

# Python 快速入门

本章主要介绍了 Python 的基本操作,包括如何使用交互式解释器进行简单的编程实践。内容涵盖表达式的算术运算、变量与赋值操作的理解,以及字符串处理、函数调用和模块导入等核心概念,这些内容为编写程序提供了基础技能。

## 2.1 Python 交互式解释器

重要性:★★; 难易度:★

Python 交互式解释器是 Python 语言的一个核心工具,它允许用户在命令行中直接输入 Python 代码,并即时得到执行结果。这种解释器以 `>>>` 为提示符,用户可以输入一行或多行代码,逐步执行和测试代码,其缺点是不能保存结果或包含丰富的文本说明。主要特点包括:

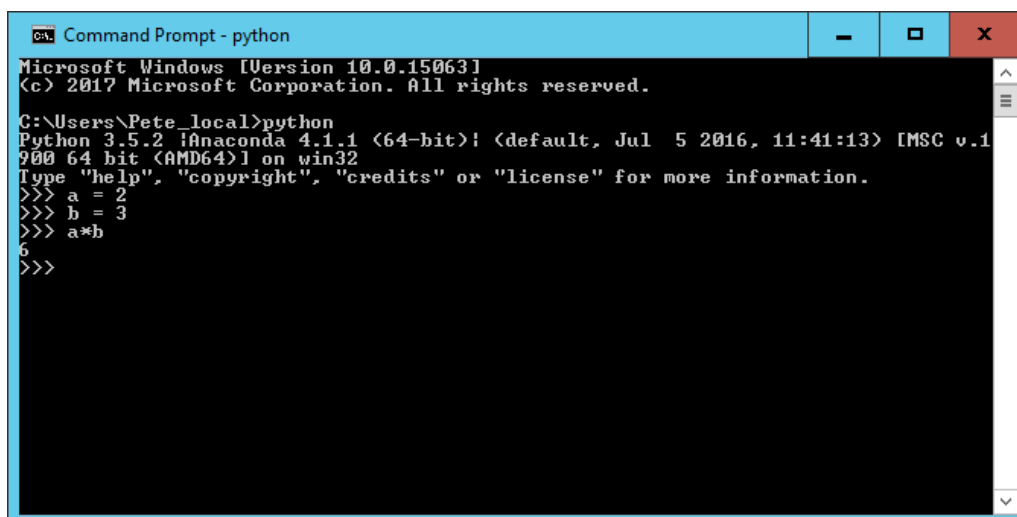
- **即时反馈:**每条语句在输入后立即执行并返回结果,非常适合调试和快速原型开发。
- **轻量简洁:**无需图形界面,只需通过终端(Windows 的 Command Prompt、Linux 的 Terminal 或 Mac 的 Terminal)即可运行,适合简单的计算或脚本运行。

### 2.1.1 使用 Python 交互式解释器

在 Windows、Linux 和 macOS 三种操作系统中,使用 Python 的原生交互式解释器(REPL)方式略有不同,以下是详细说明:

#### 1. Windows

- I. 打开命令提示符的快捷键:按下 `Win + R`,输入 `cmd`,然后按 `Enter`。
- II. 在命令提示符或 PowerShell 中,输入 `python` 或 `py` 进入 Python 解释器,显示 `>>>` 提示符后可以开始输入代码。
- III. 退出解释器的方法是输入 `exit()` 或按 `Ctrl + Z` 然后按 `Enter`。



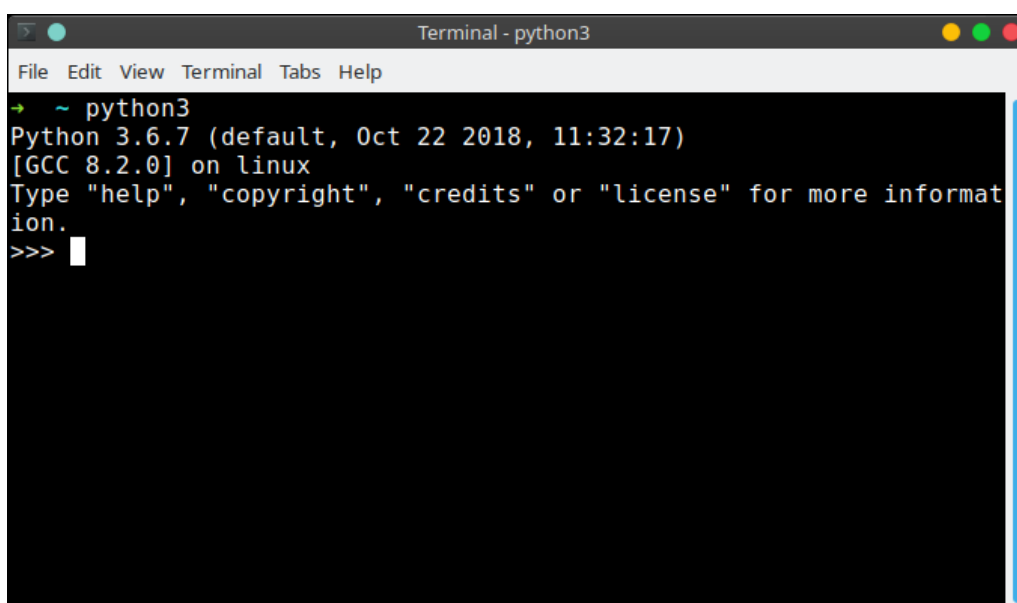
```
Command Prompt - python
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\Pete_local>python
Python 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul 5 2016, 11:41:13) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> a = 2
>>> b = 3
>>> a*b
6
>>>
```

