Introduction to Python Dictionaries

A dictionary is a highly flexible **key-value pair** data structure in Python, used to store data with mapping relationships (such as "name-age" or "student ID-score"). Unlike lists that access elements via indexes, dictionaries locate "values" through "keys", offering higher query efficiency and serving as a frequently used tool in daily development.

I. Core Features of Dictionaries

- **Unique and Immutable Keys**: Keys must be of immutable data types (e.g., strings, numbers, tuples) and cannot be duplicated in the same dictionary; values can be of any data type (including mutable types like lists and dictionaries).
- **From Unordered to Ordered**: In Python 3.7 and above, dictionaries preserve the insertion order of key-value pairs; they are unordered in Python 3.6 and below.
- **Dynamically Modifiable**: Supports adding, deleting, and modifying key-value pairs at any time without predefining the length.

II. Ways to Create a Dictionary

Dictionaries are enclosed in curly braces `{}`, with colons `:` separating keys and values, and commas `,` separating different key-value pairs. There are 4 common creation methods:

1. Direct Assignment (Most Common)

Suitable for scenarios where fixed key-value pairs are known, with a concise and intuitive syntax:

```
Plain Text
# Example: Storing student information
student = {
    "name": "Zhang San",
    "age": 20,
    "major": "Computer Science",
    "is_enrolled": True,
    "scores": [85, 92, 78]  # Value is a list type
}
print(student)  # Output: {'name': 'Zhang San', 'age': 20,
```

```
'major': 'Computer Science', 'is_enrolled': True, 'scores': [85,
92, 78]}
```

2. Creating with the dict() Function

Suitable for dynamically generating dictionaries from existing data (e.g., list of tuples):

```
Plain Text
# Method 1: Pass key-value pair parameters
fruit_price = dict(apple=5.8, banana=2.5, orange=3.2)
print(fruit_price)  # Output: {'apple': 5.8, 'banana': 2.5,
'orange': 3.2}

# Method 2: Pass an iterable containing key-value pairs (e.g.,
list of tuples)
color_code = dict([("red", "#FF0000"), ("green", "#00FF00"),
("blue", "#0000FF")])
print(color_code)  # Output: {'red': '#FF0000', 'green':
'#00FF00', 'blue': '#0000FF'}
```

3. Creating with Dictionary Comprehension

Similar to list comprehension, it quickly generates dictionaries through loops and conditional filtering, which is efficient and concise:

```
Plain Text
# Example 1: Generate a dictionary of numbers 1-5 and their
squares
square_dict = {x: x**2 for x in range(1, 6)}
print(square_dict)  # Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# Example 2: Filter squares of even numbers (add condition)
even_square_dict = {x: x**2 for x in range(1, 6) if x % 2 == 0}
print(even_square_dict)  # Output: {2: 4, 4: 16}
```

4. Creating an Empty Dictionary

Used for dynamically adding key-value pairs later, such as receiving user input and storing program running results:

```
Plain Text
empty_dict = {}  # Method 1: Empty curly braces
empty_dict2 = dict()  # Method 2: dict() function
print(type(empty_dict))  # Output: <class 'dict'>
```

III. Core Operations on Dictionaries

Dictionary operations revolve around "accessing, adding, modifying, and deleting key-value pairs". Below are examples of frequently used operations:

1. Accessing Dictionary Values

There are two core methods, and the `get()` method is recommended (to avoid errors when the key does not exist):

```
Plain Text
student = {"name": "Zhang San", "age": 20, "major": "Computer Science"}

# Method 1: Direct access via key (raises KeyError if key does not exist)
print(student["name"])  # Output: Zhang San

# Method 2: get() method (returns default value if key does not exist, default is None)
print(student.get("age"))  # Output: 20
print(student.get("gender", "Unknown"))  # Output: Unknown (key 'gender' does not exist)
```

