第七章 - 字典

张建章

阿里巴巴商学院

杭州师范大学

2024-09

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。

可以通过两种主要方法创建字典：使用花括号 `{}` 或 `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。

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

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)
```

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。

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()` 方法在字典中查找指定键的值。如果键存在，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，会插入键并将其值设为 `None`。

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

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。
- `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。

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。

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。

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。

```
# 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'}
```

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。

```
# 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。

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。

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。

```
# 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'，不设置默认值
my_dict.pop('a')  # 若键不存在，则抛出 KeyError 异常
# 使用 pop() 删除键 'a'
del my_dict['a']  # 若键不存在，则抛出 KeyError 异常
# 使用 pop() 删除键 'a'，设置默认值
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避免异常时非常有用。

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。

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。

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。

```
# 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)
```

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。

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。

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。

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

在此例中，`copy()` 方法创建了一个浅拷贝。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：

```python
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。

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。

1. 获取字典长度： `len()`

```
# 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
```

2. 成员判断： `in` 和 `not in`，检查字典中是否存在某个键。

```
# 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")
```

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，增强了字典的灵活性，提升了代码的简洁性和可读性。

`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。

```
# 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`。

THE END