第九章

函数

函数是编程中的基本构造单元，用于将特定的计算或操作封装成独立的代码块，便于重复使用和维护。在商业数据分析中，函数的应用至关重要。通过定义函数，分析人员可以将复杂的数据处理流程模块化，提高代码的可读性和可维护性。例如，使用 Python 函数可以实现数据清洗、特征提取、统计分析等操作，从而提高分析效率和准确性。此外，函数的使用有助于构建可重复的分析流程，确保分析结果的一致性和可靠性。

9.1 抽象的概念及意义

抽象是计算机科学中的核心概念，旨在隐藏复杂的实现细节，突出核心功能，从而提高程序的可读性、可维护性和扩展性。在编程实践中，函数抽象是实现这一目标的主要手段之一。通过将重复的逻辑封装在函数中，开发者可以减少代码冗余，提升代码的模块化程度。

例如，考虑一个需要计算多个矩形面积的场景。如果不使用函数，可能会多次编写相同的计算逻辑：

```
# 计算第一个矩形的面积
width1 = 5
height1 = 10
area1 = width1 * height1

# 计算第二个矩形的面积
width2 = 3
height2 = 7
area2 = width2 * height2

# 计算第三个矩形的面积
width3 = 6
height3 = 9
area3 = width3 * height3
```

上述代码存在明显的重复，增加了维护难度。通过定义一个计算矩形面积的函数，可以有效地抽象出重复的逻辑：

```python
def calculate_area(width, height):
    return width * height

# 使用函数计算面积
area1 = calculate_area(5, 10)
area2 = calculate_area(3, 7)
area3 = calculate_area(6, 9)
```

通过这种方式，计算面积的逻辑被封装在 `calculate_area` 函数中，调用者只需提供不同的参数即可复用该逻辑。这不仅减少了代码冗余，还提高了代码的清晰度和可维护性。

函数抽象在商业数据分析中尤为重要。例如，在数据预处理中，可能需要对多个数据集执行相同的清洗操作。将这些操作封装在函数中，可以确保一致性，并简化后续的分析流程。

总而言之，函数抽象通过将重复的逻辑封装在函数中，减少了代码冗余，提升了程序的模块化程度和可维护性，是编程实践中不可或缺的技术手段。

9.2 自定义函数的定义与调用

函数的定义和调用是程序设计中的基本概念，具有非常重要的作用。函数不仅帮助组织代码，还通过封装重复的逻辑减少了冗余，从而提升代码的可维护性和复用性。函数的定义使用 `def` 关键字，后跟函数名和括号中的参数列表。函数内部的代码块通常由缩进的语句组成，而 `return` 语句用于返回函数的结果。

1. 函数的定义与调用

函数的基本定义格式如下：

```python
def function_name(parameters):
    # 执行的代码块
    return result
```

在定义函数时，`def` 后接函数名，再由圆括号包围的参数列表，可以为函数的参数指定默认值。如果函数没有参数，可以省略。如果函数需要返回一个结果，可以使用 `return` 语句将计算结果返回给调用者。

2. 参数的传递

Python 支持多种参数传递方式，包括位置参数、关键字参数和混合方式。位置参数是最常见的类型，调用时传入的值按照位置匹配到相应的参数。例如：

```python
def add(a, b):
    return a + b

result = add(3, 5)
print(result)  # 输出 8
```

除了位置参数，Python 还支持使用关键字参数来进行函数调用，这使得传递参数时不受位置顺序的限制。例如：

```python
def describe_pet(animal_type, pet_name):
    print(f"I have a {animal_type} named {pet_name}.")
```

```
3
4 describe_pet(animal_type="dog", pet_name="Buddy")
```

3. 混合方式

混合使用位置参数和关键字参数进行函数调用时，为了确保参数传递的明确性和避免歧义，位置参数必须在关键字参数前面。

```
1 def multiply(a, b):
2     return a * b
3
4 print(multiply(4, b = 5))  # 输出 20
5 print(multiply(a = 4, 5))  # 错误，位置参数必须在关键字参数前面
```

4. 返回值的使用

函数可以返回值，`return` 语句用于指定返回值。如果函数没有 `return` 语句，它会默认返回 `None`。返回的值可以用于后续的计算或处理。例如：

```
1 def multiply(a, b):
2     return a * b
3
4 result = multiply(4, 5)
5 print(result)  # 输出 20
```

5. 示例代码：带参数的函数

以下是一个包含多个参数、返回值以及默认参数的完整示例：

```
1  def calculate_area(length, width=1):
2      """计算矩形的面积，宽度参数有默认值"""
3      return length * width
4
5  # 使用位置参数
6  area1 = calculate_area(5, 3)
7  print(f"Area 1: {area1}")  # 输出 Area 1: 15
8
9  # 使用默认参数
10 area2 = calculate_area(5)
11 print(f"Area 2: {area2}")  # 输出 Area 2: 5
```

在此示例中，`width` 参数有默认值，因此在调用 `calculate_area(5)` 时，`width` 会自动取默认值 `1`。

通过函数，重复的计算逻辑可以封装成一个模块，避免了代码的重复性，且使得程序结构更加清晰易懂。

5. 文档字符串

文档字符串（docstring）和函数注解（function annotation）是提高代码可读性和可维护性的关键工具。文档字符串用于描述模块、类或函数的功能和用法，而函数注解用于为函数的参数和返回值提供类型提示。

文档字符串是位于模块、类或函数定义内部的字符串字面量，通常使用三重引号（`""" """`）包裹。其主要作用是为代码提供说明性文档，便于他人理解和使用。根据 PEP 257 的建议，文档字符串应简洁明了，首行应为简短的描述，后续可包含更详细的说明。

示例：

```
def add(a, b):
    """
    返回两个数的和。

    参数：
    a (int)：第一个加数。
    b (int)：第二个加数。

    返回：
    int：两个数的和。
    """
    return a + b
```

在上述示例中，函数 `add` 的文档字符串清晰地描述了函数的功能、参数和返回值。这有助于用户快速理解函数的用途和使用方法。

6. 函数注解（Function Annotation）

函数注解是 Python 3 引入的特性，用于为函数的参数和返回值添加元数据，通常用于类型提示。注解的语法是在参数名后使用冒号加类型提示，返回值注解则在参数列表后使用箭头加类型提示。需要注意的是，函数注解仅用于提供信息，不会影响函数的实际行为。

示例：

```
def add(a: int, b: int) -> int:
    return a + b
```

在此示例中，函数 `add` 的参数 `a` 和 `b` 以及返回值均被注解为整数类型。这为阅读代码的人提供了关于参数和返回值类型的有用信息。

综合示例：

将文档字符串和函数注解结合使用，可以进一步提高代码的可读性和可维护性。

```
def greet(name: str) -> str:
    """
    返回一个问候语。

    参数：
    name (str)：被问候者的名字。

    返回：
    str：问候语。
    """
    return f"Hello, {name}!"
```

在此示例中，函数 `greet` 的文档字符串详细描述了函数的功能、参数和返回值，同时使用函数注解明确了参数和返回值的类型。这种结合使用的方式有助于开发者和用户更好地理解和使用函数。