图 2.1: Windows 系统中启动 Python 交互式解释器

## 2. Linux

- I. 打开终端的快捷键:通常为 `Ctrl + Alt + T`。
- II. 在终端中,输入 `python3` 进入 Python 3 解释器,显示 `>>>` 提示符后即可进行交互式编程。



```
Terminal - python3
File Edit View Terminal Tabs Help
→ ~ python3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

图 2.2: linux 系统中启动 Python 交互式解释器

- III. 退出解释器的方法是输入 `exit()` 或使用快捷键 `Ctrl + D`。

## 3. Mac OS

- I. 打开终端的快捷键:按 `Command + Space` 打开 Spotlight 搜索,输入 `Terminal`,然后按 `Enter`。
- II. 在终端中,输入 `python3` 启动 Python 3 解释器,显示 `>>>` 提示符后即可使用 Python。

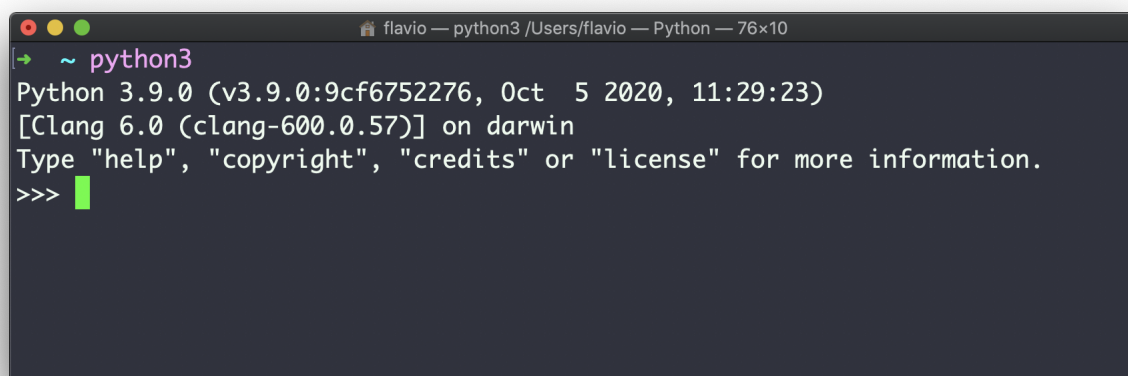
A screenshot of a macOS terminal window. The title bar at the top reads 'flavio — python3 /Users/flavio — Python — 76x10'. The terminal content shows the command '~ python3' being executed, followed by the output: 'Python 3.9.0 (v3.9.0:9cf6752276, Oct 5 2020, 11:29:23) [Clang 6.0 (clang-600.0.57)] on darwin'. Below this, it says 'Type "help", "copyright", "credits" or "license" for more information.' and the prompt '>>>' with a green cursor.

图 2.3: mac OS 系统中启动 Python 交互式解释器

III. 退出方式为输入 `exit()` 或快捷键 `Ctrl + D`。

## 2.2 Jupyter Notebook

重要性:★★★★★; 难易度:★

Jupyter Notebook 是基于 IPython 的扩展,它提供了一个更加用户友好的界面,用于管理和展示 Python 代码的执行结果。Python 交互式解释器最初是 IPython 项目的一部分,随着 Jupyter 项目的独立发展,IPython 继续作为 Python 的命令行解释器存在,而 Jupyter Notebook 则演变为一个强大的多语言交互计算平台。

Jupyter Notebook 是一个基于网页的互动式开发环境,允许用户在同一个文档中编写和执行代码、记录文本说明、显示图形和数据结果。它在数据科学和教学领域应用广泛。其主要特点包括:

- **混合文本与代码:**支持将 Python 代码与 Markdown 格式的文本、图片、数学公式等混合在一起,便于编写说明文档或生成可视化报告。
- **非线性执行:**Jupyter Notebook 的代码分为多个单元格 (cells),每个单元格可以独立执行,不要求按顺序执行。这对于数据分析、实验和反复测试代码片段非常实用。
- **持久的输出:**执行代码后,结果直接保存在单元格下方,便于浏览和复查。

### 2.2.1 使用 Jupyter Notebook

安装 Anaconda 后,打开 Anaconda,从中启动 Jupyter Notebook,参见1.3.1,使用示例如图2.4所示。

## 2.3 表达式

重要性:★★★★★; 难易度:★★

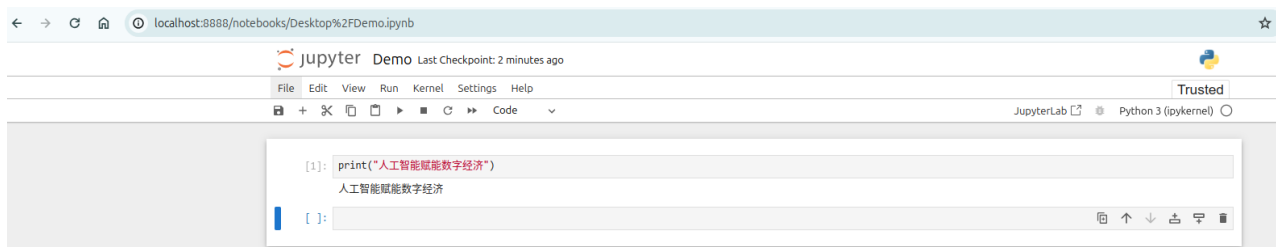


图 2.4: Jupyter Notebook 使用示例

在 Python 中,表达式 ( Expression ) 是由操作符和操作数组成的代码单元,用于计算和返回一个值。表达式可以包含常量、变量、函数调用和操作符等。根据使用的操作符类型,表达式可以分为不同的类别。

#### </> 代码 2.1: Python 表达式

