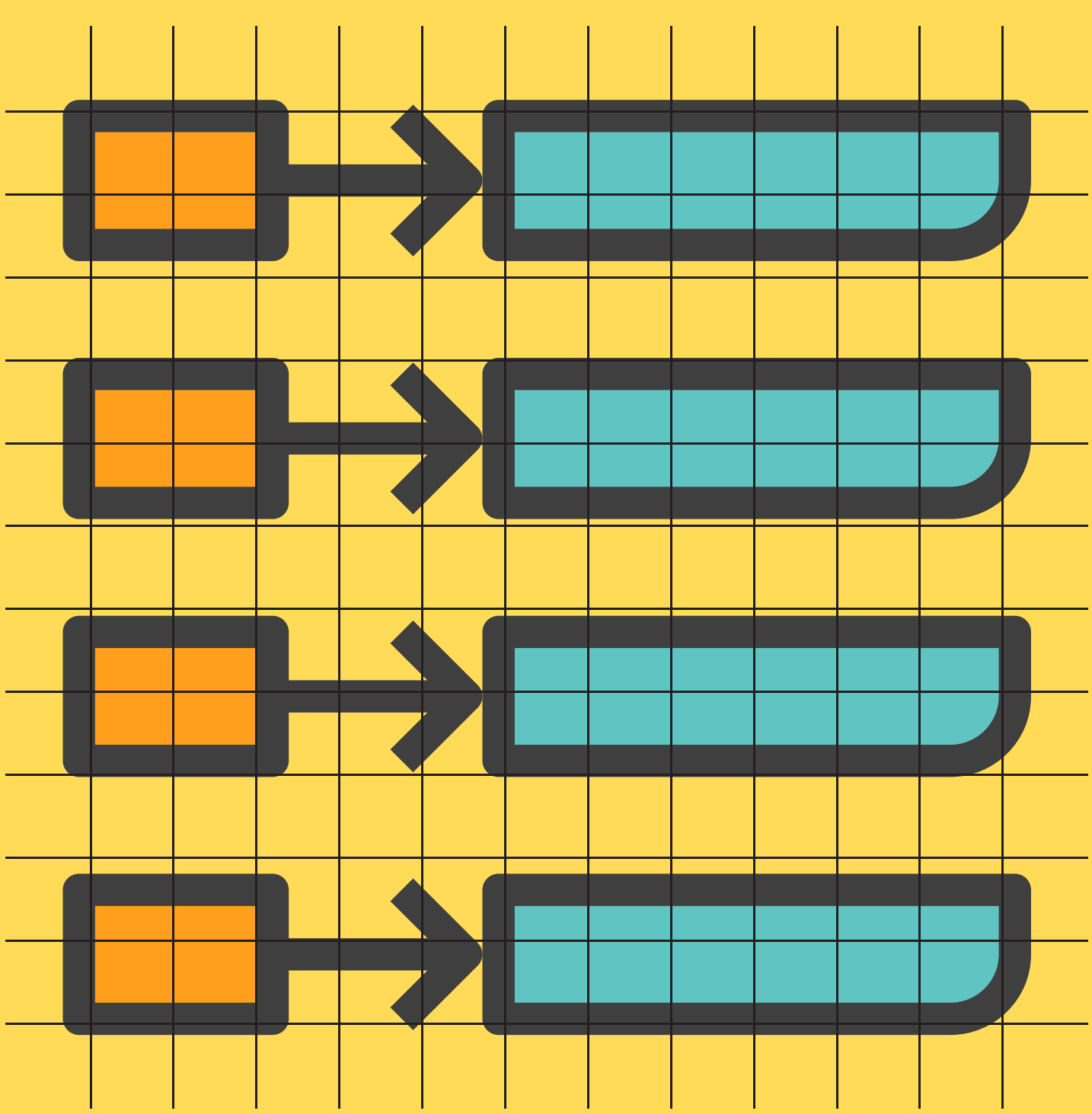
# {”Python”: “Dictonary”}

Everything you ever need to know



Your All-in-One Guide to Mastering Dictonary Operations

# Introduction

- **Definition**: Dictionaries are unordered, mutable collections of key-value pairs