2. Adding/Modifying Key-Value Pairs

Dictionary keys are unique. If the key exists, the value is modified; if not, a new key-value pair is added:

```
Plain Text
student = {"name": "Zhang San", "age": 20}

# Modify the value of an existing key (age exists)
student["age"] = 21
print(student)  # Output: {'name': 'Zhang San', 'age': 21}

# Add a new key-value pair (gender does not exist)
student["gender"] = "Male"
print(student)  # Output: {'name': 'Zhang San', 'age': 21, 'gender': 'Male'}
```

3. Deleting Key-Value Pairs

The `del` statement and `pop()` method are commonly used. The former deletes directly, while the latter deletes and returns the corresponding value:

```
Plain Text
student = {"name": "Zhang San", "age": 20, "gender": "Male"}

# Method 1: del statement (deletes directly, no return value)
del student["gender"]
print(student)  # Output: {'name': 'Zhang San', 'age': 20}

# Method 2: pop() method (deletes and returns the value; default
value can be specified if key does not exist)
age = student.pop("age")
print(age)  # Output: 20
print(student)  # Output: {'name': 'Zhang San'}

# Method 3: Clear all key-value pairs (retains empty dictionary)
student.clear()
print(student)  # Output: {}
```

4. Iterating Over Dictionaries

You can iterate over "keys", "values", and "key-value pairs" respectively, choosing the appropriate method based on requirements:

```
Plain Text
fruit_price = {"apple": 5.8, "banana": 2.5, "orange": 3.2}

# Iterate over all keys (keys are iterated by default)
for fruit in fruit_price:
    print(fruit)  # Output: apple, banana, orange

# Iterate over all keys (explicitly specify keys())
for fruit in fruit_price.keys():
    print(fruit)

# Iterate over all values (values() method)
for price in fruit_price.values():
    print(price)  # Output: 5.8, 2.5, 3.2

# Iterate over all key-value pairs (items() method, returns
tuples)
for fruit, price in fruit_price.items():
    print(f"{fruit} costs {price} yuan")  # Output: apple costs
5.8 yuan, etc.
```

IV. Practical Tips for Dictionaries

1. Merging Dictionaries

Python 3.9 and above support merging dictionaries using the `|` operator. Duplicate keys will be overwritten by the latter:

```
Plain Text
dict1 = {"a": 1, "b": 2}
dict2 = {"b": 3, "c": 4}
merged_dict = dict1 | dict2
print(merged_dict)  # Output: {'a': 1, 'b': 3, 'c': 4}
```

2. Nested Dictionaries (Dictionaries Containing Dictionaries)

Used to store multi-level data, such as "class-student-score":

```
Plain Text
# Nested dictionary: Class contains students, students contain
names and scores
class_info = {
    "Class 1": {
        "Zhang San": {"math": 90, "english": 85},
        "Li Si": {"math": 88, "english": 92}
    },
    "Class 2": {
        "Wang Wu": {"math": 95, "english": 88}
    }
}

# Access values in nested dictionaries
print(class_info["Class 1"]["Zhang San"]["math"])  # Output: 90
```

3. Converting Dictionaries to Other Data Types

```
Plain Text
# 1. Convert dictionary to list (convert keys or values)
fruit_price = {"apple": 5.8, "banana": 2.5}
keys_list = list(fruit_price.keys())  # Convert keys to list:
['apple', 'banana']
values_list = list(fruit_price.values())  # Convert values to
list: [5.8, 2.5]
```

```
# 2. Convert dictionary to tuple
items_tuple = tuple(fruit_price.items())  # Convert key-value
pairs to tuple: (('apple', 5.8), ('banana', 2.5))
```

Common Error Reminders: 1. Lists cannot be used as dictionary keys (lists are mutable types); for example, `{[1,2]: 3}` will raise a TypeError. 2. You cannot directly modify the length of a dictionary (e.g., add/delete keys) while iterating over it; you need to convert the keys to a list first before iterating.

V. Summary

With the feature of "key-value mapping", dictionaries perform excellently in data storage and query scenarios. Mastering the core operations of "creation-access-modification-iteration" and combining techniques such as dictionary comprehension and nested dictionaries can greatly improve data processing efficiency. It is recommended to practice through scenarios such as "storing user information, configuration parameters, and statistical data" to deepen understanding.