

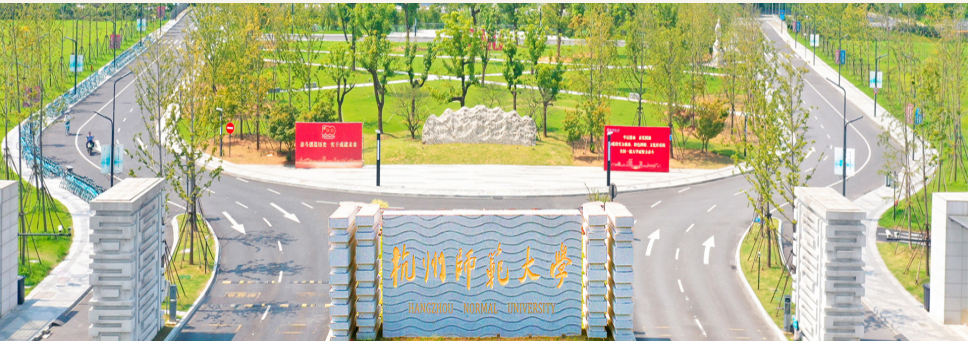
第四章 - 字符串 (编程练习 1-2)

张建章

阿里巴巴商学院

杭州师范大学

2025-09



- 1 创建字符串
- 2 字符串的基本操作
- 3 字符串的常用方法
- 4 string模块
- 5 特殊字符
- 6 字符串格式化
- 7 字符串的高级格式化设置 (了解, 浅考)
- 8 正则表达式 (自学-不考)

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。这些操作能够简化对大规模文本的分析和处理。

字符串是一种**不可变的字符序列**，创建字符串的基本语法很简单，主要通过以下几种方式实现 (代码示例):

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. `str()`

使用 `str()` 函数创建空字符串或将其他数据类型转换为字符串。

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。

字符串的基本操作与列表的基本操作非常相似，包括：通过索引访问字符、切片提取子字符串、计算长度、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。

字符串的基本操作的代码示例。

1. 索引

假设有一个字符串 `s = "Python"`，可以通过索引访问各个字符：

```
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)
```

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。

2. 切片

一种从字符串中提取子字符串的方式，**语法同列表切片**。

```
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)
```

3. 计算长度

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。

```
# 定义一个字符串
feedback = "Customer service was excellent!"
# 计算字符串的长度
length = len(feedback)
# 输出结果
print(length) # 输出: 31
```

4. 成员资格检查：通过 `in` 和 `not in` 运算符完成，用于检测子字符串是否存在于给定的字符串中。

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


5. 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。

```
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, 在处理大量字符串时节省内存:

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

6. 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: =====
```

字符串的常用方法 (代码示例)

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()` 会按照空白字符分隔字符串。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。

3. 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。

```
text = "Revenue for the year is estimated at $5 million."  
position = text.find("estimated")  
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
```

`start` 和 `end` 参数含义同字符串切片：

```
# 使用起始参数
phrase = "Python is great. Python is versatile."
position = phrase.index("Python", 10)
print(position) # 输出: 17
# 使用结束参数
phrase = "Python is great. Python is versatile."
position = phrase.index("le", 10, 35) # ValueError
position = phrase.index("le", 10, 36)
print(position) # 输出: 34
```

`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。

3. 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。

4. 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，尤其是在标准化和数据清洗。

`upper()` 方法将字符串中的所有字母转换为大写：

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

`lower()` 方法用于将字符串中的所有字母转换为小写：

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

`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：

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

`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：

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

5. 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。

`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。

`rstrip()` 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.rstrip()
print(left_trimmed_text)
# 输出: "Python is great!  "
```

`lstrip()` 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.lstrip()
print(right_trimmed_text)
# 输出: "  Python is great!"
```

6. 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。

`start` 和 `end` 参数含义同字符串切片。

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  
count = string.count(substring, 8, 26)  
print("The count is:", count) # 输出: 2
```

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。

7. 类型验证方法

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`）。如下表

表 1: 常见的字符串内容类型验证方法

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

字符串还有许多实用的常用方法，参见讲义中的表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。

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，特别适合在文本清理中使用。(代码示例)

`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)
print("数字字符:", string.digits) # 输出数字字符
# 输出标点符号
print("标点符号:", string.punctuation)
```


`translate()` 函数用于基于一个翻译表（translation table）替换或删除字符串中的字符。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)
```

Python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特殊字符和用法如下所示。（代码示例）

表 2: 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	<code>print("C:\\Users\\Path")</code> 输出为 <code>C:\Users\Path</code>
\'	单引号	<code>print('It\'s a test')</code> 输出为 <code>It's a test</code>
\"	双引号	<code>print("She said, \"Hello\"")</code> 输出为 <code>She said, "Hello"</code>
\n	换行	<code>print("Line1\nLine2")</code> 输出为两行: <code>Line1</code> 和 <code>Line2</code>
\t	制表符	<code>print("A\tB")</code> 输出为 <code>A B</code> （插入一个水平制表符）
\b	退格	<code>print("ABC\bD")</code> 输出为 <code>ABD</code> （删除 <code>C</code> ）
\r	回车	<code>print("Hello\rWorld")</code> 输出为 <code>World</code> （光标回到行首并覆盖）
\v	垂直制表符	<code>print("A\vB")</code> 输出为 <code>A</code> 和 <code>B</code> 分别位于两行,前面带有垂直制表符

1. 换行符 `\n`: 用于在字符串中插入一个换行。

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

2. 制表符 `\t`: 用于插入一个水平制表符。

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

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!\"")
```

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: \\")
```

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")
```

```
# 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
```

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，以及最新的F字符串格式化 (`f-strings`)。([代码示例](#))

1. 百分号格式化

Python最早的字符串格式化方式，使用%符号作为占位符。例如：

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

上例中，%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)
```

`{}` 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。

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)
```

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。

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)
```

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，后两个%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。

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)
```

上例中，`{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)
```

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。

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)
```

(代码示例)

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 {填充内容:格式参数}, 全部格式参数及其顺序如下:

```
[fill][align][sign][#][0][width][,][.precision][type]
```

常用的几个参数定义如下:

- **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:

```
text = "Hello"  
print(f"{text:<10}")    # 左对齐, 宽度10  
print(f"{text:^10}")    # 居中对齐, 宽度10  
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, 都会在前面加上正负号; 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:

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

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)。#通常和type参数一起使用, 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('{:#012b}'.format(255)) # 输出: 0b0011111111
```

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"))
```

`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"))
```

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。其作用是数值添加千位分隔符。

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

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))
```

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, 采用四舍五入。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))
```

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。

字符串格式化：**s**：将数据格式化为字符串。

百分比：**%**：将数字乘以100并显示为百分数。

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

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。



真的很强, 课时有限, 请参阅讲义(第5.9节)进行自学

THE END