9.3 自定义函数在商业数据分析中的应用

9.3.1 计算商品销售额

在商品数据分析中，定义和调用自定义函数有助于提高代码的模块化和可读性。以下示例展示了如何通过自定义函数计算商品的总销售额和平均销售额。

```
# 定义函数以计算总销售额
def calculate_total_sales(prices, quantities):
    """
    计算商品的总销售额。

    参数:
    prices (list): 商品价格列表。
    quantities (list): 商品销售数量列表。

    返回:
    float: 总销售额。
    """
    total_sales = 0
    for price, quantity in zip(prices, quantities):
    total_sales += price * quantity
    return total_sales

# 定义函数以计算平均销售额
def calculate_average_sales(prices, quantities):
    """
    计算商品的平均销售额。

    参数:
    prices (list): 商品价格列表。
    quantities (list): 商品销售数量列表。

    返回:
    float: 平均销售额。
    """
    total_sales = calculate_total_sales(prices, quantities)
    total_items = sum(quantities)
    if total_items == 0:
    return 0
    return total_sales / total_items

# 示例数据
product_prices = [10.0, 20.0, 15.0]  # 商品价格列表
product_quantities = [5, 3, 2]       # 商品销售数量列表

# 调用函数计算总销售额
total_sales = calculate_total_sales(product_prices, product_quantities)
print(f"总销售额: {total_sales}")  # 输出: 总销售额: 145.0
```

```
# 调用函数计算平均销售额
average_sales = calculate_average_sales(product_prices, product_quantities)
print(f"平均销售额：{average_sales}")  # 输出：平均销售额：14.5
```

在上述代码中，定义了两个函数：`calculate_total_sales` 和 `calculate_average_sales`。前者计算商品的总销售额，后者计算平均销售额。通过将这些计算逻辑封装在函数中，可以在需要时多次调用，避免代码重复，提高代码的可维护性。

在财务数据分析中，Python 函数的定义与调用提供了强大的工具，尤其是在处理复杂的财务数据时，通过自定义函数能够有效地减少代码冗余、提高可重用性，并使得代码结构更清晰。以下展示了如何定义和调用一个自定义函数，以处理与财务数据相关的计算任务。

9.3.2 计算股票的移动平均线

假设需要计算某只股票的简单移动平均（SMA），可以通过自定义函数来实现这一功能。该函数接收一个数据集和窗口大小作为输入，返回对应的移动平均值。

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# 定义计算简单移动平均的函数
def calculate_sma(data, window_size):
    return data.rolling(window=window_size).mean()

# 假设获取的股票价格数据为以下列表
stock_data = pd.Series([150.7, 152.3, 153.8, 154.2, 156.4, 157.0, 158.5, 160.0, 162.1, 163.4])

# 调用函数，计算窗口大小为3的移动平均
sma_result = calculate_sma(stock_data, 3)

# 可视化结果
plt.plot(stock_data, label="Stock Price")
plt.plot(sma_result, label="Simple Moving Average", linestyle='--')
plt.legend()
plt.show()
```

代码解释：`calculate_sma` 函数接受两个参数：`data`（包含股票价格的数据）和 `window_size`（移动平均的窗口大小）。该函数使用 Pandas 提供的 `rolling().mean()` 方法来计算每个时间点上的移动平均值。在该示例中，使用了一个包含 10 个股票价格数据点的 `pd.Series` 对象，窗口大小设置为 3。函数调用后，结果通过 `matplotlib` 库进行可视化，以便分析股票价格的变化趋势及其对应的移动平均线。

简单移动平均（SMA，Simple Moving Average）：是一种常用的时间序列数据平滑技术，用于分析数据的趋势和波动。它通过计算一个固定时间窗口内数据点的算术平均值来平滑数据，以帮助识别长期趋势，而减少短期波动的影响。SMA 特别适用于金融市场、气候变化预测以及任何具有时间序列特征的数据分析。

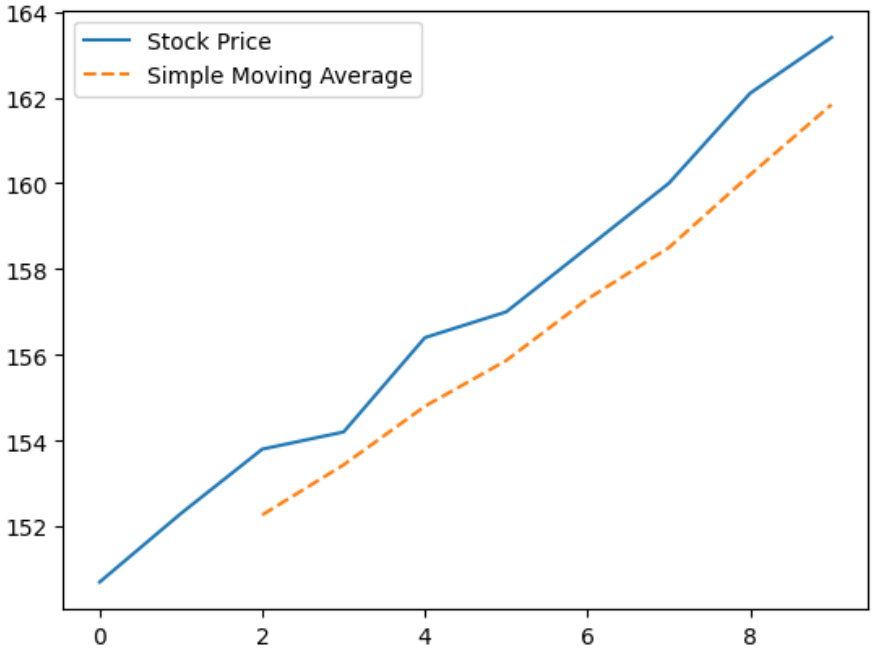

图 9.1: 股票移动平均线

假设有一组连续的时间序列数据，例如股票价格、温度、销售额等，SMA 通过对一段时间内的数据进行平均，计算得到每个时间点的“平均值”。在固定的窗口大小内（例如过去 3 天的价格），每新增一个数据点，窗口就会向前滑动，旧的数据点被移出，新数据点被加入，计算新的平均值。

具体来说，若有一个数据序列 $X = [x_1, x_2, x_3, ..., x_n]$，窗口大小为 k，则第 i 个点的 SMA 值计算公式为：

$$SMA(i) = \frac{1}{k} \sum_{j=i-k+1}^{i} x_j$$

即，SMA 值是窗口中所有数据点的平均值。

假设有一组连续的每日股票价格数据：

$$\text{股票价格} = [100, 105, 110, 115, 120, 125, 130]$$

我们可以计算该数据的简单移动平均，假设选择窗口大小为 3 天。具体步骤如下：

1. 对第 1 到第 3 天的数据取平均：$\frac{100+105+110}{3} = 105$
2. 对第 2 到第 4 天的数据取平均：$\frac{105+110+115}{3} = 110$
3. 对第 3 到第 5 天的数据取平均：$\frac{110+115+120}{3} = 115$
4. 对第 4 到第 6 天的数据取平均：$\frac{115+120+125}{3} = 120$
5. 对第 5 到第 7 天的数据取平均：$\frac{120+125+130}{3} = 125$

因此，计算得到的 SMA 序列为：

$$SMA = [105, 110, 115, 120, 125]$$

简单移动平均（SMA）是一种有效的数据平滑方法，通过计算时间窗口内数据的平均值，帮助识别数据的长期趋势和去除短期波动。在财务数据分析中，SMA 被广泛应用于股市分析、经济数据分析等领域，作为一种基础的趋势分析工具。

结论：自定义函数的应用不仅帮助简化代码结构，还能在实际的数据分析过程中大幅提高效率。特别是在财务数据分析领域，能够灵活地通过自定义函数处理不同类型的计算任务，如财务比率、时间序列分析等，从而为决策提供科学依据。

9.3.3 自定义函数计算财务比率

在财务数据分析的背景下，Python 中的自定义函数提供了一个高效的工具，可以减少重复性工作，提高代码的可维护性和可读性。通过定义一个函数，能够将特定的分析任务封装起来，只需一次定义，便可多次调用，简化代码并降低出错的概率。尤其是在处理大规模的财务数据时，自定义函数能够显著提高工作效率。

以“计算财务比率”为例，假设需要从公司的财务报表中提取并计算几项常见的财务比率，例如“资产负债率”和“流动比率”。通过创建自定义函数，可以重复使用这些函数进行不同公司的数据分析，而不必每次都写重复的代码。

下面是一个简单的代码示例，展示了如何通过自定义函数来计算财务比率：

