第七章

字典

Python 中的字典（Dictionary）是一种内置的数据结构，用于存储键值对（key-value pairs），其中每个键（key）都是唯一且不可变的。字典可以快速访问、添加、修改和删除数据，因此非常适合处理动态数据和大型数据集。这种结构在数据分析中具有重要作用，因为它能够高效存储和查找数据，并以键值对的形式灵活地组织复杂信息。例如，在财务数据分析中，可以使用字典存储不同资产及其价格或属性，快速实现数据的查询和更新操作。此外，字典还支持嵌套结构和丰富的方法，使其能够轻松管理多层次的数据。

7.1 创建字典

重要性：★★★★★；　**难易度：**★

字典是一种用于存储键值对（key-value pairs）的数据结构，每个键都是唯一且不可变的对象（如字符串、整数、元组等）。这种结构使得字典能够快速、高效地进行数据查找和管理，适合用于表示需要关联数据的场景。字典在数据分析和应用开发中非常重要，特别是当数据需要通过唯一的键进行快速查找时。例如，在国际贸易数据分析中，字典能够高效存储和检索不同国家的贸易统计信息，从而支持大规模数据集的分析和处理。这种特性使得字典成为处理复杂数据集和实现快速查询的理想选择。

在 Python 中，可以通过两种主要方法创建字典：使用花括号 `{}` 或使用 `dict()` 函数。这两种方法都能够方便地定义键值对，并将数据以字典形式存储。

1. **使用花括号 `{}`**

这是创建字典最常用的方法，将键值对以 `key: value` 的形式放入花括号中，键和值之间用冒号分隔，多个键值对之间用逗号分隔。例如：

```
student_info = {
    'name': 'John Doe',
    'age': 20,
    'grade': 'A'
}
print(student_info)
```

在上述代码中，`student_info` 字典包含了三个键值对，分别表示学生的姓名、年龄和成绩。这种方法简单直观，非常适合在明确知道键值对时使用。

2. **使用 `dict()` 函数** `dict()` 函数也可以用来创建字典，尤其适合从其他数据结构（如列表或元组）转换为字典。例如：

```
employee_data = dict([('name', 'Alice'), ('position', 'Manager')])
print(employee_data)
```

在这个示例中，`dict()` 函数接收一个包含键值对的列表并返回一个字典。这种方法可以灵活地从其他结构构建字典。

除了创建包含初始数据的字典，还可以创建一个空字典并逐步添加键值对：

```
inventory = {}
inventory['apples'] = 50
inventory['bananas'] = 30
print(inventory)
```

上述代码展示了如何从空字典开始，使用键来动态添加新的键值对。

1. **键的唯一性**：字典中的键必须是唯一的，如果在定义时有重复的键，后面的值会覆盖前面的值。2. **键的类型**：字典的键必须是不可变类型，例如字符串、整数或元组。

案例：使用字典存储国际贸易数据

在国际贸易数据分析的背景下，Python 字典可以用来组织和管理复杂的贸易数据，例如每个国家的出口和进口信息。以下示例展示了如何创建和应用字典来存储和处理国际贸易数据。

```
# 创建一个字典，记录每个国家的出口商品及其金额
trade_data = {
    'China': {'electronics': 500, 'textiles': 300, 'furniture': 150},
    'Germany': {'cars': 400, 'chemicals': 250, 'machinery': 300},
    'Brazil': {'soybeans': 350, 'coffee': 200, 'iron_ore': 150}
}

# 输出中国的电子产品出口额
print(f"China's electronics export value: {trade_data['China']['electronics']} billion USD")

# 添加一个新的国家及其出口数据
trade_data['India'] = {'pharmaceuticals': 250, 'spices': 100, 'textiles': 200}

# 更新德国的汽车出口额
trade_data['Germany']['cars'] = 450

# 删除巴西的铁矿石出口数据
del trade_data['Brazil']['iron_ore']

# 打印更新后的贸易数据
print(trade_data)
```

