Python 语法全面详解与实战指南
第一章 Python 基础语法核心要素
1.1 Python 语言概述
Python 是一种解释型、面向对象、动态数据类型的高级程序设计语言，由 Guido van Rossum 于 1989 年圣诞节期间构思，并在 1991 年正式发布第一个版本。它以“优雅、清晰、简洁”为设计哲学，摒弃了 C/C++ 等语言中繁琐的语法格式（如大括号、分号强制要求），通过缩进来划分代码块，让开发者能够更专注于业务逻辑的实现，而非语法细节的纠结。如今，Python 已经在数据分析、人工智能、Web 开发、自动化运维、爬虫开发、游戏开发等多个领域占据了重要地位，成为了全球最受欢迎的编程语言之一，其强大的生态库和活跃的社区更是为开发者提供了源源不断的支持。
Python 有两个主要的版本分支，分别是 Python 2.x 和 Python 3.x，其中 Python 2.x 已经停止官方维护，目前主流的开发和学习都以 Python 3.x 版本为主，推荐使用 3.8 及以上的稳定版本，该版本不仅修复了大量早期版本的 Bug，还新增了许多实用的语法特性和性能优化，比如 f-string 格式化、海象运算符、类型注解增强等。
1.2 标识符与关键字
在 Python 中，标识符是用来命名变量、函数、类、模块等程序实体的字符串，它的命名必须遵循严格的规则，否则会导致语法错误。具体的命名规则如下：
标识符由字母（A-Z、a-z）、数字（0-9）和下划线（_）组成，且不能以数字开头。例如，name123、_age、user_name 都是合法的标识符，而 123name、user-name 则是非法标识符，后者使用了连字符，不符合 Python 语法要求。
Python 标识符区分大小写，这意味着 Name、name、NAME 是三个完全不同的标识符，在使用过程中需要格外注意，避免因大小写混淆导致变量未定义或功能异常。例如，定义了 age = 20 之后，调用 Age 会抛出 NameError 异常。
不能使用 Python 的关键字作为标识符，关键字是 Python 语言内置的、具有特殊含义的保留字，它们已经被语言本身占用，无法被用户重新定义。
标识符的命名尽量做到“见名知义”，遵循一定的命名规范，提高代码的可读性和可维护性。常用的命名规范有驼峰命名法（如 userName、studentInfo，多用于类名）和下划线命名法（如 user_name、student_age，多用于变量、函数名）。
Python 3.x 中共有 35 个关键字，我们可以通过 keyword 模块来查看所有的关键字，具体代码如下：
python
运行
```
# 导入 keyword 模块
import keyword

# 查看所有关键字
all_keywords = keyword.kwlist
print("Python 3.x 所有关键字：")
for idx, word in enumerate(all_keywords, 1):
    print(f"{idx:2d}. {word}")

# 判断某个字符串是否为关键字
is_keyword = keyword.iskeyword("if")
print(f"\n'if' 是否为 Python 关键字：{is_keyword}")
```

运行上述代码后，我们可以看到完整的关键字列表，包括 if、else、for、while、def、class 等。这些关键字在程序中都有特定的用途，例如 if 用于条件判断，def 用于定义函数，class 用于定义类，在编写代码时，绝对不能将这些关键字用作变量名或函数名，否则会引发语法错误。

1.3 变量与数据类型

1.3.1 变量的定义与赋值

变量是程序中用于存储数据的容器，它可以看作是一个指向数据内存地址的标签。在 Python 中，定义变量非常简单，不需要像 C/C++、Java 等语言那样提前声明变量的类型，直接通过赋值语句即可创建变量，语法格式为：变量名 = 变量值。

例如：

python
运行

```
# 定义整数类型变量
age = 25
# 定义字符串类型变量
name = "张三"
# 定义浮点数类型变量
height = 1.78
# 定义布尔类型变量
is_student = False
# 变量重新赋值（可以改变数据类型）
age = "二十五"
```

从上述代码可以看出，Python 是一种动态类型语言，变量的类型会随着赋值内容的变化而自动改变，这与静态类型语言有很大的区别。在赋值过程中，等号（=）是赋值运算符，而非数学中的等于号，它的作用是将右侧的变量值存储到左侧变量名对应的内存空间中。

此外，Python 还支持多重赋值和序列解包赋值，这两种赋值方式在实际开发中非常实用，可以提高代码的简洁性。

python
运行

```
# 多重赋值（多个变量赋予相同的值）
a = b = c = 10
print(f"a={a}, b={b}, c={c}")

# 序列解包赋值（多个变量分别赋予不同的值）
name, age, gender = "李四", 22, "男"
print(f"姓名：{name}，年龄：{age}，性别：{gender}")

# 列表解包赋值
hobbies = ["读书", "跑步", "听音乐"]
hobby1, hobby2, hobby3 = hobbies
print(f"爱好 1：{hobby1}，爱好 2：{hobby2}，爱好 3：{hobby3}")
```

1.3.2 基本数据类型

Python 中的数据类型可以分为基本数据类型和复合数据类型，其中基本数据类型主要包括整数（int）、浮点数（float）、字符串（str）、布尔值（bool）和空值（NoneType）。