```
# 定义函数计算资产负债率
def calculate_debt_to_assets(total_debt, total_assets):
    return total_debt / total_assets

# 定义函数计算流动比率
def calculate_current_ratio(current_assets, current_liabilities):
    return current_assets / current_liabilities

# 示例数据
total_debt = 500000
total_assets = 1000000
current_assets = 300000
current_liabilities = 150000

# 调用函数并打印结果
debt_to_assets_ratio = calculate_debt_to_assets(total_debt, total_assets)
current_ratio = calculate_current_ratio(current_assets, current_liabilities)

print(f"资产负债率：{debt_to_assets_ratio:.2f}")
print(f"流动比率：{current_ratio:.2f}")
```

在上述代码中，首先定义了两个函数 `calculate_debt_to_assets` 和 `calculate_current_ratio`，分别用于计算资产负债率和流动比率。每个函数接收相关数据作为参数，返回计算结果。在调用函数时，只需要传入实际的数据，函数会自动执行相应的计算并返回结果。这种封装式的编程方法避免了重复的代码，

并使得后续的数据分析工作更加简洁、高效。

自定义函数在财务数据分析中的作用尤为重要，特别是在需要进行大量数据处理和重复性计算时，它能够显著减少代码冗余，提升编程效率。

9.3.4 计算跨境电商各地区销售额

在进行跨境电商销售数据分析时，可以通过自定义函数来计算每个地区的总销售额。这将有助于了解不同市场的表现，并基于此优化市场营销和物流策略。以下代码示例展示了如何定义和调用自定义函数来进行简单的数据分析：

假设有一个包含电商销售数据的 DataFrame，其中包括产品销售数量、单价和地区信息。我们将定义一个函数来计算某个地区的总销售额。

```
import pandas as pd

# 假设的数据
data = {
    'Region': ['North America', 'Europe', 'Asia', 'North America', 'Asia'],
    'Quantity': [100, 150, 200, 130, 180],
    'UnitPrice': [20, 15, 25, 18, 22]
}

# 将数据加载到DataFrame中
df = pd.DataFrame(data)

# 定义计算销售额的函数
def calculate_sales_by_region(df, region):
    """
    计算指定地区的总销售额
    参数:
    df -- 包含销售数据的DataFrame
    region -- 指定的地区名称
    返回:
    指定地区的总销售额
    """
    # 筛选出指定地区的数据
    region_data = df[df['Region'] == region]

    # 计算销售额 (数量 * 单价)
    region_data['Sales'] = region_data['Quantity'] * region_data['UnitPrice']

    # 返回该地区的总销售额
    total_sales = region_data['Sales'].sum()
    return total_sales

# 调用函数计算北美地区的总销售额
north_america_sales = calculate_sales_by_region(df, 'North America')
print("North America Sales: ", north_america_sales)
```

代码解析：

1. **数据定义与加载**：首先定义一个包含销售数据的字典，并使用 Pandas 将其转换为 DataFrame 格式。

2. **自定义函数**：定义了 `calculate_sales_by_region` 函数，该函数接收两个参数：一个是包含销售数据的 DataFrame，另一个是指定的地区。函数内部通过筛选地区数据并计算数量与单价的乘积来得到销售额，然后求和返回总销售额。

3. **函数调用**：通过调用 `calculate_sales_by_region` 函数，传入'北美'地区的数据，计算并输出该地区的总销售额。

这种方法在跨境电商数据分析中具有广泛的应用。例如，在电商平台中，常常需要对不同市场的销售表现进行跟踪与分析，使用函数可以简化这些任务，提高代码的复用性和可维护性。

9.3.5 社交媒体文本数据清洗

在社交媒体数据分析中，数据清洗是数据预处理的重要步骤之一。社交媒体文本数据通常包含许多噪声，比如不规则的符号、HTML 标签、URLs 以及情感表达等，因此需要进行清洗处理，使其更加规范化，以便后续的文本分析和机器学习任务。使用 Python 进行文本清洗可以有效提高数据的质量和分析的准确性。以下示例演示了如何使用 Python 编写自定义函数来清洗社交媒体数据。假设我们处理的是来自社交媒体平台的评论数据，目标是去除不必要的符号、HTML 标签、URLs，并将文本转化为小写。

```python
# 定义清洗函数
def clean_social_media_text(text):
    # 去除HTML标签
    while '<' in text and '>' in text:
        start = text.find('<')
        end = text.find('>', start) + 1
        text = text[:start] + text[end:]

    # 去除URLs
    words = text.split()
    words = [word for word in words if not word.startswith("http://") and not word.startswith("https://")]
    text = ' '.join(words)

    # 去除特殊字符与标点
    text = ''.join([char if char.isalnum() or char.isspace() else '' for char in text])

    # 转换为小写
    text = text.lower()

    # 去除多余的空白字符
    text = ' '.join(text.split())

    return text

# 示例调用
sample_text = "<p>Check out this amazing product at http://example.com! #bestbuy</p>"
cleaned_text = clean_social_media_text(sample_text)
print(cleaned_text)
```

代码解析：

- **去除 HTML 标签：**通过 `find()` 方法定位 `<` 和 `>` 的位置，使用字符串切片删除标签之间的内容。
- **去除 URLs：**使用 `split()` 方法将文本分割成单词，然后筛选出不是以 `http://` 或 `https://` 开头的单词。然后再用 `' '.join()` 将剩余的单词重新组合成一段文本。
- **去除特殊字符与标点：**通过遍历每个字符，使用 `isalnum()` 方法检查字符是否为字母或数字，若是则保留，若不是则删除。
- **转换为小写：**使用 `lower()` 方法将所有文本转换为小写字母。
- **去除多余的空白字符：**通过 `split()` 方法将文本分割为单词，再使用 `' '.join()` 去除额外的空白字符。

9.4 函数的参数

重要性：★★★； **难易度：**★★★★★

在 Python 中，函数参数的使用非常灵活，支持多种类型的参数传递方式。常见的函数参数类型包括位置参数、默认参数、关键字参数以及不定长参数。下面将详细介绍这些参数类型，并通过代码示例进行说明。

9.4.1 形式参数与实际参数

函数的定义和调用涉及两个关键概念：形式参数（formal parameters）和实际参数（actual parameters）。形式参数是在函数定义时指定的变量，用于接收传入的数据；实际参数则是在函数调用时提供的具体值或表达式。

形式参数（Formal Parameters）

形式参数是函数定义中声明的变量，充当占位符，表示函数预期接收的数据类型和数量。这些参数在函数体内作为局部变量使用，允许函数根据需要访问和操作输入数据。例如，定义一个计算两个数之和的函数：

```
1 def add(x, y):
2     return x + y
```

在此示例中，`x` 和 `y` 是形式参数，表示函数 `add` 预期接收两个输入。

实际参数（Actual Parameters）

实际参数是函数调用时传递给函数的具体值或表达式。它们对应于函数定义中的形式参数，为函数提供所需的数据。例如，调用上述 `add` 函数：

```
1 result = add(3, 5)
```

在此调用中，`3` 和 `5` 是实际参数，与形式参数 `x` 和 `y` 对应。

形式参数与实际参数的关系

当调用函数时，实际参数的值被传递给对应的形式参数。函数体内，形式参数作为局部变量，持有实际参数的值，供函数执行其预定操作。这种机制确保函数能够根据调用时提供的具体数据执行操作。

示例：计算两个数的乘积

