

集合在商业数据分析中具有重要作用, 因为其去重和高效的成员测试功能能够有效清理数据集, 确保数据的一致性和准确性。此外, 集合运算 (如并集、交集) 在处理大规模数据时, 能够快速合并、比较不同数据集, 从而优化数据处理流程。

6.1 创建集合

重要性:★★★★; 难易度:★

集合 (Set) 是 Python 中的一种内置数据类型, 用于存储多个元素的无序集合。集合中的元素是唯一的, 即不存在重复值, 这使其非常适合执行诸如去重、成员测试等操作。此外, 集合支持多种集合运算, 如并集、交集、差集和对称差集。Python 集合是可变的, 即可以动态添加或移除元素, 但集合中的每个元素必须是不可变的类型 (如整数、字符串或元组)。

创建集合有两种主要方法: 使用花括号 `{}` 直接定义集合, 或使用内置的 `set()` 函数。这两种方法均可用于构建一个包含唯一元素的集合。

1. 使用花括号定义集合

这是最直接的方式, 通过将元素放在花括号内并用逗号分隔即可。例如:

```
1 fruits = {"apple", "banana", "cherry"}
2 print(fruits) # 输出: {'apple', 'banana', 'cherry'}
```

在此示例中, 集合 `fruits` 包含三个字符串元素。值得注意的是, 集合不允许重复元素, 因此如果在定义时加入重复值, 它们将被自动去除。

2. 使用 `set()` 函数创建集合

这种方法尤其适合需要从其他可迭代对象 (如列表或元组) 转换为集合的情况。可以将一个可迭代对象作为参数传递给 `set()` 函数来创建集合:

```
1 numbers = set([1, 2, 3, 3, 4])
2 print(numbers) # 输出: {1, 2, 3, 4}
```

这里,列表中的重复值 3 被移除,最终得到一个只包含唯一元素的集合。

需要注意的是,使用花括号创建空集合并不可行,因为 `{}` 默认创建一个空字典。若要创建一个空集合,必须使用 `set()` 函数,如下所示:

```
1 empty_set = set()
2 print(empty_set) # 输出: set()
```

这种方法确保空集合的正确创建。

6.2 集合的基本操作

重要性:★★★★; 难度:★

集合的基本操作包括并集、交集、差集和对称差集,这些操作可以通过运算符或方法来实现。以下将结合代码示例介绍这些操作的基本语法:

1. 并集 (Union)

并集操作返回两个集合中所有不重复的元素。可以使用 `|` 运算符或 `union()` 方法:

```
1 set1 = {1, 2, 3}
2 set2 = {3, 4, 5}
3 print(set1 | set2) # 输出: {1, 2, 3, 4, 5}
4 print(set1.union(set2)) # 输出: {1, 2, 3, 4, 5}
```

运算符 `|` 和方法 `union()` 功能相同,但 `union()` 方法可以接受其他可迭代对象(如列表)。

2. 交集 (Intersection)

交集操作返回两个集合中的公共元素。可以使用 `&` 运算符或 `intersection()` 方法:

```
1 set1 = {1, 2, 3, 4}
2 set2 = {3, 4, 5, 6}
3 print(set1 & set2) # 输出: {3, 4}
4 print(set1.intersection(set2)) # 输出: {3, 4}
```

这两种方法均支持同时对多个集合进行操作。

3. 差集 (Difference)

差集操作返回一个集合中存在但另一个集合中不存在的元素。可以使用 `-` 运算符或 `difference()` 方法:

```
1 set1 = {1, 2, 3, 4}
2 set2 = {3, 4, 5}
3 print(set1 - set2) # 输出: {1, 2}
4 print(set1.difference(set2)) # 输出: {1, 2}
```

需要注意的是,差集运算的结果依赖于集合的顺序。

4. 对称差集 (Symmetric Difference)

对称差集操作返回两个集合中不重复的元素。可以使用 `^` 运算符或 `symmetric_difference()` 方法:

```
1 set1 = {1, 2, 3, 4}
2 set2 = {3, 4, 5, 6}
3 print(set1 ^ set2) # 输出: {1, 2, 5, 6}
4 print(set1.symmetric_difference(set2)) # 输出: {1, 2, 5, 6}
```