整数（int）：用于表示整数，包括正整数、负整数和 0，Python 支持任意大小的整数，不存在溢出问题，这一点与很多编程语言不同。例如，10、-5、0、1000000000000000000 都是合法的整数。整数还支持多种进制表示，包括十进制（默认）、二进制（以 0b 开头）、八进制（以 0o 开头）和十六进制（以 0x 开头）。

python
运行

```
# 不同进制的整数
decimal_num = 10  # 十进制
binary_num = 0b1010  # 二进制
octal_num = 0o12  # 八进制
hex_num = 0xa  # 十六进制

# 转换为十进制输出
print(f"二进制 0b1010 转换为十进制：{binary_num}")
print(f"八进制 0o12 转换为十进制：{octal_num}")
print(f"十六进制 0xa 转换为十进制：{hex_num}")
```

浮点数（float）：用于表示带有小数部分的数值，包括正浮点数和负浮点数。浮点数的表示有两种方式，一种是直接书写小数，例如 3.14、-0.5、0.0；另一种是科学计数法，例如 1.23e5（表示 1.23×10^5）、4.56e-3（表示 4.56×10^-3）。需要注意的是，浮点数在计算机中存储时可能会存在精度误差，这是由于二进制存储机制导致的，在进行高精度计算时，建议使用 decimal 模块。

python
运行

```
# 浮点数的定义与运算
num1 = 3.14
num2 = 2.0
result = num1 * num2
print(f"3.14 × 2.0 = {result}")

# 浮点数精度误差示例
num3 = 0.1
num4 = 0.2
print(f"0.1 + 0.2 = {num3 + num4}")  # 输出结果并非 0.3，而是 0.30000000000000004
```

字符串（str）：用于表示文本数据，由一系列字符组成，字符串可以使用单引号（'）、双引号（"）或三引号（''' 或 """）包裹。其中，单引号和双引号用于表示单行字符串，三引号用于表示多行字符串，还可以在字符串中添加注释。字符串是不可变对象，一旦创建，就无法修改其中的单个字符，只能通过切片、拼接等方式生成新的字符串。

python
运行

```
# 字符串的定义
str1 = 'Hello Python'
str2 = "你好，Python"
str3 = '''这是一个多行字符串
第一行内容
```

第二行内容'''
str4 = """这也是一个多行字符串
支持换行
还可以包含单引号(')和双引号(")"""

# 字符串拼接
str5 = str1 + " " + str2
print(f"字符串拼接结果：{str5}")

# 字符串重复
str6 = "Python " * 3
print(f"字符串重复结果：{str6}")

# 字符串切片（获取子字符串）
str7 = "Hello World"
print(f"字符串切片 [0:5]：{str7[0:5]}") # 输出 Hello
布尔值（bool）：只有两个取值，True（真）和 False（假），主要用于条件判断和逻辑运算。布尔值可以通过直接赋值获得，也可以通过比较运算、逻辑运算得到。在 Python 中，很多数据类型都可以转换为布尔值，其中，0、0.0、""（空字符串）、[]（空列表）、{}（空字典）、None 等都会被转换为 False，其他非空、非零的数据都会被转换为 True。
python
运行
# 布尔值的定义与运算
bool1 = True
bool2 = False

# 比较运算得到布尔值
bool3 = 10 > 5
bool4 = 20 == 30
print(f"10 > 5：{bool3}")
print(f"20 == 30：{bool4}")

# 逻辑运算（and、or、not）
bool5 = bool1 and bool3
bool6 = bool2 or bool4
bool7 = not bool1
print(f"True and True：{bool5}")
print(f"False or False：{bool6}")
print(f"not True：{bool7}")
空值（NoneType）：只有一个取值 None，表示“无”或“空”，它与 0、"" 不同，None 是一个独立的数据类型，专门用于表示变量没有被赋予有效数据。None 经常用于函数的返回值，表示函数没有返回有用的结果，也可以用于初始化变量，预留内存空间。
python
运行

```
# 空值的定义与使用
var = None
print(f"变量 var 的值：{var}")
print(f"变量 var 的类型：{type(var)}")

# 函数返回 None
def empty_func():
    pass  # pass 表示空语句，无实际功能

result = empty_func()
print(f"empty_func 函数返回值：{result}")
```

1.3.3 复合数据类型

复合数据类型是由多个基本数据类型或复合数据类型组成的数据结构，主要包括列表（list）、元组（tuple）、字典（dict）和集合（set），它们在实际开发中使用频率极高，各自具有不同的特性和适用场景。

列表（list）：是一种有序、可变的序列数据类型，使用中括号 [] 包裹元素，元素之间用逗号分隔。列表中的元素可以是不同的数据类型，包括整数、浮点数、字符串、列表、字典等，支持添加、删除、修改、查找等操作。

python
运行

```
# 列表的定义
list1 = [1, 2, 3, 4, 5]
list2 = ["苹果", "香蕉", "橙子"]
list3 = [1, "Python", 3.14, [True, None]]

# 列表添加元素
list2.append("葡萄")
print(f"添加元素后的列表：{list2}")

# 列表修改元素
list1[0] = 10
print(f"修改元素后的列表：{list1}")

# 列表删除元素
del list2[1]
print(f"删除元素后的列表：{list2}")

# 列表查找元素索引
index = list2.index("橙子")
print(f"橙子在列表中的索引：{index}")
```