代码解析：

1. **字典的嵌套结构：**`trade_data` 字典包含每个国家作为键，每个国家的值又是一个嵌套字典，记录该国的商品及其出口金额。这种嵌套结构非常适合管理和检索多层次的贸易数据。

2. **访问和修改字典中的值：**可以通过键访问特定国家及其商品的数值。例如，`trade_data['China']['electronics']` 直接检索中国的电子产品出口数据。此外，通过给定键值可以轻松添加、更新或删除数据，这种灵活性在数据分析中非常有用。

3. **动态更新数据：**可以根据分析需求，动态地添加新的国家或更新现有数据，从而保持数据的准确性和实时性。

使用字典可以快速处理和分析大规模的贸易数据，例如计算某一类商品的总出口额、跟踪特定国家的贸易变化或实现基于条件的筛选和计算。这种方式为国际贸易数据的组织和管理提供了高效和结构化的解决方案。

7.2 字典的基本操作

重要性：★★★★★； **难易度：**★★

7.2.1 查找元素

查找字典元素的基本操作主要有两种方式：使用方括号（`[]`）和 `get()` 方法。以下是详细的介绍及代码示例。

1. 使用方括号（`[]`）访问字典元素

通过在方括号中指定键，可以直接获取字典中与该键对应的值。这种方式要求键在字典中存在，否则会抛出 `KeyError` 异常。示例如下：

```
# 定义一个字典
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}

# 使用方括号访问字典元素
print(my_dict['name'])  # 输出：Alice
print(my_dict['age'])   # 输出：25
```

这种方法适用于已知键一定存在的情况，因为它可以直接返回值且语法简洁高效。

2. 使用 `get()` 方法

`get()` 方法是一种更安全的访问字典元素的方式，它不仅可以返回键对应的值，还允许指定一个默认值，以防键不存在时避免异常：

```
# 使用 get() 方法访问字典元素
value = my_dict.get('name')
print(value)  # 输出：Alice

# 访问不存在的键
value = my_dict.get('country', 'Not Found')
print(value)  # 输出：Not Found
```

`get()` 方法的优势在于，当键不存在时，可以返回一个自定义的默认值（默认为 `None`），从而避免了异常的产生。这在处理大型或不确定结构的字典时尤为实用。

3. 使用 `setdefault()` 方法

`setdefault()` 方法用于在字典中查找指定键的值。如果键存在，则返回对应的值；如果键不存在，则将该键与提供的默认值一起插入字典中，并返回这个默认值。如果未指定默认值，方法会插入键并将其值设为 `None`。

示例 1：键已存在的情况

```
person = {'name': 'Alice', 'age': 25}

# 键 'age' 存在于字典中
age_value = person.setdefault('age')
print(age_value)  # 输出：25
```

在这个示例中，`setdefault()` 方法返回字典中已存在的 `age` 键的值。

示例 2：键不存在的情况

```
person = {'name': 'Alice'}

```

```
# 键 'salary' 不存在，因此插入该键，值为默认的 None
salary_value = person.setdefault('salary')
print(person)  # 输出：{'name': 'Alice', 'salary': None}

# 插入一个新键 'age'，值为 30
age_value = person.setdefault('age', 30)
print(person)  # 输出：{'name': 'Alice', 'salary': None, 'age': 30}
```

在这个例子中，当键不存在时，`setdefault()` 方法将插入该键及其默认值。如果提供了默认值，则该值将被插入字典并返回；否则，返回 `None`。

`setdefault()` 方法是一种在修改或初始化字典值时的便捷方式，尤其适用于避免重复检查键是否存在的情形。

4. 选择合适的方法

- 若键在字典中是必然存在的，直接使用方括号可以提高代码效率。

- 若键的存在与否不确定，且希望有默认返回值或避免异常处理，`get()` 方法和 `setdefault()` 是更好的选择。

这三种方式各有优劣，根据具体情况选择适当的方法，可以提高代码的健壮性和可读性。

案例:基于字典的季度财务数据查找

在财务数据分析中,Python 字典可以用于快速查找和管理财务数据。例如,可以将财务报表中的信息(如收入、支出或各项资产)存储为字典,并根据特定的键(如科目名称或日期)进行查找。这种方法高效且清晰,尤其适用于需要快速访问特定财务数据的情境。

