Python Dictionary Operations: A Comprehensive Guide

Welcome to the world of Python dictionaries! Dictionaries are incredibly versatile and powerful data structures that allow you to store data in key: value pairs. Unlike lists, which are ordered sequences, dictionaries are unordered (in Python versions before 3.7, ordered from 3.7 onwards) collections where each item has a unique key.

Let's start with a practical example: storing temperatures for different cities.

1. Creating a Dictionary

A dictionary is created by placing a comma-separated list of key: value pairs inside curly braces {}. Keys must be unique and immutable (like strings, numbers, or tuples), while values can be of any data type.

```
# Example: Storing current temperatures for various cities
city_temperatures = {
    "London": 15,
    "Paris": 18,
    "New York": 22,
    "Tokyo": 25,
    "Sydney": 12
}

print(f"Our first dictionary: {city_temperatures}")
print(f"Type of city_temperatures: {type(city_temperatures)}")

# Dictionaries can also be empty
empty_dict = {}
print(f"An empty dictionary: {empty_dict}")
```

2. Accessing Values

You can access the value associated with a key by placing the key inside square brackets [] after the dictionary name.

```
city_temperatures = {
    "London": 15,
    "Paris": 18,
    "New York": 22,
    "Tokyo": 25,
    "Sydney": 12
}

# Accessing the temperature for London
london_temp = city_temperatures["London"]
print(f"Temperature in London: {london_temp}°C")

# Accessing the temperature for Tokyo
tokyo_temp = city_temperatures["Tokyo"]
print(f"Temperature in Tokyo: {tokyo_temp}°C")

# If you try to access a non-existent key, it will raise a KeyError
# print(city_temperatures["Berlin"]) # Uncommenting this will cause a KeyError
```

Using get() for safer access

The get() method is a safer way to access values, as it returns None (or a specified default value) if the key is not found, instead of raising a KeyError.

```
city_temperatures = {
    "London": 15,
    "Paris": 18,
    "New York": 22,
    "Tokyo": 25,
    "Sydney": 12
}

# Using get() for an existing key
paris_temp = city_temperatures.get("Paris")
print(f"Temperature in Paris (using get()): {paris_temp}°C")

# Using get() for a non-existent key (returns None by default)
berlin_temp = city_temperatures.get("Berlin")
print(f"Temperature in Berlin (using get()): {berlin_temp}")

# Using get() with a default value if key is not found
rome_temp = city_temperatures.get("Rome", "N/A")
print(f"Temperature in Rome (using get() with default): {rome_temp}")
```

3. Adding and Modifying Elements

Adding new key-value pairs

To add a new item to a dictionary, you simply assign a value to a new key.

```
city_temperatures = {
    "London": 15,
    "Paris": 18,
    "New York": 22
}
print(f"Original dictionary: {city_temperatures}")

# Add a new city and its temperature
city_temperatures["Rome"] = 20
print(f"After adding Rome: {city_temperatures}")

city_temperatures["Beijing"] = 28
print(f"After adding Beijing: {city_temperatures}")
```

Modifying existing values

If you assign a value to a key that already exists, the old value associated with that key will be overwritten.

```
city_temperatures = {
    "London": 15,
    "Paris": 18,
    "New York": 22
}
print(f"Original dictionary: {city_temperatures}")

# Update the temperature for London
city_temperatures["London"] = 17
print(f"After updating London's temperature: {city_temperatures}")

# Update New York's temperature
city_temperatures["New York"] = 20
print(f"After updating New York's temperature: {city_temperatures}")
```

Using update() to add/modify multiple elements

The update() method allows you to add items from another dictionary or from an iterable of key-value pairs. If keys already exist, their values are updated; otherwise, new key-value pairs are added.

```
city_temperatures = {
    "London": 15,
    "Paris": 18,
    "New York": 22
}
print(f"Original dictionary: {city_temperatures}")

# Add new cities and update existing ones
new_data = {
    "Tokyo": 25,
    "London": 16, # This will update London's temperature
    "Berlin": 19
}
city_temperatures.update(new_data)
print(f"After update() with new_data: {city_temperatures}")

# You can also update with keyword arguments
city_temperatures.update(Rome=21, Madrid=23)
print(f"After update() with keyword arguments: {city_temperatures}")
```

4. Removing Elements

a. del statement: Deleting by key

The del statement allows you to remove a key-value pair using its key. If the key doesn't exist, it will raise a KeyError.

```
city_temperatures = {
    "London": 15,
    "Paris": 18,
    "New York": 22,
    "Tokyo": 25
}
```

```
print(f"Original dictionary: {city_temperatures}")

del city_temperatures["Paris"]
print(f"After deleting 'Paris': {city_temperatures}")

# Uncommenting this will cause a KeyError
# del city_temperatures["Berlin"]
```

b. pop(): Removing and returning a value by key

The pop(key, default_value) method removes the item with the specified key and returns its value. If the key is not found, it returns default_value if provided, otherwise it raises a KeyError.

```
city_temperatures = {
    "London": 15,
    "Paris": 18,
    "New York": 22,
    "Tokyo": 25
}
print(f"Original dictionary: {city_temperatures}")

removed_temp = city_temperatures.pop("New York")
print(f"After popping 'New York': {city_temperatures}, Removed temperature:
{removed_temp}°C")

# Pop a non-existent key with a default value
non_existent_temp = city_temperatures.pop("Berlin", "Key not found")
print(f"Attempted to pop 'Berlin': {non_existent_temp}")
print(f"Dictionary after attempted pop: {city_temperatures}")

# Uncommenting this will cause a KeyError (no default value provided)
# city_temperatures.pop("Sydney")
```