- **Data Types**: Keys must be immutable (e.g., strings, numbers, tuples), values can be any type

- **Usage**: Often used for fast lookups and representing structured data

# Creating Dictionaries

- Use curly braces {} or the dict() constructor
- Examples

```
empty_dict = {}
person = {"name": "Alice", "age": 30, "city": "New York"}
grades = dict(Alice=95, Bob=87, Charlie=92)
```

# Accessing Dictionary Elements

- Use square brackets [] with the key to access values

- Use the get() method for safe access (returns None or a default value if key not found)

- Examples

```
print(person["name"])  # Output: Alice
print(grades.get("Bob", 0))  # Output: 87 (or 0 if "Bob" not in grades)
```

# Modifying Dictionaries

- Add or update key-value pairs using assignment

- Use update() method to add multiple key-value pairs

- Examples

```
print(person["name"])  # Output: Alice
print(grades.get("Bob", 0))  # Output: 87 (or 0 if "Bob" not in grades)
```

# Removing Items from Dictionaries

- Use del statement to remove a specific key-value pair
- pop() method removes and returns the value for a given key
- clear() method removes all items from the dictionary
    - Examples

```
del person["age"]
score = grades.pop("Alice")
grades.clear()
```

# DICTIONARY METHODS

# 1. keys()

- Returns a view object containing all keys in the dictionary

  - Examples

```
fruits = {"apple": 5, "banana": 3, "orange": 2}
keys = fruits.keys()
print(keys)  # Output: dict_keys(['apple', 'banana', 'orange'])

# Keys view updates when the dictionary changes
fruits["grape"] = 4
print(keys)  # Output: dict_keys(['apple', 'banana', 'orange', 'grape'])
```

# 2. values()

- Returns a view object containing all values in the dictionary

  - Examples

```
fruits = {"apple": 5, "banana": 3, "orange": 2}
values = fruits.values()
print(values)  # Output: dict_values([5, 3, 2])

# Values view updates when the dictionary changes
fruits["banana"] = 6
print(values)  # Output: dict_values([5, 6, 2])
```

# 3. items()

- Returns a view object containing all key-value pairs as tuples

  - Examples

```
fruits = {"apple": 5, "banana": 3, "orange": 2}
items = fruits.items()
print(items)
# Output: dict_items([('apple', 5), ('banana', 3), ('orange', 2)])

# Items view updates when the dictionary changes
fruits["grape"] = 4
print(items)
# Output: dict_items([('apple', 5), ('banana', 3), ('orange', 2), ('grape', 4
    )])
```

# 4. get(key[, default])

- Returns the value for the given key, or a default value if the key is not found

  - Examples

```
fruits = {"apple": 5, "banana": 3, "orange": 2}
print(fruits.get("banana"))       # Output: 3
print(fruits.get("grape"))        # Output: None
print(fruits.get("grape", 0))     # Output: 0
```

# 5. pop(key[, default])

- Removes and returns the value for the given key. Raises a KeyError if the key is not found and no default is provided

- Examples

```
fruits = {"apple": 5, "banana": 3, "orange": 2}
banana_count = fruits.pop("banana")
print(banana_count)  # Output: 3
print(fruits)        # Output: {'apple': 5, 'orange': 2}

# Using default value
grape_count = fruits.pop("grape", 0)
print(grape_count)   # Output: 0
```

# 6. popitem()

- Removes and returns an arbitrary (key, value) pair from the dictionary. Raises a KeyError if the dictionary is empty

- Examples

```
fruits = {"apple": 5, "banana": 3, "orange": 2}
item = fruits.popitem()
print(item)     # Output: ('orange', 2)
print(fruits)  # Output: {'apple': 5, 'banana': 3}
```

# 7. update([other])

- Updates the dictionary with key-value pairs from another dictionary or iterable of key-value pairs