元组（tuple）：是一种有序、不可变的序列数据类型，使用小括号 () 包裹元素，元素之间用逗号分隔。元组的特性与列表类似，支持索引、切片、查找等操作，但不支持添加、删除、修改元素。由于元组的不可变性，它比列表更安全，在需要存储固定数据、作为字典的键、函数的返回值（多个值）时，通常会使用元组。

python
运行
```
# 元组的定义
tuple1 = (1, 2, 3, 4, 5)
tuple2 = ("张三", "李四", "王五")
tuple3 = (1, "Python", 3.14, (True, None))  # 嵌套元组
tuple4 = (10,)  # 单个元素的元组，必须添加逗号，否则会被视为普通数据

# 元组索引与切片
print(f"元组 tuple2 的第 2 个元素：{tuple2[1]}")
print(f"元组 tuple1 的切片 [1:4]：{tuple1[1:4]}")

# 元组不可变验证（尝试修改元素会报错）
try:
    tuple1[0] = 100
except TypeError as e:
    print(f"修改元组元素报错：{e}")
```
字典（dict）：是一种无序（Python 3.7 及以上版本为有序）、可变的键值对（key-value）数据类型，使用大括号 {} 包裹，每个键值对之间用逗号分隔，键（key）和值（value）之间用冒号 : 分隔。字典的键必须是不可变数据类型（如整数、字符串、元组），且不能重复，值可以是任意数据类型，支持添加、删除、修改、查找键值对等操作。
python
运行
```
# 字典的定义
dict1 = {"name": "张三", "age": 25, "gender": "男"}
dict2 = {1: "苹果", 2: "香蕉", 3: "橙子"}
dict3 = {"name": "李四", "info": {"addr": "北京", "tel": "12345678901"}}

# 字典查找值（通过键）
print(f"张三的年龄：{dict1['age']}")
print(f"键 2 对应的值：{dict2[2]}")
print(f"李四的地址：{dict3['info']['addr']}")

# 字典添加键值对
dict1["height"] = 1.78
print(f"添加键值对后的字典：{dict1}")

# 字典修改值
dict1["age"] = 26
print(f"修改值后的字典：{dict1}")

# 字典删除键值对
del dict1["gender"]
print(f"删除键值对后的字典：{dict1}")
```

集合（set）：是一种无序、可变、不包含重复元素的序列数据类型，使用大括号 {} 包裹，元素之间用逗号分隔，也可以通过 set() 函数创建集合。集合主要用于去重、交集、并集、差集等集合运算，不支持索引和切片操作。

python
运行
```
# 集合的定义
set1 = {1, 2, 3, 4, 5}
set2 = {"苹果", "香蕉", "橙子", "苹果"}  # 自动去重
set3 = set([1, 2, 2, 3, 3, 3])  # 通过 list 创建集合，自动去重

print(f"去重后的 set2：{set2}")
print(f"去重后的 set3：{set3}")

# 集合添加元素
set1.add(6)
print(f"添加元素后的 set1：{set1}")

# 集合删除元素
set1.remove(3)
print(f"删除元素后的 set1：{set1}")

# 集合运算（交集、并集、差集）
set4 = {1, 2, 3, 4}
set5 = {3, 4, 5, 6}
intersection = set4 & set5  # 交集
union = set4 | set5  # 并集
difference = set4 - set5  # 差集

print(f"set4 和 set5 的交集：{intersection}")
print(f"set4 和 set5 的并集：{union}")
print(f"set4 和 set5 的差集：{difference}")
```

1.4 缩进与代码块

与其他编程语言不同，Python 不使用大括号 {} 来划分代码块，而是通过缩进来确定代码的层级关系，这是 Python 语法的重要特征之一。缩进的空格数没有严格的规定（通常建议使用 4 个空格，这是 PEP 8 编码规范的推荐要求），但在同一个代码块中，所有代码的缩进量必须保持一致，否则会引发 IndentationError 异常。

Python 中的代码块通常与控制流语句（如 if、for、while）、函数定义（def）、类定义（class）等结合使用，这些语句的末尾通常会带有一个冒号 :，表示接下来的缩进代码是该语句的代码块。

1.4.1 缩进的基本使用

python
运行
```
# 条件判断语句中的缩进
age = 18
```

```
if age >= 18:
    print("你已经成年了。")
    print("可以独立承担民事责任了。")
else:
    print("你还未成年。")
    print("需要由监护人监护。")

# 循环语句中的缩进
for i in range(5):
    print(f"当前循环次数：{i+1}")
    print(f"当前数值：{i}")

# 函数定义中的缩进
def say_hello(name):
    print(f"你好，{name}！")
    print("欢迎学习 Python 语法。")

say_hello("张三")
```

在上述代码中，if、else、for、def 语句的末尾都带有冒号 :，后续的代码都进行了缩进，形成了对应的代码块。同一个代码块中的代码具有相同的缩进量，执行时会作为一个整体被执行。