假设有一个字典存储了某公司的季度财务数据,其中包括每个季度的收入数据。可以通过字典的键直接访问特定季度的收入值,代码如下:

```
# 定义一个字典存储季度收入数据
quarterly_revenue = {
    'Q1_2023': 500000,
    'Q2_2023': 620000,
    'Q3_2023': 580000,
    'Q4_2023': 630000
}

# 查找特定季度的收入
q2_revenue = quarterly_revenue['Q2_2023']

# 打印结果
print(f"2023年第二季度的收入为: {q2_revenue}")
# 输出: 2023年第二季度的收入为: 620000
```

在上述代码中,`quarterly_revenue`字典的键表示不同的季度,如`'Q1_2023'`,值为对应季度的收入。通过`quarterly_revenue['Q2_2023']`,可以直接访问第二季度的收入数据。这种查找方式避免了在列表或其他结构中通过索引定位元素的不便,同时提高了代码的可读性和效率。

这种方法在实际财务数据分析中非常有用,例如当需要快速查询某个时间段的特定财务指标(如现金流、净利润等)时,可以通过类似的字典结构实现快速访问和分析。

7.2.2 增加元素

增加字典元素的基本语法主要有两种方式:直接赋值和使用`update()`方法。这两种方法适用于不同的场景,根据需要选择最为合适的方法。

1. 直接赋值法

直接赋值是最常见的方式。通过为字典中的新键赋值,即可添加新的键值对。如果该键已经存在,则会更新对应的值。以下是示例代码:

```
# 创建一个初始字典
my_dict = {"name": "Alice", "age": 25}

# 增加新键值对
my_dict["city"] = "New York"

print(my_dict)
# 输出: {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

在这个例子中，通过将 `"city"` 键赋值为 `"New York"`，成功将新元素添加到字典中。这种方法简单直观，适用于需要快速添加单个键值对的情况。

2. 使用 `update()` 方法

`update()` 方法可以一次添加或更新多个键值对。该方法接受一个字典或其他键值对的可迭代对象作为参数，将其内容添加到原字典中。例如：

```
# 使用 update() 方法添加元素
my_dict.update({"country": "USA", "occupation": "Engineer"})

print(my_dict)
# 输出：{'name': 'Alice', 'age': 25, 'city': 'New York', 'country': 'USA', 'occupation': 'Engineer'}
```

`update()` 方法的优点在于可以批量添加多个键值对，提高代码的可读性和效率。此外，如果传入的键已存在，方法会更新该键的值；如果键不存在，则会新增该键值对。

3. 两种方法的对比

- **直接赋值**更适合添加或更新单个键值对，语法简洁明了。

- **`update()` 方法**则适合批量操作，特别是在需要合并两个字典或一次性添加多个键值对时更加高效。

案例:更新电商平台库存信息字典

在电子商务数据分析中,Python 字典(dictionary)是非常有用的数据结构,可以存储商品信息、客户订单数据、库存状态等多种数据。字典以键值对的形式存储数据,便于快速查找和更新,因此在处理动态的数据(如实时的订单或库存变动)时极为高效。

假设需要在电子商务平台上跟踪不同商品的库存信息,初始的字典存储了部分商品的库存数量。现在需要添加新的商品及其库存信息,具体可以使用以下两种方法:通过 方括号语法 直接添加键值对,或者使用 `update()` 方法进行批量更新。

```
# 初始库存字典
inventory = {
    'item_001': {'name': 'Laptop', 'quantity': 50},
    'item_002': {'name': 'Smartphone', 'quantity': 150},
}

# 使用方括号语法添加新商品
inventory['item_003'] = {'name': 'Tablet', 'quantity': 30}

# 打印更新后的字典
print(inventory)
# 输出:
# {
    #     'item_001': {'name': 'Laptop', 'quantity': 50},
    #     'item_002': {'name': 'Smartphone', 'quantity': 150},
    #     'item_003': {'name': 'Tablet', 'quantity': 30}
    # }

