示例，展示如何使用 Python 正则表达式进行多种操作，包括提取帖文中的标签、用户提及、链接以及清理特殊字符。

```
import re

# 示例帖文
tweet = "Check out our new product! #innovation @user https://t.co/example"

# 1. 提取所有的标签（如 #innovation）
hashtags = re.findall(r"#\w+", tweet)
print("Hashtags:", hashtags)  # 输出：['#innovation']

# 2. 提取所有的用户提及（如 @user）
mentions = re.findall(r"@\w+", tweet)
print("Mentions:", mentions)  # 输出：['@user']

# 3. 提取所有的链接（如 https://t.co/example）
urls = re.findall(r"https?://[^\s]+", tweet)
print("URLs:", urls)  # 输出：['https://t.co/example']

# 4. 清理特殊字符，仅保留文本内容
cleaned_tweet = re.sub(r"[@#]\w+|https?://[^\s]+", "", tweet).strip()
print("Cleaned Tweet:", cleaned_tweet)  # 输出：'Check out our new product!'
```

代码说明

- **提取标签**：正则表达式 `#\w+` 匹配所有以 # 开头的单词，这通常表示帖文中的话题标签（Hashtags）。
- **提取用户提及**：`@\w+` 匹配所有以 `@` 开头的单词，这代表帖文中的用户提及（Mentions）。
- **提取链接**：`https?://[^\s]+` 用于匹配 HTTP 或 HTTPS 开头的 URL 链接。
- **清理帖文**：通过 `re.sub()` 函数，使用正则表达式删除标签、提及和链接，只保留帖文的主体内容。

在社交媒体分析中，正则表达式常用于从文本中提取结构化数据，比如标签和用户提及，这些数据可以进一步用于话题跟踪、用户影响力分析等。正则表达式还可以用来清理噪音数据，如移除 URL 和特殊字符，使文本更适合用于情感分析和话题建模等自然语言处理任务。

表 5.1: Python 字符串常用方法

方法名	描述	用法示例
`capitalize()`	将字符串的首字母转换为大写	`"hello".capitalize()` → `"Hello"`
`casefold()`	转换为小写，适合进行不区分大小写的比较	`"HELLO".casefold()` → `"hello"`
`center(width)`	将字符串居中，使用指定的字符填充	`"abc".center(5, "-")` → `"-abc-"`
`count(substring)`	返回子字符串出现的次数	`"hello".count("l")` → `2`
`find(substring)`	返回子字符串的首个索引，不存在时返回 -1	`"hello".find("e")` → `1`
`format()`	格式化字符串，允许多种格式化选项	`"Hello, {}".format("World")` → `"Hello, World"`
`index(substring)`	返回子字符串的首个索引，若不存在则引发异常	`"hello".index("z")` → `ValueError`
`isalnum()`	检查字符串是否只包含字母和数字	`"abc123".isalnum()` → `True`
`isalpha()`	检查字符串是否只包含字母	`"abc".isalpha()` → `True`
`isdigit()`	检查字符串是否只包含数字	`"123".isdigit()` → `True`
`join(iterable)`	将可迭代对象中的元素连接为字符串	`",".join(["a", "b", "c"])` → `"a,b,c"`
`lower()`	将字符串转换为小写	`"HELLO".lower()` → `"hello"`
`replace(old, new)`	替换字符串中的子串	`"hello".replace("l", "x")` → `"hexxo"`
`split(separator)`	将字符串分割为列表，默认以空格为分隔符	`"a b c".split()` → `["a", "b", "c"]`
`strip()`	删除字符串两端的空白字符	`" hello ".strip()` → `"hello"`
`upper()`	将字符串转换为大写	`"hello".upper()` → `"HELLO"`

表 5.2: 常见的字符串内容类型验证方法

方法	含义	示例代码	输出
`isalnum()`	判断字符串是否只包含字母和数字	`"Hello123".isalnum()`	`True`
`isalpha()`	判断字符串是否只包含字母	`"Hello".isalpha()`	`True`
`isdigit()`	判断字符串是否只包含数字	`"12345".isdigit()`	`True`
`isdecimal()`	判断字符串是否只包含十进制字符	`"12345".isdecimal()`	`True`
`islower()`	判断字符串是否全为小写字母	`"hello".islower()`	`True`
`isupper()`	判断字符串是否全为大写字母	`"HELLO".isupper()`	`True`
`istitle()`	判断字符串是否每个单词首字母大写）	`"Hello World".istitle()`	`True`
`isspace()`	判断字符串是否只包含空白字符	`" ".isspace()`	`True`

表 5.3: 常见的 Python 转义字符及其用法

转义字符	**含义**	**用法示例**
`\\`	反斜杠	`print("C:\\Users\\Path")` 输出为 `C:\Users\Path`
`\'`	单引号	`print('It\'s a test')` 输出为 `It's a test`
`\"`	双引号	`print("She said, \"Hello\"")` 输出为 `She said, "Hello"`
`\n`	换行	`print("Line1\nLine2")` 输出为两行: `Line1` 和 `Line2`
`\t`	制表符	`print("A\tB")` 输出为 `A B`（插入一个水平制表符）
`\b`	退格	`print("ABC\bD")` 输出为 `ABD`（删除 C）
`\r`	回车	`print("Hello\rWorld")` 输出为 `World`（光标回到行首并覆盖）
`\v`	垂直制表符	`print("A\vB")` 输出为 `A` 和 `B` 分别位于两行,前面带有垂直制表符