1.4.2 缩进的注意事项

避免混合使用空格和制表符（Tab）进行缩进，这会导致缩进混乱，引发语法错误。在大多数代码编辑器中，可以设置将 Tab 键转换为 4 个空格，避免出现此类问题。

嵌套代码块需要逐层增加缩进量，每一层嵌套的缩进量保持一致，这样可以清晰地体现代码的层级关系，提高代码的可读性。

python

运行

```
# 嵌套代码块的缩进
score = 95
if score >= 60:
    print("考试合格。")
    if score >= 90:
        print("成绩优秀，获得奖学金。")
    elif score >= 80:
        print("成绩良好，继续加油。")
    else:
        print("成绩及格，需要查漏补缺。")
else:
    print("考试不合格，需要补考。")
```

空语句可以使用 pass 关键字填充，pass 不执行任何实际操作，仅用于保持语法的完整性，避免因缺少代码块而引发语法错误。

python

运行

```
# pass 语句的使用
num = 10
if num > 5:
    pass  # 暂时不编写任何逻辑，避免语法错误
else:
    print("num 小于等于 5")
```

1.5 注释

注释是程序中用于解释代码功能、说明开发思路、标注注意事项的文本内容，它不会被 Python 解释器执行，仅用于提高代码的可读性和可维护性。在 Python 中，注释分为单行注释和多行注释两种。

1.5.1 单行注释

单行注释使用井号 # 开头，# 后面的所有内容都被视为注释，直到行尾结束。单行注释可以单独占据一行，也可以跟随在代码的末尾，作为代码的行内注释。

python
运行

```
# 这是一个单行注释，单独占据一行
# 定义一个变量，用于存储用户姓名
name = "张三"  # 这是一个行内注释，用于说明变量的用途

# 定义一个函数，用于计算两个数的和
def add(a, b):
    return a + b  # 返回两个数的计算结果
```

1.5.2 多行注释

多行注释用于注释较长的文本内容，通常使用三引号（''' 或 """）包裹，三引号之间的所有内容都被视为注释，可以跨越多行。多行注释经常用于函数、类、模块的开头，作为文档字符串，说明其功能、参数、返回值等信息。

python
运行

```
'''
这是一个多行注释，使用三个单引号包裹。
它可以跨越多行，用于注释较长的内容。
下面的代码用于实现两个数的乘法运算。
'''

"""
这也是一个多行注释，使用三个双引号包裹。
与单引号包裹的多行注释功能完全一致。
"""

def multiply(a, b):
    """
    函数功能：计算两个数的乘积
    参数：
        a：第一个乘数
```

```
        b：第二个乘数
    返回值：
        两个数的乘积
    """
    return a * b
```

1.5.3 注释的使用规范

注释要简洁明了，避免冗余和废话，重点突出代码的核心逻辑和注意事项，不要对显而易见的代码进行注释。

注释要与代码保持同步更新，当代码发生修改时，对应的注释也需要及时调整，避免出现注释与代码不符的情况。

尽量使用中文注释（针对中文开发团队），确保所有团队成员都能理解注释的含义，避免使用晦涩难懂的英文或专业术语。

对于复杂的算法、业务逻辑，要添加详细的注释，说明其实现思路、步骤和关键点，方便后续的代码维护和迭代开发。

第二章 Python 流程控制语句

2.1 条件判断语句

条件判断语句又称分支语句，它根据给定的条件是否成立，来决定执行不同的代码块。Python 中的条件判断语句主要包括 if 语句、if-else 语句、if-elif-else 语句，以及嵌套的条件判断语句。

2.1.1 if 语句

if 语句是最基本的条件判断语句，它的语法格式如下：

```plaintext
if 条件表达式:
    代码块
```

其中，条件表达式可以是比较运算、逻辑运算、布尔值等，当条件表达式的值为 True 时，执行后续的缩进代码块；当条件表达式的值为 False 时，跳过该代码块，继续执行后续的代码。

```python
运行
# if 语句的基本使用
num = 10
if num > 5:
    print(f"{num} 大于 5")

# 条件表达式为逻辑运算
score = 85
if score >= 60 and score < 90:
    print(f"成绩 {score} 为良好")
```

2.1.2 if-else 语句

if-else 语句用于处理“二选一”的场景，当条件表达式成立时，执行 if 对应的代码块；当条件表达式不成立时，执行 else 对应的代码块。它的语法格式如下：

```plaintext
if 条件表达式:
    代码块 1
```

```
else:
    代码块 2
```
python
运行
```
# if-else 语句的基本使用
age = 17
if age >= 18:
    print("你已经成年，可以进入网吧。")
else:
    print("你还未成年，禁止进入网吧。")

# 条件表达式为变量转换的布尔值
name = ""
if name:
    print(f"你的姓名是：{name}")
else:
    print("你还未输入姓名，请补充姓名信息。")
```
2.1.3 if-elif-else 语句

if-elif-else 语句用于处理“多选一”的场景，它可以包含多个 elif 分支，用于判断多个不同的条件，当某个 elif 条件成立时，执行对应的代码块，后续的 elif 和 else 分支将被跳过。它的语法格式如下：

plaintext
```
if 条件表达式 1:
    代码块 1
elif 条件表达式 2:
    代码块 2
elif 条件表达式 3:
    代码块 3
...
else:
    代码块 n
```
python
运行
```
# if-elif-else 语句的基本使用
score = 92
if score >= 90:
    print(f"成绩 {score}，等级为优秀。")
elif score >= 80:
    print(f"成绩 {score}，等级为良好。")
elif score >= 70:
    print(f"成绩 {score}，等级为中等。")
elif score >= 60:
    print(f"成绩 {score}，等级为及格。")
else:
```

```
    print(f"成绩 {score}，等级为不及格，需要补考。")
```