该操作适用于查找在两个集合中互不相同的元素。

这些集合操作为数据分析和大数据管理中的数据去重、交叉比对及合并操作提供了高效工具。

案例:电商客户购买行为分析

在电子商务数据分析的背景下,Python 集合操作可用于分析客户购买行为,例如识别购买多种产品的客户或确定购买特定商品组合的客户。

```
# 初始化集合, 分别存储购买鞋子、腰带和衬衫的客户ID
shoes = {"1001", "1002", "1005", "1007"}
belts = {"1002", "1004", "1006", "1008"}
shirts = {"1002", "1003", "1007", "1009"}

# 分析不同集合的交集、并集和差集
print(f"购买了鞋子的客户数量: {len(shoes)}")
print(f"购买了腰带的客户数量: {len(belts)}")
print(f"同时购买了鞋子和腰带的客户数量: {len(shoes & belts)}")
print(f"购买了鞋子但未购买腰带的客户数量: {len(shoes - belts)}")
print(f"购买了鞋子、腰带和衬衫的客户数量: {len(shoes & belts & shirts)}")

# 输出同时购买所有三种商品的客户
print("以下客户购买了鞋子、腰带和衬衫:")
for customer in shoes & belts & shirts:
    print(customer)
```

代码解析

1. **集合初始化:** 定义了三个集合, `shoes`、`belts` 和 `shirts`, 分别表示购买特定商品的客户 ID 集合。

2. **集合操作:**

- 交集操作 (`&`) 用于找出购买了多种商品的客户, 例如同时购买鞋子和腰带的客户。

- 差集操作 (`-`) 用于识别购买了一种商品但未购买另一种商品的客户。

- 多集合交集 (`shoes & belts & shirts`) 可用于确定购买所有三种商品的客户。

此类集合操作在电子商务数据分析中非常有用, 例如用于识别忠实客户群体, 分析购买模式, 以及优化营销策略。通过应用这些操作, 可以快速定位购买特定组合商品的客户群体, 从而提供定制化服务或精准营销。

6.3 集合常用方法

重要性:★★★★; 难度度:★

集合提供了多种内置方法来操作和管理数据, 这些方法在数据分析、数据清洗和集合运算等场景中非

常有用。以下是几个常用的集合方法及其基本语法和应用示例：

1. add()

`add()` 方法将一个元素添加到集合中,如果元素已经存在,则不会有任何变化。例如:

```
1 my_set = {1, 2, 3}
2 my_set.add(4)
3 print(my_set) # 输出: {1, 2, 3, 4}
```

此方法适合在动态数据处理中逐个添加元素。

2. remove() 和 discard()

`remove()` 和 `discard()` 用于从集合中删除指定元素。`remove()` 在元素不存在时会引发 `KeyError`, 而 `discard()` 不会引发错误。例如:

```
1 my_set = {1, 2, 3}
2 my_set.remove(2) # 移除元素2
3 print(my_set) # 输出: {1, 3}
```

使用 `discard()` 更为安全,因为可以避免潜在的错误。

3. issubset() 和 issuperset()

`issubset()` 和 `issuperset()` 分别用于检查一个集合是否为另一个集合的子集或超集:

```
1 set1 = {1, 2}
2 set2 = {1, 2, 3}
3 print(set1.issubset(set2)) # 输出: True
4 print(set2.issuperset(set1)) # 输出: True
```

在数据分析中,这些方法可以帮助快速验证集合之间的包含关系。

4. clear()

`clear()` 方法用于清空集合中的所有元素:

```
1 my_set = {1, 2, 3}
2 my_set.clear()
3 print(my_set) # 输出: set()
```

该方法在需要重置集合时非常有用。

6.4 frozenset 与 set 的区别

重要性:★★★★; 难易度:★★

`set` 和 `frozenset` 都是用来存储无序、唯一元素的集合数据类型,但它们有一个关键区别:`set` 是可变的,而 `frozenset` 是不可变的。这使得它们在不同场景中具有不同的用途。以下是两者的具体区别和应用示例:

1. 可变性

set: 是可变的集合类型,支持多种修改操作。例如,可以使用 `add()` 方法添加元素,或者使用 `remove()` 方法删除元素:

```
1 my_set = {1, 2, 3}
2 my_set.add(4)
3 print(my_set) # 输出: {1, 2, 3, 4}
4 my_set.remove(2)
5 print(my_set) # 输出: {1, 3, 4}
```

frozenset : 是 **set** 的不可变版本, 创建后无法修改。任何试图使用诸如 **add()** 或 **remove()** 的操作都会导致 **AttributeError** 错误:

```
1 my_frozenset = frozenset([1, 2, 3])
2 my_frozenset.add(4) # 这行会引发错误
```

因此, **frozenset** 适合用在不希望集合内容被修改的场景中, 如作为字典的键或其他集合的元素。

2. 用途和应用场景

set : 适合在需要动态修改集合内容的场景下使用, 比如数据清理和过滤。当数据集合需要频繁更新或元素需要动态增删时, **set** 提供了灵活性。

frozenset : 由于不可变性, 它是哈希的, 这使得它可以作为字典的键或其他集合的元素。例如, 当需要一个稳定且不可更改的键来表示状态或标识某一组数据时, **frozenset** 是一个理想选择。

以下示例展示了如何创建和使用 **frozenset** 与 **set** :

```
1 # 使用set
2 mutable_set = {1, 2, 3}
3 mutable_set.add(4)
4 print(mutable_set) # 输出: {1, 2, 3, 4}
5
6 # 使用frozenset
7 immutable_frozenset = frozenset([1, 2, 3])
8 print(immutable_frozenset) # 输出: frozenset({1, 2, 3})
9 # 试图修改frozenset将会引发错误
10 immutable_frozenset.add(4) # AttributeError: 'frozenset' object has no attribute 'add'
```

set 和 **frozenset** 在数据管理和分析中都非常有用, 前者适合动态操作集合的场景, 而后者则在需要集合内容固定的场景中表现出色。

案例:利用 frozenset 定义不变的投资组合

在财务数据分析中, `frozenset` 可以用于存储不可变的金融数据组合, 确保数据在分析过程中不被意外修改。以下代码示例展示了如何在财务背景下使用 `frozenset` 来记录和操作不变的投资组合数据:

```
# 定义不同客户的投资组合, 每个组合为一个frozenset
portfolio1 = frozenset(['AAPL', 'MSFT', 'GOOGL'])
portfolio2 = frozenset(['AMZN', 'TSLA', 'NFLX'])
portfolio3 = frozenset(['AAPL', 'TSLA', 'NVDA'])

# 将投资组合存储为字典的键, 以映射组合到其持有的风险等级
portfolio_risk = {
    portfolio1: '低风险',
    portfolio2: '中风险',
    portfolio3: '高风险'
}

# 查询一个特定组合的风险等级
query_portfolio = frozenset(['TSLA', 'AAPL', 'NVDA'])
print(f"查询投资组合的风险等级: {portfolio_risk.get(query_portfolio)}")
```

代码解析

- 创建 `frozenset`**: 每个 `frozenset` 代表一个不可变的投资组合, 包含特定股票代码。在金融分析中, 这种数据结构确保了投资组合在存储和分析过程中不会被修改。
- 使用 `frozenset` 作为字典键**: 由于 `frozenset` 是不可变和可哈希的, 可以将它们作为字典的键。此特性允许根据投资组合映射不同的属性 (如风险等级)。
- 查询组合**: 通过创建一个相同内容的 `frozenset`, 可以快速查询组合的相关信息, 例如风险等级。这种方法确保组合的内容一致性, 并且能够在字典中快速检索。

在金融数据分析中, `frozenset` 非常适合用于存储需要保持不变的集合数据, 如客户的固定资产组合、特定日期的交易记录或不变的市场指数集合。由于 `frozenset` 的不可变性, 在处理多线程或分布式系统时也能确保数据的一致性和安全性。