```python
def multiply(a, b):
    return a * b

result = multiply(4, 7)
print(result)  # 输出：28
```

在此示例中，`a` 和 `b` 是形式参数，`4` 和 `7` 是实际参数。调用 `multiply(4, 7)` 时，实际参数 `4` 和 `7` 被传递给形式参数 `a` 和 `b`，函数返回它们的乘积 `28`。

9.4.2 位置参数（Positional Arguments）

位置参数是函数定义时指定的参数类型，传入的实参按顺序依次匹配形参。在调用函数时，传入的值将按照参数顺序赋值给函数的形参。

```python
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

greet("Alice", 30)
```

输出：

```
Hello, Alice! You are 30 years old.
```

在这个例子中，`name` 和 `age` 是位置参数，调用函数时依次传入的实参 `"Alice"` 和 `30` 分别赋值给形参 `name` 和 `age`。

9.4.3 默认参数（Default Arguments）

默认参数是函数定义时为参数指定的默认值。如果在调用时没有为该参数提供值，函数将使用默认值。

```python
def greet(name, age=25):
    print(f"Hello, {name}! You are {age} years old.")

greet("Bob")  # 使用默认值
greet("Alice", 30)  # 使用提供的值
```

输出：

```
Hello, Bob! You are 25 years old.
Hello, Alice! You are 30 years old.
```

在这个例子中，`age` 参数有一个默认值 `25`。如果在调用函数时没有传入 `age` 的值，默认值 `25` 将被使用。

9.4.4 关键字参数（Keyword Arguments）

关键字参数允许在调用函数时通过指定参数名称来传递值，这样可以不必关注参数的顺序。这使得函数调用更为清晰。

```python
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

greet(age=40, name="Charlie")  # 关键字参数
```

输出：

```
Hello, Charlie! You are 40 years old.
```

在这个例子中，虽然参数 `name` 和 `age` 的顺序发生了变化，但由于使用了关键字参数，Python 能够正确地将值传递给对应的参数。

关键字参数（Keyword Arguments）允许在函数调用时通过参数名称明确地传递值，从而提高代码的可读性和灵活性。此外，关键字参数还支持为函数参数指定默认值，使函数调用更加简洁。在定义函数时，可以为参数指定默认值。调用函数时，若未提供该参数的值，函数将使用预设的默认值。这在处理具有可选参数的函数时尤为有用。

```python
def create_account(username, password, account_type='standard'):
    print(f"Username: {username}")
    print(f"Password: {password}")
    print(f"Account Type: {account_type}")

    # 调用函数时未指定 account_type 参数，使用默认值
    create_account('user1', 'pass123')

    # 调用函数时指定 account_type 参数，覆盖默认值
    create_account('user2', 'pass456', account_type='premium')
```

输出：

```
Username: user1
Password: pass123
Account Type: standard

Username: user2
Password: pass456
Account Type: premium
```

代码解析：

在上述示例中，函数 `create_account` 定义了三个参数：`username`、`password` 和 `account_type`。其中，`account_type` 具有默认值 `'standard'`。在第一次调用函数时，未提供 `account_type` 参数，函数使用默认值 `'standard'`。在第二次调用函数时，明确指定了 `account_type` 为 `'premium'`，因此覆盖了默认值。

关键字参数的优势：

- 提高可读性：通过在函数调用时明确指定参数名称，代码的意图更加清晰，易于理解。

- 灵活性：关键字参数允许以任意顺序传递参数，避免了必须按照函数定义的参数顺序传递值的限制。
- 简化函数调用：通过为参数指定默认值，调用函数时可以省略那些具有默认值的参数，从而简化函数调用。

9.4.5 不定长参数（Arbitrary Arguments）

当不确定传入函数的参数个数时，可以使用不定长参数。Python 提供了 `*args` 和 `**kwargs` 两种方式来处理这种情况。

- `*args` 用于传递任意数量的位置参数。
- `**kwargs` 用于传递任意数量的关键字参数。

```
def sum_numbers(*numbers):
    total = sum(numbers)
    print(f"Sum: {total}")

sum_numbers(1, 2, 3, 4)  # 传入多个位置参数
```

输出：

```
Sum: 10
```

在这个例子中，`*numbers` 接受了多个位置参数并将它们放入一个元组 `numbers` 中，之后通过 `sum()` 函数计算它们的和。

```
def display_info(**info):
    for key, value in info.items():
        print(f"{key}: {value}")

display_info(name="David", age=45, job="Engineer")
```

输出：

```
name: David
age: 45
job: Engineer
```

在这个例子中，`**info` 接受了多个关键字参数并将它们存储为一个字典 `info`，可以通过 `items()` 方法遍历字典中的每一对键值。

9.5 局部变量与全局变量

重要性：★★★★； **难易度：**★★★★

在 Python 编程中，变量的作用域决定了变量的可访问范围。主要分为局部变量和全局变量两种类型。

1. 局部变量（Local Variables）

局部变量是在函数内部定义的变量，其作用域仅限于该函数内部。当函数被调用时，局部变量被创建；函数执行结束后，局部变量被销毁。局部变量在函数外部无法访问。

```python
def example_function():
    local_var = "I am a local variable"
    print(local_var)

example_function()
print(local_var)  # 试图在函数外部访问局部变量
```

输出：

```
I am a local variable


Traceback (most recent call last):
 File "script.py", line 5, in <module>
  print(local_var)  # 试图在函数外部访问局部变量
NameError: name 'local_var' is not defined
```

解析：

在上述示例中，`local_var` 是在函数 `example_function` 内部定义的局部变量。在函数内部打印该变量时，输出正常。然而，当尝试在函数外部访问 `local_var` 时，Python 抛出 `NameError`，提示未定义该变量。这表明局部变量的作用域仅限于定义它的函数内部。

2. 全局变量（Global Variables）

全局变量是在函数外部定义的变量，其作用域覆盖整个模块。全局变量可以在函数内部和外部访问。然而，在函数内部如果需要修改全局变量的值，必须使用 `global` 关键字声明，否则 Python 会将其视为新的局部变量。

```python
global_var = "I am a global variable"

def example_function():
    print(global_var)  # 在函数内部访问全局变量

example_function()
print(global_var)  # 在函数外部访问全局变量
```

输出：

```
I am a global variable
I am a global variable
```

解析：

在上述示例中，`global_var` 是在函数外部定义的全局变量。在函数 `example_function` 内部和外部均可访问该变量，且输出结果一致。

3. 在函数内部修改全局变量

如果需要在函数内部修改全局变量的值，必须使用 `global` 关键字声明该变量。否则，Python 会在函数内部创建一个同名的局部变量，而不会影响全局变量的值。

```python
global_var = "I am a global variable"

def example_function():
    global global_var
    global_var = "I have been modified"
    print(global_var)

example_function()
print(global_var)  # 检查全局变量是否被修改
```

输出：

```
I have been modified
I have been modified
```

解析：

在上述示例中，使用`global`关键字声明`global_var`，表示在函数内部对全局变量进行修改。因此，函数内部和外部的`global_var`值均被修改。

4. 变量遮蔽（variable shadowing）

变量遮蔽（variable shadowing）是指在不同的作用域中使用相同的变量名称，导致内层作用域的变量覆盖（或“遮蔽”）外层作用域的同名变量。这种情况可能引发意外的行为和难以调试的错误，因此在编写代码时需谨慎处理。

变量遮蔽的示例