2.1.4 嵌套条件判断语句

嵌套条件判断语句是指在 if、else、elif 对应的代码块中，再次包含条件判断语句，用于处理复杂的多层条件场景。嵌套的层级没有严格限制，但建议尽量减少嵌套层级（不超过 3 层），避免代码过于复杂，难以阅读和维护。

python

运行

```
# 嵌套条件判断语句的基本使用
num = 15
if num > 10:
    print(f"{num} 大于 10")
    if num < 20:
        print(f"{num} 小于 20")
        print(f"{num} 在 10 和 20 之间")
    else:
        print(f"{num} 大于等于 20")
else:
    print(f"{num} 小于等于 10")
```

2.2 循环语句

循环语句用于重复执行某一段代码块，直到满足终止条件为止。Python 中的循环语句主要包括 for 循环和 while 循环，此外，还提供了 break、continue、else 等关键字，用于控制循环的执行流程。

2.2.1 for 循环

for 循环是一种遍历循环，它可以遍历序列类型（如列表、元组、字符串、字典、集合）中的所有元素，也可以通过 range() 函数生成一个整数序列，用于指定循环次数。它的语法格式如下：

plaintext

```
for 变量 in 序列/可迭代对象:
    代码块
```

遍历序列类型

python

运行

```
# 遍历列表
fruits = ["苹果", "香蕉", "橙子", "葡萄"]
for fruit in fruits:
    print(f"当前水果：{fruit}")

# 遍历字符串
str = "Python"
for char in str:
    print(f"当前字符：{char}")

# 遍历字典
student = {"name": "张三", "age": 25, "gender": "男"}
```

```
# 遍历字典的键
for key in student:
    print(f"字典的键：{key}，对应的值：{student[key]}")
# 遍历字典的键值对
for key, value in student.items():
    print(f"字典的键：{key}，对应的值：{value}")
```

使用 range() 函数控制循环次数

range() 函数用于生成一个整数序列，它的语法格式为 range(start, stop, step)，其中：

start：序列的起始值，默认值为 0；

stop：序列的终止值，不包含该值（左闭右开）；

step：序列的步长，默认值为 1。

python

运行

```
# 基本使用（指定终止值）
for i in range(5):
    print(f"当前循环次数：{i+1}，当前数值：{i}")

# 指定起始值和终止值
for i in range(5, 10):
    print(f"当前数值：{i}")

# 指定起始值、终止值和步长
for i in range(0, 10, 2):
    print(f"当前数值：{i}")

# 倒序循环
for i in range(10, 0, -1):
    print(f"当前数值：{i}")
```

2.2.2 while 循环

while 循环是一种条件循环，它根据给定的条件是否成立，来决定是否继续执行循环体。当条件表达式的值为 True 时，重复执行循环体；当条件表达式的值为 False 时，终止循环。它的语法格式如下：

plaintext

```
while 条件表达式:
    代码块（循环体）
```

需要注意的是，使用 while 循环时，必须确保循环体中包含修改条件表达式的语句，否则会导致循环无法终止，形成死循环。

python

运行

```
# 基本使用（计算 1 到 10 的累加和）
sum_num = 0
i = 1
while i <= 10:
    sum_num += i
```

```
    i += 1  # 修改循环变量，避免死循环
print(f"1 到 10 的累加和：{sum_num}")

# 死循环示例（谨慎运行，需手动终止）
# while True:
#     print("这是一个死循环")
```

2.2.3 循环控制关键字

break 关键字：用于立即终止当前循环，跳出循环体，不再执行循环体中剩余的代码，也不再判断循环条件，直接执行循环后续的代码。

python
运行

```
# for 循环中使用 break
fruits = ["苹果", "香蕉", "橙子", "葡萄"]
for fruit in fruits:
    if fruit == "橙子":
        print("找到橙子，终止循环。")
        break
    print(f"当前水果：{fruit}")

# while 循环中使用 break
i = 1
while i <= 10:
    if i == 6:
        print("i 等于 6，终止循环。")
        break
    print(f"当前数值：{i}")
    i += 1
```

continue 关键字：用于跳过当前循环的剩余代码，直接进入下一次循环的条件判断（for 循环直接遍历下一个元素，while 循环重新判断条件表达式）。

python
运行

```
# for 循环中使用 continue
for i in range(1, 11):
    if i % 2 == 0:
        continue  # 跳过偶数，只打印奇数
    print(f"当前奇数：{i}")

# while 循环中使用 continue
i = 0
while i < 10:
    i += 1
    if i % 2 == 1:
        continue  # 跳过奇数，只打印偶数
    print(f"当前偶数：{i}")
```

else 关键字：循环中的 else 关键字用于在循环正常终止（没有被 break 关键字终止）时，执行对应的代码块。如果循环被 break 关键字终止，则 else 代码块不会被执行。

python
运行
```
# for 循环中使用 else
fruits = ["苹果", "香蕉", "橙子", "葡萄"]
for fruit in fruits:
    print(f"当前水果：{fruit}")
else:
    print("循环正常终止，所有水果都已遍历完毕。")

# for 循环被 break 终止，else 代码块不执行
for fruit in fruits:
    if fruit == "橙子":
        print("找到橙子，终止循环。")
        break
    print(f"当前水果：{fruit}")
else:
    print("循环正常终止，所有水果都已遍历完毕。")
```