```
1 x = 3 + 5 # 3 + 5 是一个表达式, 结果为 8
2 y = abs(-7) # abs(-7) 是一个函数调用表达式, 结果为 7
```

上面例子中, `3 + 5` 是一个算术表达式,使用了加法操作符 `+`,而 `abs(-7)` 是一个调用内置函数 `abs()` 的表达式。

### 2.3.1 算数表达式

算术表达式涉及数字和算术操作符 (如 `+`、`-`、`*`、`/` 等),用于执行数学运算。例如,表达式 `5 * 3 - 2` 将首先计算乘法,结果为 `13`。这些表达式返回数值结果,并遵循标准的操作符优先级。

算术操作符用于执行数学计算,每个操作符具有不同的优先级,决定了在复杂表达式中各操作的执行顺序。下表列出了常用的 Python 算术操作符、其含义、优先级及相应的示例。

表 2.1: Python 常用算术操作符

操作符	含义	优先级	示例	结果
<code>()</code>	括号 (最高优先级)	最高	<code>(3 + 2) * 4</code>	<code>20</code>
<code>**</code>	幂运算	高	<code>2 ** 3</code>	<code>8</code>
<code>*</code>	乘法	中	<code>3 * 4</code>	<code>12</code>
<code>/</code>	除法	中	<code>10 / 2</code>	<code>5.0</code>
<code>//</code>	整数除法	中	<code>10 // 3</code>	<code>3</code>
<code>\%</code>	取余	中	<code>10 \% 3</code>	<code>1</code>
<code>+</code>	加法	低	<code>3 + 5</code>	<code>8</code>
<code>-</code>	减法	低	<code>10 - 4</code>	<code>6</code>

Python 中的操作符优先级遵循数学中的规则。例如,乘法和除法的优先级高于加法和减法,这意味着在表达式 `3 + 2 * 4` 中,乘法 `2 * 4` 将先执行,结果为 `8`,然后进行加法,最终结果为 `11`。如果希望改变这种顺序,可以使用括号,例如 `(3 + 2) * 4`,结果为 `20`。

</> 代码 2.2: 算数表达式练习题

```
1 # 涉及括号、指数运算、整数除法、取余和多级嵌套运算
2 result = (3 + 5 * 2) ** 2 // (4 % 3 + 2 * (5 - 3))
3 print(result)
4
5 # 该表达式包含多个优先级相同的运算符，需要理解从左到右的关联性
6 result = 5 ** 3 // 7 % 4 + 6 * (7 // 2)
7 print(result)
8
9 # 该表达式包含多个不同优先级的操作符，需要掌握优先级表并正确使用括号
10 result = (2 ** 3) * (3 // 2 + 4 % 3) ** 2 - 5 / 2 + 7
11 print(result)
```

2.3.2 关系表达式

关系表达式 (Relational Expressions) 用于比较两个操作数,并返回一个布尔值,即 `True` 或 `False`。这些表达式常用于条件判断和循环控制。Python 中的关系操作符包括: `==` (等于)、`!=` (不等于)、`>` (大于)、`<` (小于)、`>=` (大于或等于) 以及 `<=` (小于或等于)。

表 2.2: Python 常用关系操作符

操作符	含义	示例	结果
<code>==</code>	等于	<code>5 == 5</code>	<code>True</code>
<code>!=</code>	不等于	<code>5 != 3</code>	<code>True</code>
<code>&gt;</code>	大于	<code>10 &gt; 5</code>	<code>True</code>
<code>&lt;</code>	小于	<code>3 &lt; 7</code>	<code>True</code>
<code>&gt;=</code>	大于或等于	<code>4 &gt;= 4</code>	<code>True</code>
<code>&lt;=</code>	小于或等于	<code>6 &lt;= 8</code>	<code>True</code>

2.3.3 逻辑表达式

逻辑操作符用于对布尔表达式进行运算,最终返回 `True` 或 `False`。这些操作符包括 `and`、`or` 和 `not`,它们按照特定的优先级进行运算。

表 2.3: Python 中的逻辑操作符

操作符	含义	优先级	示例	结果
<code>and</code>	逻辑与:仅当所有操作数为 <code>True</code> 时返回 <code>True</code>	中等	<code>True and False</code>	<code>False</code>
<code>or</code>	逻辑或:只要有一个操作数为 <code>True</code> 就返回 <code>True</code>	低	<code>True or False</code>	<code>True</code>
<code>not</code>	逻辑非:将 <code>True</code> 变为 <code>False</code> ,反之亦然	高	<code>not True</code>	<code>False</code>

在 Python 中,逻辑操作符按照以下优先级顺序执行:

1. `not`: 具有最高优先级, 首先执行。
2. `and`: 在 `not` 之后执行。
3. `or`: 优先级最低, 最后执行。

例如, 表达式 `not (True or False and True)` 会按照以下顺序计算:

1. 首先执行 `and`, 因为它的优先级高于 `or`, 结果是 `False`。
2. 然后执行 `or`, 其结果为 `True`。
3. 最后, `not` 将结果反转为 `False`。

可以通过使用括号来更改操作顺序, 例如 `(True or False) and True`。

### </> 代码 2.3: 逻辑表达式练习题 1

---

```
1 # 逐步解析每个条件表达式, 讨论not、and和or的结合使用如何影响最终结果
2 x = 8
3 y = 20
4 z = 15
5 result = not (x > 5 and y == 20) or (z < 10)
6
7 # 详细分析每一步的计算过程, 并说明在多种逻辑操作符混合使用时, 如何决定最终结果
8 p = 3
9 q = 6
10 r = 9
11 result = (p + q > r) and (q != r or p <= q)
12
13 # 解释该表达式的执行顺序, 特别是如何处理短路运算 (即在逻辑表达式中尽量减少计算), 推导并验证最终结果
14 x = 0
15 y = 4
16 z = 7
17 result = x > y or (y < z and z > x) or not y
```

---

### </> 代码 2.4: 逻辑表达式练习题 2

---