- Examples

```
fruits = {"apple": 5, "banana": 3}
more_fruits = {"orange": 2, "grape": 4}
fruits.update(more_fruits)
print(fruits)  # Output: {'apple': 5, 'banana': 3, 'orange': 2, 'grape': 4}

# Update with keyword arguments
fruits.update(pear=3, mango=6)
print(fruits)
# Output: {'apple': 5, 'banana': 3, 'orange': 2, 'grape': 4, 'pear': 3,
    'mango': 6}
```

# 8. setdefault(key[, default])

- Returns the value of the key if it exists, otherwise inserts the key with the given default value and returns the default

- Examples

```
fruits = {"apple": 5, "banana": 3}
orange_count = fruits.setdefault("orange", 0)
print(orange_count)  # Output: 0
print(fruits)        # Output: {'apple': 5, 'banana': 3, 'orange': 0}

apple_count = fruits.setdefault("apple", 0)
print(apple_count)   # Output: 5 (existing value is returned)
```

# 9. copy()

- Returns a shallow copy of the dictionary
- Examples

```python
original = {"a": 1, "b": [2, 3]}
copied = original.copy()
print(copied)  # Output: {'a': 1, 'b': [2, 3]}

# Modifying the copy doesn't affect the original
copied["a"] = 10
print(original)  # Output: {'a': 1, 'b': [2, 3]}
print(copied)    # Output: {'a': 10, 'b': [2, 3]}

# But nested mutable objects are shared
copied["b"].append(4)
print(original)  # Output: {'a': 1, 'b': [2, 3, 4]}
print(copied)    # Output: {'a': 10, 'b': [2, 3, 4]}
```

# 10. clear()

- Removes all items from the dictionary
- Examples

```
fruits = {"apple": 5, "banana": 3, "orange": 2}
fruits.clear()
print(fruits)  # Output: {}
```

# DICTIONARY

# VIEW OBJECTS

# Introduction

Dictionary view objects are returned by the keys(), values(), and items() methods.

They provide a dynamic view of the dictionary's entries, which means they change as the dictionary changes

Key features of view objects

- Dynamic updates
- Support for set operations
- Iterable
- Length and membership testing

# Dynamic updates

- Examples

```
fruits = {"apple": 5, "banana": 3}
keys_view = fruits.keys()
print(keys_view)
# Output: dict_keys(['apple', 'banana'])

fruits["orange"] = 2
print(keys_view)
# Output: dict_keys(['apple', 'banana', 'orange'])
```

# Support for set operations

- Examples

```
dict1 = {"a": 1, "b": 2, "c": 3}
dict2 = {"b": 2, "c": 3, "d": 4}

keys1 = dict1.keys()
keys2 = dict2.keys()

print(keys1 & keys2)  # Intersection: {'b', 'c'}
print(keys1 | keys2)  # Union: {'a', 'b', 'c', 'd'}
print(keys1 - keys2)  # Difference: {'a'}
```

# Iterable

- Examples

```python
fruits = {"apple": 5, "banana": 3, "orange": 2}
for fruit in fruits.keys():
    print(fruit)
# Output:
# apple
# banana
# orange

for value in fruits.values():
    print(value)
# Output:
# 5
# 3
# 2

for fruit, count in fruits.items():
    print(f"{fruit}: {count}")
# Output:
# apple: 5
# banana: 3
# orange: 2
```

# Length and membership testing

- Examples

```
fruits = {"apple": 5, "banana": 3, "orange": 2}
keys_view = fruits.keys()

print(len(keys_view))  # Output: 3
print("banana" in keys_view)  # Output: True
print("grape" in keys_view)   # Output: False
```

# DICTIONARY

# BUILT-IN FUNCTIONS

# 1. len(dict)

- Returns the number of key-value pairs in the dictionary

- Examples

```
fruits = {"apple": 5, "banana": 3, "orange": 2}
print(len(fruits))  # Output: 3
```

# 2. sorted(dict)

- Returns a new sorted list of keys from the dictionary

- Examples

```python
grades = {"Alice": 85, "Bob": 92, "Charlie": 78, "David": 95}
sorted_names = sorted(grades)
print(sorted_names)
# Output: ['Alice', 'Bob', 'Charlie', 'David']

# Sort by values
sorted_by_grades = sorted(grades, key=grades.get, reverse=True)
print(sorted_by_grades)
# Output: ['David', 'Bob', 'Alice', 'Charlie']
```