2.2.4 嵌套循环

嵌套循环是指在一个循环体中，再次包含另一个循环语句，外层循环控制循环的整体次数，内层循环控制每次外层循环中的细节次数。Python 支持多层嵌套循环，但同样建议尽量减少嵌套层级，避免代码过于复杂。

python
运行
```
# 嵌套 for 循环（打印 9*9 乘法表）
for i in range(1, 10):
    for j in range(1, i+1):
        print(f"{j} × {i} = {i*j}\t", end="")
    print()  # 换行

# 嵌套 while 循环（计算 1 到 5 中，每个数的 1 到 3 倍）
i = 1
while i <= 5:
    j = 1
    while j <= 3:
        print(f"{i} × {j} = {i*j}")
        j += 1
    print("-" * 20)
    i += 1
```

第三章 Python 函数定义与使用

3.1 函数的基本概念

函数是一段封装好的、可重复使用的代码块，它用于实现特定的功能，接受输入参数，经过内部逻辑处理后，返回输出结果。将代码封装为函数，可以提高代码的复用性、可读性和可

维护性，减少重复代码的编写，便于后续的代码调试和迭代开发。
在 Python 中，函数可以分为内置函数和自定义函数：
内置函数：Python 语言自带的函数，无需用户定义，可以直接调用，例如 print()、len()、type()、range() 等，这些函数覆盖了常用的基础功能，方便开发者快速使用。
自定义函数：用户根据自身的业务需求，自行定义的函数，用于实现特定的功能，自定义函数需要先定义，后调用。
3.2 函数的定义与调用
3.2.1 函数的定义
Python 中使用 def 关键字来定义函数，函数定义的语法格式如下：
plaintext
```
def 函数名(参数列表):
    """文档字符串（可选）"""
    函数体（核心逻辑代码）
    return 返回值（可选）
```
各部分说明：
函数名：遵循 Python 标识符的命名规则，尽量做到“见名知义”，使用下划线命名法，例如 add_num、get_student_info。
参数列表：用于接收外部传递给函数的数据，参数可以是多个，也可以没有，多个参数之间用逗号分隔，参数列表又称为形参（形式参数），它仅在函数内部有效。
文档字符串：用于说明函数的功能、参数、返回值等信息，可选填，使用三引号包裹，可通过 help() 函数查看文档字符串。
函数体：函数的核心逻辑代码，包含一系列的语句和表达式，需要注意缩进，与 def 语句保持 4 个空格的缩进量。
return 语句：用于将函数的处理结果返回给函数调用者，可选填，如果省略 return 语句，函数默认返回 None。return 语句执行后，函数会立即终止，后续的代码不会被执行。
python
运行
```
# 定义一个无参数、无返回值的函数
def say_hello():
    """打印欢迎信息"""
    print("欢迎学习 Python 函数！")
    print("祝你学习顺利，学有所成！")

# 定义一个有参数、有返回值的函数（计算两个数的和）
def add_num(a, b):
    """
    计算两个数的和
    参数:
        a：第一个加数
        b：第二个加数
    返回值:
        两个数的和
    """
    result = a + b
```

```
    return result

# 定义一个多参数、多返回值的函数（计算两个数的和与积）
def calculate(a, b):
    """计算两个数的和与积"""
    sum_result = a + b
    mul_result = a * b
    return sum_result, mul_result  # 多返回值以元组形式返回
```

3.2.2 函数的调用

函数定义完成后，不会自动执行，需要通过函数调用的方式来执行函数体中的代码。函数调用的语法格式如下：

plaintext

```
函数名(实参列表)
```

其中，实参（实际参数）是传递给函数的具体数据，实参的数量、类型需要与函数定义中的形参保持一致（特殊情况除外），实参将按照位置顺序或关键字对应关系，传递给对应的形参。

python

运行

```
# 调用无参数、无返回值的函数
say_hello()

# 调用有参数、有返回值的函数
num1 = 10
num2 = 20
sum_num = add_num(num1, num2)
print(f"{num1} + {num2} = {sum_num}")

# 调用多参数、多返回值的函数
num3 = 5
num4 = 8
sum_res, mul_res = calculate(num3, num4)
print(f"{num3} + {num4} = {sum_res}")
print(f"{num3} × {num4} = {mul_res}")

# 查看函数的文档字符串
help(add_num)
help(calculate)
```

3.3 函数的参数类型

Python 中的函数参数类型非常灵活，主要包括位置参数、关键字参数、默认参数、可变位置参数和可变关键字参数，不同的参数类型适用于不同的场景，合理使用可以提高函数的灵活性和易用性。