```
1 # 详细解析各层嵌套表达式的优先级, 推导每个子表达式的计算过程, 并给出最终结果。使用括号展示如何影响逻辑运算
  的执行顺序
2 a = 5
3 b = 10
4 c = 15
5 d = 20
6 result = ((a < b or b > d) and (c > b and not d == 20)) or a == 5
7
8 # 逐步解析该复杂表达式, 讨论其中的逻辑优先级、括号的作用以及 not 操作符的影响。推导出整个表达式的结果, 并
  解释短路计算如何优化了表达式的评估过程
9 p = 3
10 q = 8
11 r = 12
12 s = 0
13 result = not (p > r and q <= r) or (s == 0 and r > q) and not (p == 3 or s > r)
```

---

### 2.3.4 身份运算符

Python 提供两个身份运算符：`is` 和 `is not`。它们用于判断两个对象是否是内存中的同一对象，而不仅仅是比较它们的值。

- `is` 运算符：当两个变量引用同一个对象时，`is` 返回 `True`。例如：

```
1 a = [1, 2, 3]
2 b = a
3 print(a is b) # 输出: True
```

这里，`a` 和 `b` 引用的是同一个列表对象，所以 `a is b` 为 `True`。

- `is not` 运算符：用于判断两个变量是否引用不同的对象。如果它们不是同一个对象，`is not` 返回 `True`：

```
1 c = [1, 2, 3]
2 print(a is not c) # 输出: True
```

即使 `a` 和 `c` 的内容相同，但它们在内存中是不同的对象。

### 2.3.5 成员关系运算符

成员关系运算符用于检查元素是否存在于某个序列中，如列表、元组或字符串。Python 提供了 `in` 和 `not in` 运算符。

- `in` 运算符：用于检查某个值是否是序列的成员。如果该值存在，返回 `True`：

```
1 fruits = ['apple', 'banana', 'cherry']
2 print('banana' in fruits) # 输出: True
```

- `not in` 运算符：用于检查某个值是否不存在于序列中。如果该值不存在，返回 `True`：

```
1 print('mango' not in fruits) # 输出: True
```

### 2.3.6 操作符优先级

在 Python 中，操作符的优先级决定了在表达式中各个操作符的计算顺序。高优先级的操作符会先计算，除非使用括号明确改变运算顺序。以下是 Python 中所有操作符的优先级表，从最高到最低优先级：

当多个操作符的优先级相同时，通常采用**从左到右**的结合顺序，但对于如 `**` 这样的指数运算符，则是**从右到左**结合。理解操作符优先级和结合性对于正确编写复杂的表达式尤为重要，例如数学运算、逻辑判断等。

优先级	操作符	描述
1	<code>()</code>	圆括号用于改变优先级
2	<code>**</code>	指数运算（从右到左）
3	<code>+x</code> , <code>-x</code> , <code>~x</code>	一元加、减,按位取反
4	<code>*</code> , <code>/</code> , <code>//</code> , <code>%</code>	乘法、除法、取整除、取余
5	<code>+</code> , <code>-</code>	加法、减法
6	<code>&lt;&lt;</code> , <code>&gt;&gt;</code>	位移运算符
7	<code>&amp;</code>	按位与
8	<code>^</code>	按位异或
9	<code> </code>	按位或
10	<code>==</code> , <code>!=</code> , <code>&gt;</code> , <code>&lt;</code> , <code>&gt;=</code> , <code>&lt;=</code> , <code>is</code> , <code>is not</code> , <code>in</code> , <code>not in</code>	比较运算符、身份运算符、成员运算符
11	<code>not</code>	逻辑非
12	<code>and</code>	逻辑与
13	<code>or</code>	逻辑或

### 2.3.7 语句

**语句**（Statement）和**表达式**（Expression）是两个重要的概念,它们在 Python 编程结构中扮演着不同的角色。

#### 语句（Statement）

语句是一条完整的指令,用于告诉 Python 解释器执行某个动作。它们通常执行操作或更改程序的状态。常见的语句包括赋值语句、条件语句（如 `if`）、循环语句（如 `for` 和 `while`）、函数定义等。例如：

```
1 x = 5 # 赋值语句
2 if x > 0:
3     print("Positive") # 条件语句
```

语句不一定返回值,其主要作用是执行操作或触发某些效果。

#### 表达式（Expression）

表达式是由变量、操作符、函数调用等组合而成的代码片段,其目的是**计算并返回一个值**。表达式可以非常简单,也可以非常复杂。例如：

```
1 y = 3 + 5 # 3 + 5 是一个表达式, 返回 8
2 z = max(1, 2, 3) # max(1, 2, 3) 是一个表达式, 返回 3
```

在赋值语句中,右侧的内容通常是一个表达式,其结果被赋给变量。表达式的特点是它总是返回一个值。

#### 语句与表达式的区别

- **执行结果:** 语句执行操作,但不一定返回值;表达式总是返回一个值。



- **用途:** 语句是完整的操作指令,而表达式是程序中用来计算值的元素。语句中可以包含表达式,例如在条件判断或循环中。

## 2.4 变量

**重要性:**★★★★★; **难易度:**★

**变量**是用于存储数据的容器。每个变量都指向存储在计算机内存中的特定值。变量可以存储不同类型的数据,包括数字、字符串、布尔值等。Python 与静态类型语言不同,不需要提前声明变量的类型,只需直接通过赋值操作创建一个变量即可。例如:

```
1 x = 10          # 整数
2 name = "Alice" # 字符串
3 is_active = True # 布尔值
```

在上述示例中, `x` 存储一个整数, `name` 存储一个字符串,而 `is_active` 则存储布尔值。

### 2.4.1 变量赋值

通过赋值操作符 `=`,可以将某个值赋给变量。比如:

```
1 age = 25
```

此操作将整数 `25` 赋给变量 `age`。Python 根据所赋值的数据类型,动态决定该变量的类型。

Python 允许在一行中同时为多个变量赋值:

```
1 a, b, c = 1, 2, 3
```

此操作分别将 `1` 赋值给 `a`, `2` 赋值给 `b`, `3` 赋值给 `c`。

### 2.4.2 变量命名

变量的名称可以是字母、数字和下划线的组合,但不能以数字开头。此外,变量命名遵循一系列最佳实践和约定,以确保代码的可读性、可维护性以及团队协作时的易理解性。以下是 Python 变量命名的最佳实践:

- **使用描述性和有意义的名称:** 变量名称应清晰地描述变量的用途或内容。避免使用简短、不具描述性的名称(如 `x`、`y`),而应使用如 `student_age` 或 `total_price` 这样的名称,便于他人阅读和理解。
- **遵循命名约定:** Python 推荐使用 **snake\_case** 作为变量命名规范,即用小写字母并以下划线分隔单词,例如 `first_name`、`total_amount`。这种命名方式符合 Python 的 PEP 8 风格指南,并且使代码更易于阅读。
- **避免使用保留关键字:** 不要使用 Python 的保留字(如 `if`、`else`、`for`)作为变量名,以免导致语法错误。保留关键字(Reserved Keywords)是预定义的词语,它们具有特殊的语法含义,不能用作变量

名、函数名或其他标识符。例如,变量名 `class` 是非法的,但可以使用 `class_name`。Python 3.x 中的保留关键字见表2.4。

- 一致性与简洁性:在代码中保持一致的命名风格至关重要。无论是 `snake_case` 还是 `CamelCase`,都应在整个项目中保持一致。此外,变量名应简洁明了,避免过长或过于冗余的名称。
- 除非是常见的缩写,否则应避免在变量名称中使用缩写。例如,使用 `customer_address` 而不是 `cust_addr`。缩写会增加阅读和理解代码的难度。

通过遵循这些最佳实践,可以显著提高代码的可读性和可维护性,特别是在团队开发或代码重构时。



### 思考题: 纠正变量名错误

**问题:** 根据 Python 的变量命名规则,找出每个变量名中的问题并进行纠正。提示:变量名不能以数字开头、不能包含特殊字符(如 # 和空格),并且避免使用 Python 的保留关键字。

```
1 2ndPlaceWinner = "John"
2 total#ofBooks = 150
3 User Email = "example@domain.com"
4 float = 7.5
5 VARiable = 100
```

#### 答案示例:

```
1 second_place_winner = "John"
2 total_of_books = 150
3 user_email = "example@domain.com"
4 float_number = 7.5
5 variable = 100
```



### 思考题: 创建描述性变量名

根据以下描述,创建符合 Python 命名规则的变量名:

1. 表示用户年龄的变量。
2. 存储博客文章标题的变量。
3. 记录购物车中总金额的变量。
4. 统计已下载文件数量的变量。
5. 表示服务器连接状态的变量。

**问题:** 为每个描述创建合适的变量名,要求变量名简洁明了,符合 Python 的 `snake_case` 命名规范。

#### 答案示例:

```
1 user_age = 30
2 blog_post_title = "Python Basics"
3 total_cart_amount = 250.75
4 downloaded_files_count = 120
5 server_connection_status = True
```

## 2.5 常见 Python 数据类型

重要性:★★★★★; 难易度:★

Python 中的数据类型用于定义和存储不同类型的值。以下是 Python 常见的数据类型分类:

### 2.5.1 数字 (Numeric)

#### 整数

在 Python 中,不同进制的整数可以通过特定的前缀来表示,并且可以轻松在不同进制之间进行转换。Python 支持二进制、八进制、十进制和十六进制等常见的数制系统。以下是表示和转换不同进制整数的方法:

1. **二进制 (Binary)** 通过在数字前加前缀 `0b` 或 `0B` 表示二进制。例如, `0b1010` 表示二进制数 1010, 其十进制值为 10。
2. **八进制 (Octal)** 使用前缀 `0o` 或 `0O` 来表示八进制数。例如, `0o123` 表示八进制数 123, 对应的十进制值为 83。
3. **十进制 (Decimal)** 默认情况下,整数以十进制形式表示,无需添加任何前缀。例如, `123` 即表示十进制数 123。
4. **十六进制 (Hexadecimal)** 前缀 `0x` 或 `0X` 用于表示十六进制数。例如, `0x1A` 表示十六进制数 1A, 对应的十进制值为 26。

#### 进制之间的转换

Python 还提供了内置函数来进行进制转换:

- `bin()`: 将整数转换为二进制字符串表示,例如 `bin(10)` 结果为 `'0b1010'`。
- `oct()`: 将整数转换为八进制字符串表示,例如 `oct(83)` 结果为 `'0o123'`。
- `hex()`: 将整数转换为十六进制字符串表示,例如 `hex(26)` 结果为 `'0x1a'`。

此外, `int()` 函数可以将字符串形式的数值转换为指定进制的整数,例如:

---

```
1 int("0b1010", 2) # 结果为10
2 int("0o123", 8)  # 结果为83
3 int("0x1A", 16)  # 结果为26
```

---

Python 支持的进制范围从 2 到 36。对于大于 10 的基数,使用字母 `A-Z` 来表示 10 到 35 的数值。

#### 浮点数

在 Python 中,浮点数 (`float`) 用于表示带有小数部分的实数,通常是通过 IEEE 754 双精度标准实现的。这意味着浮点数在内存中占用 8 字节 (64 位),其中包含符号位、指数位和尾数位。由于浮点数使用二进制表示,某些十进制小数 (如 `0.1`) 无法精确表示,这可能导致计算结果出现微小的误差。

#### Python 中浮点数的表示与操作

1. **创建浮点数:**浮点数可以直接通过带小数点的数字创建,如 `3.14`,或者使用 `float()` 函数将整数或字符串转换为浮点数:

```
1 a = 3.14      # 直接赋值
2 b = float(5)  # 将整数转换为浮点数
```

2. **浮点数的精度问题:**由于浮点数在计算机中是用二进制表示的,某些十进制小数在二进制中不能精确表示。例如,`0.1` 在 Python 内部的表示并非精确的 `0.1`,而是一个近似值,这可能在数值比较时导致问题。为了解决此类问题,可以使用 `math.isclose()` 函数进行浮点数的比较。

3. **浮点数的算术运算:**浮点数支持标准的加、减、乘、除等运算。例如:

```
1 result = 7.25 + 3.5 # 结果为10.75
```

4. **特殊值:**Python 还支持表示特殊的浮点数值,如正负无穷大 (`inf`) 和非数 (`NaN`)。这些值可以使用 `float('inf')` 或 `float('nan')` 生成,并在科学计算中广泛应用。

在 Python 中,科学计数法 (Scientific Notation) 用于简洁地表示非常大的或非常小的浮点数。科学计数法的形式为  $N \times 10^M$ ,在 Python 中表示为 `NeM`,例如 `1.23e4` 表示的数值是  $1.23 \times 10^4$  (即 12300)。当数值大于 `1e16` 或小于 `1e-4` 时,Python 会自动将浮点数以科学计数法显示。科学计数法在科学计算和金融分析等领域非常有用,特别是在处理极大或极小的数值时。在 Python 中通过直接使用 `e` 来表示科学计数法。例如:

```
1 value = 3.45e-4
2 print(value) # 输出: 0.000345
```

## 浮点数的应用

浮点数在财务计算和科学计算中非常常用,但由于精度限制,某些场景下可能会导致误差。因此,Python 提供了 `decimal` 模块,用于需要高精度的场合,比如会计计算。

## 复数

复数表示为 `a + bj` 的形式,其中 `a` 是实部,`b` 是虚部,`j` 表示虚数单位 (即  $\sqrt{-1}$ )。这与数学中的复数表达式相对应,只是 Python 中用 `j` 代替了传统的 `i` 来表示虚部。可以通过直接赋值和 `complex` 函数两种方式在 Python 中定义复数,使用 `.real` 和 `.imag` 属性,分别访问复数的实部和虚部。Python 也支持对复数进行基本的算术运算,如加法、减法、乘法和除法。

```
1 # 直接赋值
2 z = 3 + 4j
3
4 # 使用 complex() 函数,该函数接受两个参数,分别表示实部和虚部
5 z = complex(3, 4)
6
7 z = 3 + 4j
8 print(z.real) # 输出: 3.0
9 print(z.imag) # 输出: 4.0
10
```

```
11 z1 = 2 + 3j
12 z2 = 1 + 2j
13 result = z1 + z2 # (3+5j)
```

---

此外,Python 还支持计算复数的模、共轭和相位角。例如,使用 `abs()` 函数可以计算复数的模长。

## 2.5.2 布尔值 (Boolean)

布尔值 (Boolean) 是一种表示逻辑真值的数据类型,只有两个可能的取值: `True` 和 `False`。布尔值通常用于控制程序的逻辑流程,例如条件判断和循环控制。

### 布尔值的使用

布尔值可以通过比较运算符 (如 `<`, `>`, `==` 等) 来产生。例如,表达式 `5 > 3` 会返回 `True`,而 `2 == 3` 则返回 `False`。除了直接的布尔值,Python 中的任何数据类型都可以被转换为布尔值,非零数字、非空对象会被视为 `True`,而 `0`、`None` 和空对象 (如空字符串、空列表) 会被视为 `False`。

```
1 bool(1)      # 返回 True
2 bool(0)      # 返回 False
3 bool("")     # 返回 False
4 bool("abc")  # 返回 True
```

---

### 布尔值的应用

布尔值在条件控制中非常重要,比如在 `if` 语句中根据条件的真假来决定程序的执行路径。此外,布尔值也可以用作计数器或状态指示器,帮助管理复杂的逻辑流程。

## 2.5.3 空值 None

在 Python 中, `None` 是一个表示“空”或“无值”的特殊对象。它属于 `NoneType` 类,且是 Python 中的单例对象,这意味着在整个程序中只存在一个 `None` 对象。与其他语言中的 `null` 或 `nil` 类似, `None` 通常用于表示变量没有赋值或函数没有返回任何有意义的值。

### `None` 的常见用途:

- **作为变量的初始值:** 当你不希望立即为变量赋值时,可以使用 `None` 来占位。例如:

```
1 result = None
```

---

之后,你可以检查这个变量是否被赋予了实际值:

```
1 if result is None:
2     result = "some_value"
```

---

- **作为函数的返回值:** 如果一个函数没有明确的返回值,Python 会默认返回 `None`。例如:

```
1 def greet():
2     print("Hello!")
3
4 result = greet() # result 的值为 None
```

---

- **避免可变参数的陷阱**: 在定义函数时, 使用 `None` 作为默认参数的占位符, 可以避免使用像列表这样的可变对象。例如:

```
1 def append_to_list(item, lst=None):
2     if lst is None:
3         lst = []
4     lst.append(item)
5     return lst
```

### 比较 `None`

在 Python 中, 使用 `is` 或 `is not` 来与 `None` 进行比较, 而不是使用 `==`。这是因为 `None` 是单例对象, 而 `is` 操作符用于比较对象的身份, 而非内容。

## 2.5.4 序列 (Sequence)

`str`: 字符串, 用于存储文本数据, 例如 `name = "Alice"`。

`list`: 有序的、可变的元素集合, 例如 `fruits = ["apple", "banana", "cherry"]`。

`tuple`: 有序的、不可变的元素集合, 例如 `coordinates = (10, 20)`。

`range`: 用于生成一系列数字的范围对象, 例如 `range(0, 10)`。

## 2.5.5 映射 (Mapping)

`dict`: 字典类型, 用于存储键值对, 例如 `person = {"name": "Alice", "age": 25}`。

## 2.5.6 集合 (Set)

`set`: 无序且唯一的元素集合, 例如 `unique_numbers = {1, 2, 3}`。

`frozenset`: 不可变的集合, 元素不可更改。

## 2.5.7 类型转换

类型转换是开发者通过调用内置函数手动将一种数据类型转换为另一种常用的数据类型, 例如 `int()`、`float()`、`str()` 等。类型转换通常用于需要确保数据的类型正确时, 特别是在处理用户输入或进行不同类型的数据运算时。例如, 将字符串转换为整数:

```
1 num_str = "123"
2 num_int = int(num_str) # 将字符串'123'转换为整数123
```

通过类型转换, 开发者可以明确指定期望的数据类型, 以防止由于类型不匹配导致的错误。常见的类型转换函数包括:

- `int()`: 将其他类型转换为整数;
- `float()`: 将其他类型转换为浮点数;
- `str()`: 将其他类型转换为字符串。

在进行复杂的类型转换时使用 `try/except` 语句来处理可能发生的转换错误。

## 2.6 函数

重要性:★★★★★; 难易度:★

**函数**是一个用于组织和重用代码的核心工具。函数将相关的代码逻辑封装在一个命名的代码块中,使程序更加模块化和易于维护。函数定义由两个主要部分组成:函数定义和函数体。

### 2.6.1 函数定义与调用

#### 定义函数

函数通过 `def` 关键字定义,后面跟随函数名称和圆括号。圆括号中可以包含**参数**,这些参数是函数在调用时需要传入的值。以下是一个简单的函数示例:

```
1 def greet():  
2     print("Hello, World!")
```

此函数名为 `greet()`,每次调用该函数时,都会打印“Hello, World!”。

#### 调用函数

定义函数后,必须显式调用它才能执行内部代码。例如:

```
1 greet() # 调用函数
```

这会输出 `Hello, World!`。函数调用时,程序控制权会传递给函数的代码,执行完函数内部代码后,程序继续执行调用后面的语句。

#### 参数与返回值

函数可以接受**参数**,用于在函数内处理不同的输入。例如:

```
1 def greet(name):  
2     print(f"Hello, {name}")
```

调用该函数时,需要传入参数:

```
1 greet("John") # 输出: Hello, John
```

此外,函数可以使用 `return` 语句返回一个值:

```
1 def add(a, b):  
2     return a + b  
3  
4 result = add(5, 3) # result 的值为 8
```

这里,函数 `add()` 返回两个数的和。

### 2.6.2 常用 Python 内置函数

Python 中的内置函数提供了许多常用的功能,帮助简化编程任务,表2.5列出了一些常用的 Python 内置函数的简要介绍和它们的功能描述:



## 2.7 输入和输出

重要性:★★★★★; 难度度:★

输入与输出是程序与用户交互的基础功能。Python 提供了内置的 `input()` 和 `print()` 函数, 分别用于接收用户输入和向屏幕输出结果。

### `input()` 函数

`input()` 函数用于从用户处接收输入。调用该函数时, 程序会暂停运行并等待用户输入, 直到用户按下回车键。输入的内容会作为字符串返回。如果需要将用户输入转换为其他类型 (如整数或浮点数), 可以使用类型转换函数, 例如:

---

```
1 age = int(input("Enter your age: ")) # 将输入转换为整数
```

---

这个函数在编写交互式程序时非常有用, 例如接受用户的配置或数据。

### `print()` 函数

`print()` 函数用于向控制台输出结果。它可以接收多个参数, 并将它们以空格分隔输出。`print()` 还可以使用可选的 `sep` 和 `end` 参数, 来控制输出的分隔符和结束符。例如:

---

```
1 print("Hello", "World", sep=" ", end="!") # 输出: Hello, World!
```

---

通过这些参数, 你可以灵活地定制输出格式。

### 输出格式化

Python 还提供了多种方式格式化输出, 例如使用 `str.format()` 或 f-string (格式化字符串)。这些方法允许在输出时插入变量或表达式, 从而创建更易读的输出格式:

---

```
1 name = "Alice"
2 age = 30
3 print(f"{name} is {age} years old.") # 输出: Alice is 30 years old.
```

---

这种方式非常适合需要清晰、美观输出的应用。

## 2.8 模块

重要性:★★★★★; 难度度:★

模块是 Python 中一种将相关功能组织在一起的方式。通过导入模块, 可以使用模块中定义的函数、常量和其他对象。Python 自带的 `math` 模块就是一个常用的数学模块, 它提供了基本的数学运算函数和常量。

### 模块的导入

要使用 `math` 模块, 首先需要通过 `import` 语句将其导入。导入后, 你可以通过 `math.` 前缀访问模块中的函数和常量。例如:

---

```
1 import math
2
3 result = math.sqrt(16) # 使用math模块计算平方根
4 print(result) # 输出: 4.0
```

---



这里使用了 `math.sqrt()` 函数来计算 16 的平方根。`sqrt()` 是 `math` 模块中的一个函数,用于返回非负数的平方根。

### 常用的 `math` 模块函数

`math` 模块提供了许多常用的数学函数和常量,如表2.6所示:

### 模块的好处

通过将相关函数和常量封装在模块中,Python 的 `math` 模块使得数学计算更加方便和可靠。例如,使用 `math.pi` 比手动输入圆周率更精确且方便。此外,模块化的设计也增强了代码的可读性和可维护性。

## 2.9 流程控制

重要性:★★★★★; 难易度:★★

**流程控制 (Control Flow)** 是指程序代码的执行顺序。通常情况下,Python 程序会从上到下、逐行执行代码。但通过使用流程控制语句,程序可以根据特定条件、循环结构等来控制代码的执行顺序。Python 中的主要流程控制结构包括:顺序执行、选择 (条件语句) 和重复 (循环语句)。

### 2.9.1 条件语句

**条件语句**是 Python 编程中控制流程的基础,它允许程序根据特定条件执行不同的代码块。常见的条件语句包括 `if`、`elif` 和 `else`。

#### 基本条件语句的结构

`if` 语句用于检查条件是否为真,如果条件成立,则执行相应的代码块:

---