# 另一种方法:使用 update() 方法批量添加商品
inventory.update({
    'item_004': {'name': 'Headphones', 'quantity': 75},
    'item_005': {'name': 'Monitor', 'quantity': 20}
})

# 打印更新后的字典
print(inventory)
# 输出:
# {
    #     'item_001': {'name': 'Laptop', 'quantity': 50},
    #     'item_002': {'name': 'Smartphone', 'quantity': 150},
    #     'item_003': {'name': 'Tablet', 'quantity': 30},
    #     'item_004': {'name': 'Headphones', 'quantity': 75},
    #     'item_005': {'name': 'Monitor', 'quantity': 20}
    # }
```

在上述代码中,`inventory` 字典用于记录商品 ID 及其对应的商品信息。首先,通过 `inventory['item_003'] = {'name': 'Tablet', 'quantity': 30}` 这种方括号的语法方式,向字典中添加了一个新的商品信息。如果该键已存在,则会更新其值。在此基础上,又通过 `update()` 方法一次性添加了多个商品,这种方法尤其适合需要同时添加多个数据的场景。

7.2.3 删除元素

删除字典元素的基本操作有几种常用的方法，包括 `del` 关键字、`pop()` 方法、`popitem()` 方法以及 `clear()` 方法。这些方法各有特点，适用于不同的场景。

1. 使用 `del` 关键字

`del` 关键字可以直接删除字典中的特定键值对。当指定的键存在时，这个操作会移除该键及其对应的值。如果键不存在，则会抛出 `KeyError` 异常。示例如下：

```
# 创建一个字典
my_dict = {'a': 1, 'b': 2, 'c': 3}

# 删除键 'b'
del my_dict['b']

print(my_dict)  # 输出：{'a': 1, 'c': 3}
```

这种方法适用于当确信要删除的键在字典中存在时使用，语法简洁高效。

2. 使用 `pop()` 方法

`pop()` 方法可以删除指定键的键值对，并返回被删除的值。与 `del` 不同的是，`pop()` 允许设置一个默认值，如果键不存在，则返回该默认值而不是抛出异常：

```
# 使用 pop() 删除键 'a'
value = my_dict.pop('a', None)  # 若键不存在，则返回 None

print(value)     # 输出：1
print(my_dict)   # 输出：{'c': 3}
```

`pop()` 方法在希望获取被删除元素的值或需要设置默认值时非常有用。

3. 使用 `popitem()` 方法

`popitem()` 用于删除字典中的最后一个键值对（LIFO 顺序，即“后进先出”）。此方法会返回被删除的键值对作为一个元组：

```
# 创建一个字典
my_dict = {'x': 10, 'y': 20}

# 删除并返回最后一个键值对
key, value = my_dict.popitem()

print(key, value)  # 输出：y 20
print(my_dict)     # 输出：{'x': 10}
```

如果字典为空，`popitem()` 会抛出 `KeyError` 异常。此方法适合在处理最近添加的元素时使用。

4. 使用 `clear()` 方法

`clear()` 方法会清空字典中的所有元素，使其变为空字典：

```
# 清空字典
my_dict.clear()

```

```
print(my_dict)  # 输出：{}
```

`clear()` 适用于需要一次性移除所有字典元素的情况。

- **`del`**：适合直接删除特定键值对，但在键不存在时需要特别注意异常处理。
- **`pop()`**：推荐在需要返回被删除值或希望避免异常时使用。
- **`popitem()`**：用于删除最后一个键值对，适合处理最新添加的元素。
- **`clear()`**：用于清空整个字典。

案例：基于字典的国际贸易数据清理

在国际贸易数据分析中，Python 字典可以用于存储和管理贸易数据，如商品类别、出口数量、进口金额等。当需要清理或更新数据时，可以使用字典的删除操作来移除不再需要的元素。例如，某贸易数据包含国家与其出口总值的映射，可以通过删除特定国家的条目来更新数据集。以下是一个基于国际贸易数据的代码示例：