3.3.1 位置参数

位置参数是最基本的参数类型，它要求实参的顺序与形参的顺序完全一致，实参的数量与形参的数量完全相同，Python 会按照位置顺序，将实参传递给对应的形参。

python
运行
```
# 定义一个使用位置参数的函数（打印学生信息）
def print_student_info(name, age, gender):
    """打印学生信息"""
    print(f"姓名：{name}")
    print(f"年龄：{age}")
    print(f"性别：{gender}")

# 调用函数（位置参数，顺序与形参一致）
print_student_info("张三", 25, "男")

# 错误示例（参数顺序错误）
# print_student_info(25, "张三", "男")  # 输出结果不符合预期
```
3.3.2 关键字参数
关键字参数是通过“形参名 = 实参值”的形式传递参数，它不要求实参的顺序与形参的顺序一致，Python 会根据关键字对应关系，将实参传递给对应的形参。使用关键字参数可以提高代码的可读性，避免因参数顺序错误导致的问题。
python
运行
```
# 调用函数（关键字参数，顺序可任意）
print_student_info(age=26, name="李四", gender="女")

# 混合使用位置参数和关键字参数（位置参数必须在前，关键字参数必须在后）
print_student_info("王五", gender="男", age=24)
```
3.3.3 默认参数
默认参数是在函数定义时，为形参指定一个默认值，当函数调用时，如果没有为该形参传递实参，函数将使用默认值；如果传递了实参，函数将使用传递的实参值。默认参数可以简化函数的调用，减少不必要的实参传递，适用于形参值相对固定的场景。
需要注意的是，默认参数的默认值必须是不可变对象（如整数、字符串、元组），不能使用可变对象（如列表、字典、集合），否则会导致后续函数调用时，默认值被累积修改，引发意想不到的问题。
python
运行
```
# 定义一个使用默认参数的函数（打印学生信息，性别默认值为"男"）
def print_student_info_default(name, age, gender="男"):
    """打印学生信息，性别默认值为男"""
    print(f"姓名：{name}")
    print(f"年龄：{age}")
    print(f"性别：{gender}")

# 调用函数（不传递 gender 参数，使用默认值）
print_student_info_default("赵六", 23)
```

```
# 调用函数（传递 gender 参数，覆盖默认值）
print_student_info_default("钱七", 22, "女")
```

3.3.4 可变位置参数

可变位置参数用于接收任意数量的位置参数，它在函数定义时，以 *args 的形式表示，args 是一个元组，用于存储所有传递进来的可变位置参数。可变位置参数可以接收 0 个或多个实参，适用于函数参数数量不确定的场景。

python
运行

```
# 定义一个使用可变位置参数的函数（计算多个数的累加和）
def sum_nums(*args):
    """计算多个数的累加和"""
    sum_result = 0
    for num in args:
        sum_result += num
    return sum_result

# 调用函数（传递 0 个参数）
print(f"0 个数的累加和：{sum_nums()}")

# 调用函数（传递 3 个参数）
print(f"1 + 2 + 3 = {sum_nums(1, 2, 3)}")

# 调用函数（传递 5 个参数）
print(f"1 + 2 + 3 + 4 + 5 = {sum_nums(1, 2, 3, 4, 5)}")

# 传递列表或元组（需要在前面添加 *，进行解包）
nums_list = [6, 7, 8]
print(f"6 + 7 + 8 = {sum_nums(*nums_list)}")
```

3.3.5 可变关键字参数

可变关键字参数用于接收任意数量的关键字参数，它在函数定义时，以 **kwargs 的形式表示，kwargs 是一个字典，用于存储所有传递进来的可变关键字参数，键为形参名，值为实参值。可变关键字参数可以接收 0 个或多个关键字参数，适用于函数参数数量和名称都不确定的场景。

python
运行

```
# 定义一个使用可变关键字参数的函数（打印任意数量的学生信息）
def print_student_any(**kwargs):
    """打印任意数量的学生信息"""
    for key, value in kwargs.items():
        print(f"{key}：{value}")
    print("-" * 30)

# 调用函数（传递 0 个关键字参数）
print_student_any()
```

```
# 调用函数（传递 2 个关键字参数）
print_student_any(name="孙八", age=27)

# 调用函数（传递 4 个关键字参数）
print_student_any(name="周九", age=28, gender="男", addr="北京")

# 传递字典（需要在前面添加 **，进行解包）
student_dict = {"name": "吴十", "age": 29, "gender": "女", "addr": "上海", "tel": "12345678901"}
print_student_any(**student_dict)
```

3.4 函数的高级特性

3.4.1 函数的嵌套定义

函数的嵌套定义是指在一个函数内部，再次定义另一个函数，外层函数称为外部函数，内层函数称为内部函数。内部函数只能在外部函数内部被调用，无法在外部函数外部直接调用，这种特性可以实现数据的封装和隐藏，避免全局变量的污染。

python
运行

```
# 函数的嵌套定义
def outer_func():
    """外部函数"""
    print("这是外部函数。")
    a = 10

    def inner_func():
        """内部函数"""
        print("这是内部函数。")
        print(f"访问外部函数的变量 a：{a}")
        b = 20
        print(f"内部函数的变量 b：{b}")

    # 外部函数内部调用内部函数
    inner_func()

# 调用外部函数
outer_func()

# 错误示例（外部函数外部无法调用内部函数）
# inner_func()  # 抛出 NameError 异常
```