c. popitem(): Removing and returning the last inserted item

The popitem() method removes and returns an arbitrary (key, value) pair. In Python 3.7+ it removes the last inserted item. If the dictionary is empty, it raises a KeyError.

```
city_temperatures = {
    "London": 15,
    "Paris": 18,
    "New York": 22,
    "Tokyo": 25
}
print(f"Original dictionary: {city_temperatures}")

last_item = city_temperatures.popitem()
print(f"After popitem(): {city_temperatures}, Removed item: {last_item}")

last_item_again = city_temperatures.popitem()
print(f"After another popitem(): {city_temperatures}, Removed item:
{last_item_again}")

# Uncommenting this will cause a KeyError if the dictionary becomes empty
# empty_dict = {}
# empty_dict.popitem()
```

d. clear(): Removing all elements

The clear() method removes all items from a dictionary, leaving it empty.

```
city_temperatures = {
    "London": 15,
    "Paris": 18,
    "New York": 22
}
print(f"Original dictionary: {city_temperatures}")

city_temperatures.clear()
print(f"After clear(): {city_temperatures}")
```

5. Iterating Through a Dictionary

You can iterate through dictionaries in several ways:

a. Iterating through keys (default)

When you loop directly over a dictionary, you iterate through its keys.

```
city_temperatures = {
    "London": 15,
    "Paris": 18,
    "New York": 22,
    "Tokyo": 25
}

print("Iterating through keys:")
for city in city_temperatures:
    print(f"City: {city}")

# This is equivalent to using .keys()
print("\nIterating through keys using .keys():")
for city in city_temperatures.keys():
    print(f"City: {city}")
```

b. Iterating through values

Use the values() method to get a view object that displays a list of all the values in the dictionary.

```
city_temperatures = {
    "London": 15,
    "Paris": 18,
    "New York": 22,
    "Tokyo": 25
}

print("Iterating through values:")
for temp in city_temperatures.values():
    print(f"Temperature: {temp}°C")
```

c. Iterating through key-value pairs (items)

Use the items() method to get a view object that displays a list of a dictionary's key-value tuple pairs. This is often the most useful way to iterate.

```
city_temperatures = {
    "London": 15,
    "Paris": 18,
    "New York": 22,
    "Tokyo": 25
}

print("Iterating through key-value pairs:")
for city, temp in city_temperatures.items():
    print(f"The temperature in {city} is {temp}°C.")
```

6. Other Useful Dictionary Methods

a. len(): Getting the number of items

The len() function returns the number of key-value pairs (items) in a dictionary.

```
city_temperatures = {
    "London": 15,
    "Paris": 18,
    "New York": 22
}
print(f"Dictionary: {city_temperatures}")
print(f"Number of cities: {len(city_temperatures)}")

empty_dict = {}
print(f"Number of items in empty_dict: {len(empty_dict)}")
```

b. copy(): Creating a shallow copy of a dictionary

Similar to lists, direct assignment (dict_b = dict_a) creates a reference, not a copy. Use copy() or dict() to create a shallow copy.

```
original_temps = {
    "London": 15,
    "Paris": 18
}
print(f"Original dictionary: {original_temps}")

# Method 1: using copy()
```

```
copied_temps = original_temps.copy()
copied_temps["Rome"] = 20
print(f"Copied dictionary after modification: {copied_temps}")
print(f"Original dictionary (unchanged): {original_temps}")

# Method 2: using dict() constructor
another_copy = dict(original_temps)
another_copy["Berlin"] = 19
print(f"Another copy after modification: {another_copy}")
print(f"Original dictionary (still unchanged): {original_temps}")
```

c. fromkeys(): Creating a dictionary from a sequence of keys

The fromkeys() class method creates a new dictionary with keys from a sequence and values set to a specified value (defaults to None).

```
cities = ["London", "Paris", "New York"]
default_temp = 0

# Create a dictionary with default temperature
initial_temps = dict.fromkeys(cities, default_temp)
print(f"Dictionary from keys with default temp: {initial_temps}")

# Create a dictionary with None as default value
cities_no_temp = ["Tokyo", "Sydney"]
unknown_temps = dict.fromkeys(cities_no_temp)
print(f"Dictionary from keys with None values: {unknown_temps}")
```

7. Important Concepts Related to Dictionaries

- **Mutable:** Dictionaries are **mutable**, meaning you can add, remove, or modify key-value pairs after the dictionary is created.
- **Keys are Unique:** Each key in a dictionary must be unique. If you try to add an item with an existing key, the old value will be overwritten.
- **Keys are Immutable:** Dictionary keys must be of an immutable type (e.g., strings, numbers, tuples). You cannot use mutable types like lists or other dictionaries as keys.
- **Values are Heterogeneous:** Dictionary values can be of any data type and can be different from each other.

- **Ordered (Python 3.7+):** As of Python 3.7, dictionaries maintain insertion order. This means the order in which items are added is preserved when you iterate over the dictionary. In older versions (Python 3.6 and earlier), dictionaries were inherently unordered.

This comprehensive guide should give you a solid understanding of how to work with Python dictionaries. Practice these operations, and you'll find them incredibly useful for organizing and accessing data efficiently!