```
# 定义一个字典，存储国家及其出口金额（单位：百万美元）
trade_data = {
    'China': 3000,
    'USA': 2500,
    'Germany': 1800,
    'Japan': 1500,
    'India': 1100
}

# 删除某个国家的出口数据，例如删除 'Germany'
del trade_data['Germany']

# 打印更新后的字典
print(trade_data)
# 输出：
# {'China': 3000, 'USA': 2500, 'Japan': 1500, 'India': 1100}
```

在上述代码中，`trade_data` 字典中存储了多个国家的出口金额。通过 `del` 语句删除 `'Germany'` 的记录，从而更新了字典内容。删除操作适用于清理不再需要或无效的数据，使得分析更加精确和高效。

7.2.4 修改元素

修改字典元素的基本语法主要有两种方法：使用方括号（`[]`）直接赋值和 `update()` 方法。这两种方法灵活性强，适用于不同场景。

1. 使用方括号（`[]`）直接赋值

要修改字典中的特定元素，可以直接通过方括号引用该键并赋予新的值。如果该键存在，操作会更新值；如果不存在，则会添加一个新的键值对。以下是示例：

```
# 创建一个字典
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}

# 修改现有键的值
my_dict['age'] = 30

# 新增一个键值对
my_dict['country'] = 'USA'

print(my_dict)
# 输出：{'name': 'Alice', 'age': 30, 'city': 'New York', 'country': 'USA'}
```

这种方法直接且高效，适合修改单个键的值或添加新键值对的操作。

2. 使用 `update()` 方法

`update()` 方法可以同时更新或添加多个键值对。它接受一个字典作为参数，并将其中的键值对合并到目标字典中。这种方法尤其适合在一次操作中需要修改或添加多个键值对的情况：

```
# 使用 update() 方法更新字典
my_dict.update({'age': 35, 'city': 'Los Angeles'})

print(my_dict)
# 输出：{'name': 'Alice', 'age': 35, 'city': 'Los Angeles', 'country': 'USA'}
```

在这个例子中，`update()` 方法不仅更新了已有键（如 `age` 和 `city`），还可以添加新的键值对。该方法的优点在于灵活性高，可以高效地进行批量修改。

- **方括号直接赋值**适合修改单个键的值或简单地新增一个键值对，语法简洁。

- **`update()` 方法**更适合批量修改或合并字典，在需要处理多个键值对时效率更高。

案例:基于字典的销售数据更新

在销售数据分析中,Python 字典非常适合存储和修改产品信息及其销售数据。例如,当一个企业的销售数据需要更新或调整时,可以直接通过修改字典元素来实现。下面是一个基于销售数据的代码示例,用于演示如何修改字典中的元素。以下代码示例展示了如何更新特定产品的销售额:

```
# 定义一个字典存储产品及其销售额(单位:美元)
sales_data = {
    'Product_A': 15000,
    'Product_B': 23000,
    'Product_C': 18000,
    'Product_D': 21000
}

# 更新 'Product_B' 的销售额
sales_data['Product_B'] = 25000

# 打印更新后的字典
print(sales_data)
# 输出:
# {'Product_A': 15000, 'Product_B': 25000, 'Product_C': 18000, 'Product_D': 21000}
```

在此代码中,`sales_data` 字典存储了多种产品及其对应的销售额。通过 `sales_data['Product_B'] = 25000` 语句,可以直接更新 `'Product_B'` 的销售数据。这种操作非常高效,适合于实时更新销售数据或根据新的市场情况进行调整。

7.2.5 复制字典

复制字典有多种方法,最常用的包括 `copy()` 方法、`=` 操作符以及 `deepcopy()` 函数。以下是这些方法的详细介绍及代码示例。

1. 使用 `copy()` 方法

`copy()` 方法是 Python 字典对象的一个内置方法,用于创建字典的浅拷贝。浅拷贝仅复制字典本身及其键值对,但如果字典中包含可变对象(例如列表或其他字典),这些对象在新旧字典中依然共享引用。示例如下:

```
1 # 创建一个字典
2 original_dict = {'a': 1, 'b': 2, 'c': [1, 2, 3]}
3
4 # 使用 copy() 方法进行浅拷贝
5 copied_dict = original_dict.copy()
6
7 print(copied_dict)  # 输出: {'a': 1, 'b': 2, 'c': [1, 2, 3]}
```

在此例中,`copy()` 方法成功创建了一个浅拷贝。在修改浅拷贝中不可变类型(如整数和字符串)的值时,原始字典不会受到影响;但如果修改可变对象(如列表)的内容,原始字典和浅拷贝中的内容都会同时更新。

2. 使用 `=` 操作符

简单地将一个字典赋值给另一个变量并不会创建真正的副本，而是生成一个对原始字典的引用。修改其中任何一个字典都会影响到另一个：

```
# 使用 = 操作符复制字典
dict_a = {'name': 'Alice', 'age': 25}
dict_b = dict_a

dict_b['age'] = 30

print(dict_a)  # 输出：{'name': 'Alice', 'age': 30}
print(dict_b)  # 输出：{'name': 'Alice', 'age': 30}
```

在这个例子中，`dict_b` 和 `dict_a` 共享相同的内存地址，修改其中之一会直接影响另一个。因此，`=` 操作符并不适合在需要独立副本的情况下使用。

3. 使用 `deepcopy()` 函数

若需要复制字典及其所有嵌套对象（即实现深拷贝），可以使用 `copy` 模块中的 `deepcopy()` 函数。此方法会递归复制所有嵌套对象，使得新字典完全独立于原字典：

```
from copy import deepcopy

# 创建一个包含可变对象的字典
original_dict = {'a': 1, 'b': [2, 3, 4]}

# 使用 deepcopy() 方法进行深拷贝
deep_copied_dict = deepcopy(original_dict)

# 修改深拷贝中的值
deep_copied_dict['b'].append(5)

print(original_dict)       # 输出：{'a': 1, 'b': [2, 3, 4]}
print(deep_copied_dict)    # 输出：{'a': 1, 'b': [2, 3, 4, 5]}
```

在这个例子中，修改深拷贝中的列表并不会影响到原始字典中的列表内容，因此 `deepcopy()` 适用于需要完全独立副本的场景。

7.2.6 其他常见操作

除了基本的增删改查操作外，字典还提供了一些其他常见且有用的操作，例如获取字典长度、判断成员以及其他迭代方法。以下是这些操作的详细介绍及代码示例。

1. 获取字典长度：`len()`

Python 内置函数 `len()` 可以返回字典中键值对的数量。这个函数适用于所有可迭代对象，因此在计算字典的长度时非常方便：

```
# 创建一个字典
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}

# 获取字典的长度
length = len(my_dict)
```

```

print(length)  # 输出：3
```

上述代码中，`len(my_dict)` 返回字典中键值对的数量。

2. 成员判断：`in` 和 `not in`

Python 允许使用 `in` 关键字检查字典中是否存在某个键。`in` 返回 `True` 或 `False`，具体取决于键是否存在。示例如下：

```
# 判断键是否存在于字典中
if 'name' in my_dict:
    print("键 'name' 存在于字典中")

if 'email' not in my_dict:
    print("键 'email' 不存在于字典中")
```

这种操作对于验证某个键是否存在非常有用，避免在访问不存在的键时引发 `KeyError` 异常。

3. 获取所有键、值或键值对：`keys()`、`values()` 和 `items()`

- **`keys()`** 方法返回字典中所有键的视图对象，可以用于迭代或转换为列表：

```
# 获取所有键
keys = my_dict.keys()
print(list(keys))  # 输出：['name', 'age', 'city']
```

- **`values()`** 方法返回所有值的视图对象，也可以迭代或转换为列表：

```
# 获取所有值
values = my_dict.values()
print(list(values))  # 输出：['Alice', 25, 'New York']
```

- **`items()`** 方法返回键值对的视图对象，每个元素为一个包含键和值的元组：