```python
x = 10  # 全局变量

def outer_function():
    x = 20  # 外层函数的局部变量

    def inner_function():
        x = 30  # 内层函数的局部变量
        print(f"内层函数中的 x: {x}")

    inner_function()
    print(f"外层函数中的 x: {x}")

outer_function()
print(f"全局作用域中的 x: {x}")
```

输出：

```
内层函数中的 x: 30
外层函数中的 x: 20
全局作用域中的 x: 10
```

解析：

在上述示例中，变量 `x` 在全局作用域、外层函数和内层函数中分别被定义。每个作用域中的 `x` 相互独立，内层作用域的 `x` 遮蔽了外层作用域的同名变量。为避免这种情况，建议在不同作用域中使用不同的变量名称，以提高代码的可读性和维护性。

避免变量遮蔽的方法

(1) **使用不同的变量名称**：在不同的作用域中，采用具有描述性的变量名称，避免使用相同的名称，从而减少混淆。

(2) **使用 `global` 关键字**：当需要在函数内部修改全局变量时，使用 `global` 关键字声明该变量。例如：

```
    x = 10  # 全局变量

def modify_global():
    global x
    x = 20  # 修改全局变量
    print(f"函数内部的 x: {x}")

modify_global()
print(f"全局作用域中的 x: {x}")
```

输出：

```
函数内部的 x: 20
全局作用域中的 x: 20
```

(3) **使用 `nonlocal` 关键字**：在嵌套函数中，若需要在内层函数中修改外层（非全局）函数的变量，可使用 `nonlocal` 关键字声明该变量。例如：

```
def outer_function():
    x = 10  # 外层函数的局部变量

    def inner_function():
        nonlocal x
        x = 20  # 修改外层函数的变量
        print(f"内层函数中的 x: {x}")

    inner_function()
    print(f"外层函数中的 x: {x}")

outer_function()
```

输出：

```
内层函数中的 x: 20
外层函数中的 x: 20
```

9.6 函数式编程

函数式编程（Functional Programming）是一种编程范式，强调使用纯函数进行计算，避免副作用，并鼓励函数作为一等公民。在 Python 中，尽管其主要是面向对象的，但也提供了对函数式编程的支持。

1. 纯函数

纯函数是指在相同输入下总是产生相同输出且没有副作用的函数。这意味着函数的执行不依赖于外部状态，也不改变外部状态。

示例：

```
def add(a, b):
    return a + b
```

上述函数 `add` 在相同的输入 `a` 和 `b` 下，总是返回相同的结果 `a + b`，且不影响外部状态，因此是一个纯函数。

2. 高阶函数

高阶函数是指接受一个或多个函数作为参数，或返回一个函数作为结果的函数。Python 内置的 `map`、`filter` 和 `reduce` 函数就是高阶函数的典型例子。

示例：

```
# 使用 map 函数将列表中的每个元素平方
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x**2, numbers))
print(squared_numbers)  # 输出：[1, 4, 9, 16, 25]
```

在此示例中，`map` 函数接受一个匿名函数 `lambda x: x**2` 和一个列表 `numbers`，返回一个新的迭代器 `map`，其中每个元素都是原列表元素的平方。

9.6.1 高阶函数 map、filter、reduce

高阶函数是指接受一个或多个函数作为参数，或返回一个函数作为结果的函数。其中，`map`、`filter` 和 `reduce` 是常用的高阶函数，广泛应用于数据处理和函数式编程。

1. `map` 函数

`map` 函数用于将指定的函数依次作用于可迭代对象的每个元素，返回一个包含结果的迭代器。其语法为：

```
map(function, iterable, ...)
```

- `function`：应用于每个元素的函数。
- `iterable`：一个或多个可迭代对象。

示例：

```
# 将列表中的每个元素平方
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(lambda x: x**2, numbers)
print(list(squared_numbers))  # 输出：[1, 4, 9, 16, 25]
```

在此示例中，`map` 函数将匿名函数 `lambda x: x**2` 应用于列表 `numbers` 的每个元素，返回其平方值的迭代器。使用 `list` 函数将其转换为列表后，输出结果为 `[1, 4, 9, 16, 25]`。

2. `filter` 函数

`filter` 函数用于过滤可迭代对象中的元素，保留使指定函数返回 `True` 的元素，返回一个迭代器。其语法为：

```
filter(function, iterable)
```

- `function`：用于判断每个元素是否保留的函数，返回布尔值。
- `iterable`：可迭代对象。

示例：

```
# 过滤出列表中的偶数
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers))  # 输出：[2, 4, 6]
```

在此示例中，`filter` 函数将匿名函数 lambda x: x % 2 == 0 应用于列表 `numbers` 的每个元素，保留偶数元素。使用 `list` 函数将其转换为列表后，输出结果为 `[2, 4, 6]`。

3. `reduce` 函数

`reduce` 函数用于对可迭代对象中的元素进行累积操作，依次将前两个元素传递给指定函数，得到结果后，再与下一个元素继续进行累积，直到处理完所有元素，返回最终结果。在 Python 3 中，`reduce` 函数位于 `functools` 模块中，需要先导入。其语法为：

```
from functools import reduce
reduce(function, iterable[, initializer])
```

- `function`：用于累积操作的函数，接受两个参数。
- `iterable`：可迭代对象。
- `initializer`（可选）：初始值，若提供，则首先与可迭代对象的第一个元素进行累积。

示例：

```
from functools import reduce

# 计算列表元素的累积乘积
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # 输出：120
```

在此示例中，`reduce` 函数将匿名函数 `lambda x, y: x * y` 依次应用于列表 `numbers` 的元素，计算其累积乘积。最终结果为 `120`。

9.6.2 lambda 表达式

lambda 表达式用于创建匿名函数，即没有名称的简短函数。其语法形式为：

```
lambda 参数列表：表达式
```

其中，参数列表可以包含多个参数，表达式是对这些参数进行操作并返回结果。lambda 表达式常用于需要简短函数且不想正式定义函数的场景。

示例 1：基本用法

```
# 定义一个lambda表达式，将输入值加10
add_ten = lambda x: x + 10
print(add_ten(5))  # 输出：15
```

在此示例中，定义了一个 lambda 表达式 `add_ten`，接受一个参数 `x`，返回 `x` 加 10 的结果。调用 `add_ten(5)` 时，输出为 15。

示例 2：与内置函数结合使用

lambda 表达式常与 Python 内置的高阶函数如 `map`、`filter` 和 `reduce` 结合使用。

```
# 使用lambda表达式和map函数，将列表中的每个元素平方
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x**2, numbers))
print(squared_numbers)  # 输出：[1, 4, 9, 16, 25]
```

在此示例中，`map` 函数接受一个 lambda 表达式和一个列表 `numbers`，返回一个新的迭代器，其中每个元素都是原列表元素的平方。使用 `list` 函数将其转换为列表后，输出为 `[1, 4, 9, 16, 25]`。

示例 3：作为函数的返回值

lambda 表达式也可作为函数的返回值，创建闭包。

```
def make_incrementor(n):
    return lambda x: x + n

increment_by_5 = make_incrementor(5)
print(increment_by_5(10))  # 输出：15
```

在此示例中，函数 `make_incrementor` 返回一个 lambda 表达式，该表达式接受一个参数 `x`，返回 `x` 加上外部参数 `n` 的结果。调用 `make_incrementor(5)` 生成一个新的函数 `increment_by_5`，调用 `increment_by_5(10)` 时，输出为 15。

9.7 函数式编程在商业数据分析中的应用

9.7.1 商品销售数据分析

在商品销售数据分析中，常需要对大量数据进行清洗、转换和汇总。Python 提供的 `lambda` 表达式和高阶函数（如 `map`、`filter` 和 `reduce`）可用于简化这些操作。

假设有一个包含商品销售记录的列表，每条记录是一个字典，包含商品名称、单价和销售数量。目标是计算所有商品的销售总额。