```
1 if condition:
2     # 条件为真的情况下执行的代码
```

---

当需要处理多个条件时,可以使用 `elif` 来扩展条件判断。`else` 则用于在所有条件都不成立时执行默认代码块:

---

```
1 if condition1:
2     # 条件1为真时执行
3 elif condition2:
4     # 条件2为真时执行
5 else:
6     # 所有条件为假时执行
```

---

### 案例:基于销售金额的产品分类

假设我们在一个电子商务平台上,根据销售金额对产品进行分类。我们可以使用条件语句判断产品属于“低销量”、“中等销量”还是“高销量”:

```
sales = 15000 # 假设这是产品的销售金额
```

```
if sales > 20000:
    print("高销量产品")
elif 10000 <= sales <= 20000:
    print("中等销量产品")
else:
    print("低销量产品")
```

在这个例子中,程序根据产品的销售额来动态分类,为企业的决策提供自动化支持。在数字经济和电子商务中,企业可以根据多种条件(如用户行为、销售额、市场趋势)进行决策。例如,当分析用户消费数据时,可以使用嵌套 `if` 语句或者 `if-elif-else` 链来判断不同消费习惯的用户群体。通过条件语句,Python 程序能够灵活应对复杂的商业逻辑,使得企业在数字经济环境下更有效地处理数据和做出决策。

## 2.9.2 循环语句

**for** 循环和 **while** 循环是 Python 中常用的控制流程结构,广泛用于商业数据分析中的多种任务,如遍历数据集或动态处理数据。

### for 循环

`for` 循环用于遍历一个序列(如列表、元组或字符串),每次迭代提取一个元素。常见的使用场景包括逐行读取数据、遍历数据集中每一条记录等。

例如,假设我们有一个代表公司季度销售额的列表,我们可以用 `for` 循环来计算总销售额:

```
1 sales = [12000, 18000, 25000, 30000]
2 total_sales = 0
3
4 for sale in sales:
5     total_sales += sale
6
7 print(f"总销售额为: {total_sales}")
```

这个例子展示了如何遍历每个季度的销售额并计算出全年总销售额。

### while 循环

`while` 循环根据条件执行代码块,直到条件不再满足为止。它通常用于需要不确定的迭代次数时,例如数据监控或数据收集过程。

例如,假设我们想持续监控某一产品的库存量,当库存低于某个阈值时自动停止销售:

```
1 stock = 50 # 初始库存
2 threshold = 10 # 库存阈值
3
```

```
4 while stock > threshold:
5     print(f"当前库存: {stock}, 继续销售...")
6     stock -= 5 # 每次销售5个
7 print("库存低于阈值, 停止销售")
```

---

这个例子展示了一个常见的业务场景, `while` 循环会不断检查库存量,并在库存低于阈值时停止销售。

在商业数据分析中, `for` 循环常用于遍历数据集或生成报表,而 `while` 循环则适用于实时数据监控和需要动态处理的数据场景。这两种循环结构为业务决策的自动化提供了灵活的支持。

## 2.10 保存和执行 Python 程序

重要性:★★★★; 难易度:★★

编写 Python 程序可以使用任何文本编辑器或专门的集成开发环境 (IDE)。要编写一个 Python 程序,首先打开一个文本编辑器并输入代码。例如,以下是一段简单的代码:

```
1 # 输出 "Hello, World!"
2 print("Hello, World!")
```

---

在这段代码中, `#` 开头的行是注释,注释不会被 Python 解释器执行,用于帮助开发者理解代码。编写完程序后,保存文件时需使用 `.py` 作为扩展名,例如 `hello.py`。

### 命令行运行 Python 脚本

要通过命令行运行 Python 脚本,可以使用以下步骤:

- 打开命令行 (Windows 上的 `cmd` 或 Mac/Linux 上的 `Terminal`)。
- 使用 `cd` 命令导航到脚本所在的目录。
- 输入以下命令运行脚本:

```
python hello.py
```

如果使用的是 Python 3,可以使用 `python3` 来执行。

### 使用 IDE 运行 Python 脚本

大多数 IDE,如 PyCharm 或 Visual Studio Code,都提供了集成的“运行”按钮。编写完代码后,可以直接在 IDE 中点击运行按钮,代码将在 IDE 的控制台中执行,输出结果会显示在同一窗口中。

在 Python 编程中,**Jupyter Notebook** 是一种交互式的开发环境,广泛应用于数据分析、可视化和机器学习等领域。Jupyter Notebook 将代码和文档整合在一个可共享的界面中,既能执行代码,又能撰写说明,极大地方便了数据科学工作流。

### 使用 Jupyter Notebook 编写和运行代码

Jupyter Notebook 中的每个代码块称为“单元 (Cell)”,可以在其中编写和运行 Python 代码。要执行代码,按下 `Shift + Enter`,或者点击工具栏上的“运行”按钮。每个代码单元的输出会直接显示在其下方,方便快速查看结果。例如:

```
1 print("Hello, World!")
```

---

当运行这段代码时,输出会立刻显示在单元下方。

Jupyter 还支持 Markdown 格式,允许用户在笔记本中添加格式化文本,便于解释代码或分析结果。例如,可以使用 Markdown 来创建标题、列表,或插入公式。通过这种方式,代码和说明可以整齐地组合在一起,适合用于数据报告。