```
# 获取所有键值对
items = my_dict.items()
for key, value in items:
    print(f"{key}: {value}")
# 输出：
# name: Alice
# age: 25
# city: New York
```

这些方法使得在不改变字典结构的情况下，可以方便地访问字典的各个组成部分。

这些操作方法不仅增强了字典的灵活性，还提供了更高效的数据处理方式。无论是在迭代、成员判断还是获取字典长度时，这些操作都能显著提升代码的简洁性和可读性。

案例：基于字典视图的销售数据遍历

在商业数据分析中，Python 字典常用于存储商品销售数据、客户信息等结构化数据。通过字典视图（`items()`、`keys()`、`values()`方法），可以方便地遍历和访问字典的元素，从而实现对数据的快速分析和处理。以下代码示例展示了如何使用字典视图方法来遍历和获取数据：

```
# 定义一个字典，存储产品及其销售额（单位：美元）
sales_data = {
    'Product_A': 15000,
    'Product_B': 23000,
    'Product_C': 18000,
    'Product_D': 21000
}

# 使用 items() 方法遍历字典的键值对
for product, revenue in sales_data.items():
    print(f"{product}: {revenue}美元")

# 使用 keys() 方法获取所有产品名称
product_names = list(sales_data.keys())
print("产品列表:", product_names)

# 使用 values() 方法获取所有销售额
revenues = list(sales_data.values())
print("销售额列表:", revenues)
```

在此代码中，`items()`方法用于同时遍历字典中的键和值，使得每次迭代都能访问到一个产品及其对应的销售额。`keys()`方法则返回一个包含所有键（产品名称）的视图对象，便于将其转换为列表形式。而`values()`方法返回所有值（销售额），也可以转换为列表，便于进一步的统计分析或计算。这些视图对象在数据更新时会动态变化，非常适合在商业数据分析中的实时数据处理场景。

7.3 字典的字符串格式化使用

重要性：★★； 难易度：★★

`format_map()`方法用于将字典中的值替换到字符串中定义的占位符。这一方法与`format()`方法类似，但`format_map()`直接接受一个映射（通常为字典）作为参数，而不使用解包操作符`**`。这种方法在处理字典子类或需要动态填充字符串的场景中非常有用。

`format_map()`的基本语法为：

```
string.format_map(mapping)
```

其中`mapping`是包含键值对的字典。这些键与字符串中定义的占位符相匹配，然后将相应的值填充到字符串中。例如：

```
# 定义字典
data = {'name': 'Alice', 'age': 30}

```

```
# 使用 format_map() 方法进行格式化
print('My name is {name} and I am {age} years old'.format_map(data))

# 使用 format() 方法和解包操作符**进行格式化
print('My name is {name} and I am {age} years old'.format(**data))
```

在上述代码中，字典 `data` 中的键 `'name'` 和 `'age'` 与字符串中的占位符匹配，成功替换为相应的值。

如果字典中缺少某个占位符对应的键，`format_map()` 将抛出 `KeyError`。

案例：基于 `format_map()` 的财务报表生成

在财务数据分析中，Python 的 `format_map()` 方法可以结合字典实现动态字符串的格式化输出，非常适用于财务报表和数据呈现场景。例如，可以将财务数据（如收入、利润等）存储在字典中，通过 `format_map()` 方法格式化输出为完整的报表。以下是一个基于财务数据的代码示例，用于展示如何使用 `format_map()` 方法：

```
# 定义一个字典存储财务数据
financial_data = {
    'company': 'TechCorp',
    'revenue': 5000000,
    'profit': 1200000
}

# 使用 format_map() 方法格式化字符串
report = "Company: {company}\nRevenue: ${revenue}\nProfit: ${profit}".format_map(financial_data)

# 打印财务报表
print(report)
# 输出：
# Company: TechCorp
# Revenue: $5000000
# Profit: $1200000
```

在该代码中，`financial_data` 字典包含了公司名称、收入和利润的数据。通过 `format_map()` 方法，可以直接将字典中的值插入到字符串模板中，实现动态内容替换。这种方式特别适合生成财务报表或其他商业报告时进行自动化输出，保证内容的准确性和一致性。