```
from functools import reduce

# 商品销售数据
sales_data = [
```

```
{'product': 'A', 'price': 100, 'quantity': 30},
{'product': 'B', 'price': 200, 'quantity': 20},
{'product': 'C', 'price': 150, 'quantity': 50},
{'product': 'D', 'price': 300, 'quantity': 10},
]

# 计算每个商品的销售额
sales_amounts = map(lambda item: item['price'] * item['quantity'], sales_data)

# 过滤销售额大于5000的商品
filtered_sales = filter(lambda amount: amount > 5000, sales_amounts)

# 计算总销售额
total_sales = reduce(lambda x, y: x + y, filtered_sales)

print(f'总销售额: {total_sales}')
```

代码解析：

- **数据准备：**定义一个包含商品销售数据的列表，每个元素是一个字典，包含商品名称、单价和销售数量。
- **计算销售额：**使用 `map` 函数和 `lambda` 表达式，计算每个商品的销售额（单价乘以数量），生成一个销售额的迭代器。
- **过滤高销售额商品：**使用 `filter` 函数和 `lambda` 表达式，过滤出销售额大于 5000 的商品。
- **计算总销售额：**使用 `reduce` 函数和 `lambda` 表达式，累加过滤后的销售额，得到总销售额。
- **输出结果：**打印总销售额。

此示例展示了如何在商品销售数据分析中，综合应用 `lambda` 表达式和高阶函数 `map`、`filter`、`reduce`，以简洁、高效地处理数据。

9.7.2 税后净收入计算

在会计数据分析中，常需要对收入数据进行处理，以计算税后净收入。Python 提供的 `lambda` 表达式和高阶函数（如 `map`、`filter` 和 `reduce`）可用于简化这些操作。

假设有一个包含收入记录的列表，每条记录是一个字典，包含收入金额和税率。目标是计算所有收入的税后净收入总和。

```
from functools import reduce

# 收入数据
income_data = [
{'amount': 5000, 'tax_rate': 0.1},
{'amount': 7000, 'tax_rate': 0.15},
{'amount': 6000, 'tax_rate': 0.2},
{'amount': 8000, 'tax_rate': 0.25},
```

```
]

# 计算每笔收入的税后净收入
net_incomes = map(lambda item: item['amount'] * (1 - item['tax_rate']), income_data)

# 过滤税后净收入大于5000的记录
filtered_net_incomes = filter(lambda net: net > 5000, net_incomes)

# 计算税后净收入总和
total_net_income = reduce(lambda x, y: x + y, filtered_net_incomes)

print(f'税后净收入总和: {total_net_income}')
```

代码解析:

- **数据准备:**定义一个包含收入数据的列表,每个元素是一个字典,包含收入金额和税率。
- **计算税后净收入:**使用 `map` 函数和 `lambda` 表达式,计算每笔收入的税后净收入(收入金额乘以 `1 - 税率`),生成一个税后净收入的迭代器。
- **过滤高净收入记录:**使用 `filter` 函数和 `lambda` 表达式,过滤出税后净收入大于 5000 的记录。
- **计算净收入总和:**使用 `reduce` 函数和 `lambda` 表达式,累加过滤后的税后净收入,得到总和。
- **输出结果:**打印税后净收入总和。

此示例展示了如何在会计数据分析中,综合应用 `lambda` 表达式和高阶函数 `map`、`filter`、`reduce`,以简洁、高效地处理数据。

9.7.3 财务报表数据的关键指标计算

在财务报表分析中,常需要从大量数据中提取关键财务指标,以评估企业的财务状况。Python 提供的 `lambda` 表达式和高阶函数(如 `map`、`filter` 和 `reduce`)可用于简化这些数据处理任务。

假设有一个包含多家公司财务数据的列表,每条记录是一个字典,包含公司名称、收入(revenue)和净利润(net_income)。目标是计算每家公司的净利润率,并筛选出净利润率高于 15% 的公司。

```
from functools import reduce

# 财务数据
financial_data = [
{'company': 'Company A', 'revenue': 1000000, 'net_income': 200000},
{'company': 'Company B', 'revenue': 1500000, 'net_income': 100000},
{'company': 'Company C', 'revenue': 500000, 'net_income': 80000},
{'company': 'Company D', 'revenue': 2000000, 'net_income': 500000},
]

# 计算每家公司的净利润率
profit_margins = map(lambda x: {'company': x['company'], 'profit_margin': x['net_income'] / x['revenue']},
      financial_data)
```

```
# 筛选出净利润率高于15%的公司
high_margin_companies = filter(lambda x: x['profit_margin'] > 0.15, profit_margins)

# 提取公司名称列表
company_names = map(lambda x: x['company'], high_margin_companies)

# 将公司名称列表转换为字符串
result = reduce(lambda x, y: x + ', ' + y, company_names)

print(f'净利润率高于15%的公司有：{result}')
```

代码解析：

- **数据准备：**定义一个包含多家公司财务数据的列表，每个元素是一个字典，包含公司名称、收入和净利润。
- **计算净利润率：**使用 `map` 函数和 `lambda` 表达式，计算每家公司的净利润率（净利润除以收入），生成一个包含公司名称和净利润率的迭代器。
- **筛选高净利润率公司：**使用 `filter` 函数和 `lambda` 表达式，筛选出净利润率高于 15% 的公司。
- **提取公司名称列表：**使用 `map` 函数和 `lambda` 表达式，从筛选结果中提取公司名称。
- **生成结果字符串：**使用 `reduce` 函数和 `lambda` 表达式，将公司名称列表连接成一个字符串，方便输出。
- **输出结果：**打印净利润率高于 15% 的公司名称。

此示例展示了如何在财务报表数据分析中，综合应用 `lambda` 表达式和高阶函数 `map`、`filter`、`reduce`，以简洁、高效地处理数据，提取关键财务指标。

9.7.4 文本数据清洗与词频统计

在处理大规模文本数据时，预处理步骤至关重要。Python 提供的 `lambda` 表达式和高阶函数（如 `map`、`filter` 和 `reduce`）可用于简洁高效地执行文本清洗和词频统计等任务。

假设有一个包含多条文本记录的列表，目标是对每条文本进行清洗，包括转换为小写、移除标点符号和数字，然后统计所有文本中每个词的出现频率。

```
from functools import reduce
import string

# 示例文本数据
texts = [
"Hello World! This is a test.",
"Python is great. 123 numbers should be removed.",
"Data Science is the future; let's learn it.",
]
```

```
# 定义标点符号和数字字符集
punctuations = string.punctuation + string.digits

# 文本清洗函数：转换为小写，移除标点符号和数字
def clean_text(text):
return ''.join(filter(lambda char: char not in punctuations, text.lower()))