3.4.2 闭包

闭包是一种特殊的嵌套函数，它满足以下三个条件：

外部函数中定义了内部函数；

内部函数引用了外部函数的变量（非全局变量）；

外部函数返回了内部函数的引用（即返回内部函数本身，不执行内部函数）。

闭包可以保留外部函数的变量状态，即使外部函数执行完毕，内部函数仍然可以访问和修改外部函数的变量，这种特性在实际开发中非常实用，例如用于实现装饰器、工厂函数等。

python
运行

```
# 闭包的实现（创建一个加法器，固定一个加数）
def create_adder(fixed_num):
    """创建一个加法器，固定一个加数"""
    def adder(num):
        """加法器，接收另一个加数，返回和"""
        return fixed_num + num
    # 返回内部函数的引用
    return adder

# 创建一个固定加数为 10 的加法器
adder10 = create_adder(10)
# 创建一个固定加数为 20 的加法器
adder20 = create_adder(20)

# 调用加法器
print(f"10 + 5 = {adder10(5)}")
print(f"20 + 5 = {adder20(5)}")
print(f"10 + 100 = {adder10(100)}")
```

3.4.3 装饰器

装饰器是一种基于闭包实现的高级语法，它用于在不修改原函数代码和调用方式的前提下，为函数添加额外的功能，例如日志记录、性能统计、权限验证等。Python 中的装饰器使用 @ 符号来标识，放在函数定义的上方，语法格式如下：

plaintext

```
@装饰器名
def 函数名(参数列表):
    函数体
```

python
运行

```
# 定义一个简单的装饰器（记录函数的调用日志）
def log_decorator(func):
    """日志装饰器，记录函数的调用信息"""
    def wrapper(*args, **kwargs):
        """包装函数，添加日志功能"""
        print(f"开始调用函数：{func.__name__}")
        result = func(*args, **kwargs)
        print(f"函数 {func.__name__} 调用完毕")
        return result
    return wrapper

# 使用装饰器装饰函数
```

```
@log_decorator
def add(a, b):
    """计算两个数的和"""
    return a + b

@log_decorator
def multiply(a, b):
    """计算两个数的积"""
    return a * b

# 调用装饰后的函数
print(f"3 + 5 = {add(3, 5)}")
print(f"3 × 5 = {multiply(3, 5)}")
```

3.5 函数的递归调用

函数的递归调用是指函数在执行过程中，直接或间接地调用自身。递归调用通常用于解决具有递归结构的问题，例如阶乘计算、斐波那契数列、二叉树遍历等，它可以将复杂的问题分解为简单的、重复的子问题，简化代码的编写。

递归调用必须满足以下两个条件：

递归终止条件：存在一个或多个基本情况，当满足该条件时，递归终止，不再调用自身，否则会形成死递归，导致栈溢出。

递归递推关系：将原问题分解为规模更小的子问题，子问题的解决方法与原问题相同，通过递推逐步逼近递归终止条件。

python

运行

```
# 递归函数（计算 n 的阶乘，n! = n × (n-1) × (n-2) × ... × 1）
def factorial(n):
    """计算 n 的阶乘"""
    # 递归终止条件
    if n == 0 or n == 1:
        return 1
    # 递归递推关系
    return n * factorial(n - 1)

# 调用递归函数
print(f"5! = {factorial(5)}")
print(f"10! = {factorial(10)}")

# 递归函数（计算斐波那契数列的第 n 项，F(0)=0, F(1)=1, F(n)=F(n-1)+F(n-2)）
def fibonacci(n):
    """计算斐波那契数列的第 n 项"""
    # 递归终止条件
    if n == 0:
        return 0
    elif n == 1:
```

```
        return 1
    # 递归递推关系
    return fibonacci(n - 1) + fibonacci(n - 2)

# 打印斐波那契数列的前 10 项
for i in range(10):
    print(f"F({i}) = {fibonacci(i)}", end=" ")
```

需要注意的是，递归调用会占用较多的内存空间（因为每次递归都会在栈中创建新的函数调用帧），对于大规模的问题，递归调用的效率可能较低，此时可以考虑使用迭代的方式来替代递归。

总结

本文详细介绍了 Python 语法的核心内容，涵盖了 Python 基础语法要素、流程控制语句、函数定义与使用三个主要部分。从标识符、关键字、变量、数据类型，到条件判断、循环语句，再到函数的定义、调用、参数类型和高级特性，每个知识点都配有详细的解释和实战代码示例，能够帮助读者快速理解和掌握 Python 语法的核心要点。

Python 语法的学习是一个循序渐进的过程，需要不断地阅读、练习和总结，只有通过大量的实战练习，才能真正掌握 Python 语法的使用技巧，为后续的高级开发（如数据分析、人工智能、Web 开发等）打下坚实的基础。希望本文能够为读者提供有价值的参考和帮助，祝愿读者在 Python 学习的道路上越走越远，取得优异的成绩。