# 对每条文本进行清洗
cleaned_texts = map(clean_text, texts)

# 将清洗后的文本拆分为单词列表
words_lists = map(lambda text: text.split(), cleaned_texts)

# 合并所有单词列表为一个列表
all_words = reduce(lambda x, y: x + y, words_lists)

# 统计每个词的频率
word_freq = {}
for word in all_words:
word_freq[word] = word_freq.get(word, 0) + 1

print(word_freq)
```

代码解析：

- **数据准备：**定义一个包含多条文本记录的列表。
- **定义标点符号和数字字符集：**使用 `string.punctuation` 和 `string.digits` 获取所有标点符号和数字字符。
- **文本清洗函数：**定义函数 `clean_text`，将文本转换为小写，并移除标点符号和数字。
- **清洗文本：**使用 `map` 函数，将 `clean_text` 函数应用于每条文本，生成清洗后的文本迭代器。
- **拆分单词：**使用 `map` 函数，将清洗后的每条文本拆分为单词列表。
- **合并单词列表：**使用 `reduce` 函数，将所有单词列表合并为一个列表。
- **统计词频：**遍历合并后的单词列表，统计每个词的出现频率，存储在字典 `word_freq` 中。
- **输出结果：**打印词频统计结果。

此示例展示了如何在大规模文本数据预处理中，综合应用 `lambda` 表达式和高阶函数 `map`、`filter`、`reduce`，以简洁、高效地完成文本清洗和词频统计任务。

9.8 递归函数

递归是一种通过重复调用自身来解决问题的方法，适用于具有自相似结构的问题，即可以分解为若干规模缩小的相似子问题。递归函数是一类通过调用自身来解决问题的函数。递归函数特别适用于以下几类

问题：

- **分治问题**：这类问题可以划分为若干小规模的同类型子问题，并通过递归解决后合并结果。例如，归并排序（Merge Sort）和快速排序（Quick Sort）都通过递归分解排序任务，分别解决子数组的排序问题。
- **树形或图形结构遍历**：树结构的遍历（如二叉树的前序、中序和后序遍历）天然适合递归。通过递归，函数可以直接处理每个节点并对其子节点继续递归调用。图结构的遍历如深度优先搜索（DFS）也可通过递归实现。
- **回溯问题**：对于路径选择和解空间搜索问题，递归是实现回溯算法的自然方式。例如，解决数独问题、生成排列组合以及八皇后问题等，递归函数能更清晰地表达解空间的遍历过程。
- **数学归纳类问题**：像阶乘、斐波那契数列、幂运算等问题，具有明显的数学递归定义，因此使用递归实现代码简洁明了。

9.8.1 递归函数的定义与实现

递归函数的基本结构通常如下所示：

```
def recursive_function(parameters):
    if base_case_condition:  # 基线条件
        return base_case_value  # 返回基线值
    else:
        return recursive_function(modified_parameters)  # 递归调用
```

为了避免无限循环和栈溢出，递归函数必须定义基线条件（Base Case）和递归条件（Recursive Case）。

- **基线条件**：基线条件是递归的终止条件，当问题规模足够小且可直接解决时，递归停止并返回结果。没有基线条件，递归会一直进行，导致栈溢出。
- **递归条件**：递归条件指函数在某些情况下调用自身，解决更小规模的子问题，直到达到基线条件。

示例 1：计算阶乘

阶乘是一个经典的递归问题，定义为：

- $0! = 1$
- $n! = n \times (n-1)!$，其中 $n > 0$

递归实现阶乘函数的代码如下：

```
def factorial(n):
    if n <= 1:  # 基线条件
        return 1
    else:  # 递归条件
        return n * factorial(n - 1)
```

在这个实现中：

- **基线条件**是当 $n \leq 1$ 时，返回 1。
- **递归条件**是返回 $n \times factorial(n-1)$，每次将问题规模减小 1。

示例 2：斐波那契数列

斐波那契数列是另一个经典的递归问题，其定义为：

- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$，其中 $n > 1$

递归实现斐波那契数列的代码如下：

```
def fibonacci(n):
    if n <= 0:  # 基线条件
        return 0
    elif n == 1:  # 基线条件
        return 1
    else:  # 递归条件
        return fibonacci(n - 1) + fibonacci(n - 2)
```

在此实现中：

- **基线条件**分别是当 $n = 0$ 时返回 0，当 $n = 1$ 时返回 1。
- **递归条件**是返回 $fibonacci(n-1) + fibonacci(n-2)$，通过递归逐步减小问题规模。

9.8.2 使用递归函数的注意事项

递归函数需要定义至少一个基线条件和至少一个递归条件。基线条件保证递归能够终止，而递归条件则确保递归在每次调用中向基线条件靠近。如果递归条件未能逐步接近基线条件，程序将陷入无限递归，最终导致栈溢出。

在使用递归时，还需注意以下方面：

1. **基线条件（终止条件）**：确保函数定义了正确的基线条件，避免无限递归。基线条件是递归终止的必要条件，缺少基线条件可能导致栈溢出。
2. **递归深度和性能**：递归会占用调用栈，每次递归调用都将使用额外的栈空间。因此，递归深度过大会导致栈溢出。Python 默认的递归深度是有限的，深度递归会触发 `RecursionError`。可以考虑通过迭代、缓存或动态规划来替代深度递归。
3. **重复计算和效率优化**：某些递归算法可能导致大量重复计算，例如在计算斐波那契数列时，可对已计算的结果进行缓存（如通过 `functools.lru_cache` 实现）来提升效率。这种优化技术称为记忆化（Memoization），能显著降低时间复杂度。
4. **可读性和平衡性**：在某些情况下，递归函数虽然简洁，但若过度依赖递归，可能会降低代码的可读性。编写时应权衡递归和迭代，选择更直观、效率更高的实现方式。

9.8.3 递归函数实例

1. 检查字符串是否是回文

回文是指从前往后读和从后往前读都相同的字符串。递归可以用于检查字符串是否为回文。

```python
def is_palindrome(s):
    if len(s) <= 1:  # 基线条件
        return True
    elif s[0] != s[-1]:  # 如果首尾字符不同，则不是回文
        return False
    else:
        return is_palindrome(s[1:-1])  # 去掉首尾字符，继续检查子字符串

# 示例
print(is_palindrome("racecar"))  # 输出：True
print(is_palindrome("hello"))    # 输出：False
```

在此函数中，基线条件是当字符串长度小于等于 1 时返回 `True`。递归条件是首先检查首尾字符是否相同，然后递归检查去掉首尾字符的子字符串，直到满足基线条件。

2. 找出长度为 n 的梵文长短音组合

在梵文中，短音节 S 占一个长度单位，长音节 L 占两个长度单位，请定义函数，找出所有可能的长短音节组合方式，使得组合之后的结果长度为 n。例如 $V_4 = \{LL, SSL, SLS, LSS, SSSS\}$，$V_4$ 可进一步划分为两个子集合：$\{LL, LSS\}$ (将 L 与 $V_2 = \{L, SS\}$ 中的每个元素组合可得)，$\{SSL, SLS, SSSS\}$ (将 S 与 $V_3 = \{SL, LS, SSS\}$ 中的每个元素组合可得)。

```python
def virahanka(n):
    # 第一种基本情况
    if n == 0:
        return [""]
    # 第二种基本情况
    elif n == 1:
        return ["S"]
    else:
        s = ["S" + prosody for prosody in virahanka(n-1)]
        l = ["L" + prosody for prosody in virahanka(n-2)]
        return s + l
```

3. 汉诺塔游戏

将图9.2中柱子 A 上的 n 个盘子移动到柱子 C 上，需遵守如下规则:

- 一次只能移动一个圆盘；
- 只能移动最顶部的圆盘；
- 大圆盘必须在小圆盘下；

请思考所需次数最少的移动步骤。点我在线试玩。

最小步数的基本思路类似于要把大象装冰箱，拢共分三步 ($n \geq 2$)：

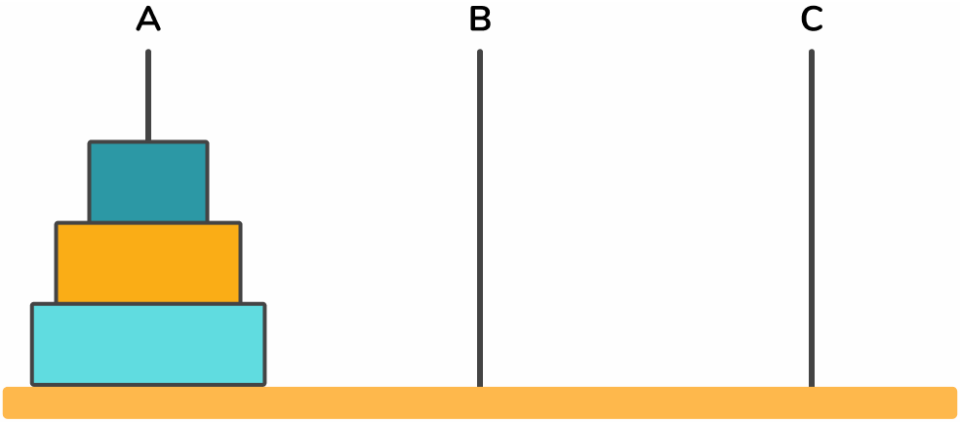

图 9.2: 汉诺塔游戏 ($n = 3$)

- 把前 $n-1$ 个盘子放到 B 柱 (把冰箱门打开);
- 把第 n 个盘子放到 C 柱 (把大象装冰箱);
- 把前 $n-1$ 个盘子从 B 柱放到 C 柱 (把冰箱门带上)。

```
def move(n, left, center, right):
    if n == 1:
        print('{} ---> {}'.format(left, right))
    else:
        move(n-1, left, right, center)
        move(1, left, center, right)
        move(n-1, center, left, right)
```

对于 n 个盘子,最少步数为 $2^n - 1 = 2 \times (2^{n-1} - 1) + 1$, $(2^{n-1} - 1)$ 对应上面的第一步和第三步,1 对应第二步。

4. 递归遍历目录获取所有文件

在文件系统操作中,递归函数常用于遍历目录结构,以获取指定文件夹及其子文件夹中的所有文件。以下示例展示了如何使用递归函数实现这一功能。

```
import os

def list_all_files(directory):
    """
    递归获取指定目录及其子目录中的所有文件。

    参数:
    directory (str): 目标目录的路径。

    返回:
    list: 包含所有文件路径的列表。
    """
    files_list = []
    for entry in os.scandir(directory):
```

```
        if entry.is_file():
            files_list.append(entry.path)
        elif entry.is_dir():
            files_list.extend(list_all_files(entry.path))
    return files_list

# 示例用法
target_directory = '/path/to/your/folder'  # 替换为实际目录路径
all_files = list_all_files(target_directory)
for file_path in all_files:
print(file_path)
```

在上述代码中，`list_all_files` 函数接受一个参数：目标目录的路径 `directory`。函数通过递归遍历目录及其子目录，获取所有文件的路径，并将其添加到列表中。最终，函数返回包含所有文件路径的列表。

此方法利用了 Python 的 `os` 模块中的 `scandir` 函数，该函数提供了高效的目录遍历能力。

5. 递归查找指定后缀文件

在文件系统操作中，递归函数常用于遍历目录结构，以查找特定类型的文件。以下示例展示了如何使用递归函数在指定文件夹中查找所有具有特定后缀的文件。

```
import os

def find_files_with_extension(directory, extension):
    """
    递归查找指定目录及其子目录中具有特定后缀的所有文件。

    参数:
    directory (str): 目标目录的路径。
    extension (str): 目标文件的后缀（例如 '.txt'）。

    返回:
    list: 包含所有符合条件的文件路径的列表。
    """
    matching_files = []
    for entry in os.scandir(directory):
        if entry.is_file() and entry.name.endswith(extension):
            matching_files.append(entry.path)
        elif entry.is_dir():
            matching_files.extend(find_files_with_extension(entry.path, extension))
    return matching_files

# 示例用法
target_directory = '/path/to/your/folder'  # 替换为实际目录路径
file_extension = '.txt'  # 替换为所需的文件后缀
result_files = find_files_with_extension(target_directory, file_extension)
for file_path in result_files:
print(file_path)
```

在上述代码中，`find_files_with_extension` 函数接受两个参数：目标目录的路径 `directory`

和目标文件的后缀 `extension`。函数通过递归遍历目录及其子目录，查找所有以指定后缀结尾的文件，并将其路径添加到列表中。最终，函数返回包含所有符合条件的文件路径的列表。

此方法利用了 Python 的 `os` 模块中的 `scandir` 函数，该函数提供了高效的目录遍历能力。

6. 爬楼梯问题的递归解决方案

在计算机科学中，经典的“爬楼梯问题”常用于演示递归函数的应用。该问题描述如下：一个人站在楼梯底部，目标是到达第 n 阶，每次可以选择迈上一级或两级台阶。需要计算从底部到达第 n 阶共有多少种不同的方式。

```
def climb_stairs(n):
    """
    计算到达第 n 阶楼梯的不同方式数。

    参数:
    n (int): 楼梯的总阶数。

    返回:
    int: 不同的方式数。
    """
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    elif n == 2:
        return 2
    else:
        return climb_stairs(n - 1) + climb_stairs(n - 2)

# 示例用法
stairs = 5
print(f"到达第 {stairs} 阶楼梯的不同方式数为: {climb_stairs(stairs)}")
```

在上述代码中，函数 `climb_stairs` 采用递归方式计算到达第 n 阶楼梯的不同方式数。其逻辑如下：

- **基线条件**：
  - 如果 $n \leq 0$，返回 0，表示无效的楼梯阶数。
  - 如果 $n = 1$，返回 1，表示只有一种方式，即迈上一阶。
  - 如果 $n = 2$，返回 2，表示有两种方式：一次迈上一阶两次，或一次迈上两阶。
- **递归条件**：对于 $n > 2$ 的情况，到达第 n 阶的方式数等于到达第 $n-1$ 阶的方式数与到达第 $n-2$ 阶的方式数之和。

该递归关系类似于斐波那契数列的定义。然而，需要注意的是，纯递归方法在计算较大 n 值时效率较低，可能导致大量重复计算。为提高效率，可采用动态规划或记忆化技术来优化。

7. 递归求解 0-1 背包问题

在计算机科学中，经典的背包问题（Knapsack Problem）是一个组合优化问题，旨在在给定容量的背包中选择若干物品，使得总价值最大化。该问题可通过递归函数求解，以下示例展示了如何使用递归方法解决 0-1 背包问题。

```
def knapsack_recursive(values, weights, capacity, n):
    """
    递归求解0-1背包问题。

    参数:
    values (list): 物品的价值列表。
    weights (list): 物品的重量列表。
    capacity (int): 背包的容量。
    n (int): 可选物品的数量。

    返回:
    int: 背包能获取的最大价值。
    """
    # 基线条件：无物品或容量为零
    if n == 0 or capacity == 0:
        return 0

    # 如果第n个物品的重量超过当前容量，则不选该物品
    if weights[n-1] > capacity:
        return knapsack_recursive(values, weights, capacity, n-1)
    else:
        # 计算不选和选第n个物品的价值，取两者中的最大值
        return max(
        knapsack_recursive(values, weights, capacity, n-1),
        values[n-1] + knapsack_recursive(values, weights, capacity - weights[n-1], n-1)
        )

# 示例用法
values = [60, 100, 120]
weights = [10, 20, 30]
capacity = 50
n = len(values)
max_value = knapsack_recursive(values, weights, capacity, n)
print(f"背包能获取的最大价值为：{max_value}")
```

在上述代码中，函数 `knapsack_recursive` 采用递归方式求解 0-1 背包问题。其逻辑如下：

- **基线条件**：当物品数量 `n` 为 0 或背包容量 `capacity` 为 0 时，返回价值 0。
- **递归条件**：
  - 如果当前物品的重量超过背包剩余容量，则跳过该物品，递归求解剩余物品。
  - 否则，计算不选和选当前物品两种情况下的总价值，取两者中的最大值。

需要注意的是，纯递归方法在处理较大规模问题时可能导致大量重复计算，影响效率。为提高性能，可采用动态规划或记忆化技术进行优化。