# 3. any(dict)

- Returns True if any key in the dictionary is True (non-zero, non-empty, or True)

- Examples

```
dict1 = {0: False, "": [], (): {}}
dict2 = {0: False, 1: True, 2: False}
dict3 = {"a": [], "b": [1, 2, 3]}


print(any(dict1))  # Output: False
print(any(dict2))  # Output: True
print(any(dict3))  # Output: True
```

# 4. all(dict)

- Returns True if all keys in the dictionary are True (non-zero, non-empty, or True)

- Examples

```
dict1 = {1: True, 2: False, 3: True}
dict2 = {1: True, "a": [1, 2], 3: {}}
dict3 = {0: False, "": [], (): {}}

print(all(dict1))  # Output: True
print(all(dict2))  # Output: True
print(all(dict3))  # Output: False
```

# 5. dict()

- Creates a new dictionary. Can be used to create dictionaries from various input types

- Examples

```
# Empty dictionary
empty_dict = dict()
print(empty_dict)  # Output: {}

# From a list of tuples
items = [("a", 1), ("b", 2), ("c", 3)]
dict_from_items = dict(items)
print(dict_from_items)  # Output: {'a': 1, 'b': 2, 'c': 3}

# From keyword arguments
dict_from_kwargs = dict(x=10, y=20, z=30)
print(dict_from_kwargs)  # Output: {'x': 10, 'y': 20, 'z': 30}


# From two lists using zip()
keys = ["name", "age", "city"]
values = ["Alice", 30, "New York"]
dict_from_zip = dict(zip(keys, values))
print(dict_from_zip)
# Output: {'name': 'Alice', 'age': 30, 'city': 'New York'}
```

# DICTIONARY

# USE CASES

# 1. Caching and Memoization

- Dictionaries are excellent for storing computed results to avoid redundant calculations

- Examples

```python
def fibonacci(n, cache={}):
    # Check if the value for n is already in the cache
    if n in cache:
        return cache[n]

    # Base case: if n is 0 or 1, return n
    if n <= 1:
        return n

    # Recursive case: compute the Fibonacci number
    result = fibonacci(n-1, cache) + fibonacci(n-2, cache)

    # Store the computed result in the cache
    cache[n] = result

    return result

# Example usage: Calculate the 100th Fibonacci number
# This will be computed quickly due to the caching mechanism
print(fibonacci(100))
```

# 2. Configuration Settings

- Store application settings as key-value pairs for easy access and modification

  - Examples

```
# Configuration dictionary containing settings for database, API, and logging
config = {
    "database": {
        "host": "localhost",  # Database host address
        "port": 5432,         # Database port number
        "user": "admin",      # Database username
        "password": "secret"  # Database password
    },
    "api": {
        "url": "https://api.example.com",  # API endpoint URL
        "key": "your-api-key"              # API access key
    },
    "logging": {
        "level": "INFO",         # Logging level
        "file": "/var/log/app.log"  # Path to the log file
    }
}

# Access and print specific configuration values
print(f"Database host: {config['database']['host']}")
print(f"API URL: {config['api']['url']}")
```

# 3. JSON-like Data Structures

- Represent structured data in a human-readable format, often used for API responses or configuration files

- Examples

```python
import json

user_data = {
    "id": 12345,
    "name": "Alice Smith",
    "email": "alice@example.com",
    "active": True,
    "preferences": {
        "theme": "dark",
        "notifications": ["email", "push"]
    }
}

# Convert to JSON string
json_data = json.dumps(user_data, indent=2)
print(json_data)

# Parse JSON string back to dictionary
parsed_data = json.loads(json_data)
print(parsed_data["preferences"]["theme"])  # Output: dark
```

# 4. Frequency Counting

- Count occurrences of items in a collection efficiently

- Examples

```python
from collections import defaultdict

def word_frequency(text):
    words = text.lower().split()
    frequency = defaultdict(int)
    for word in words:
        frequency[word] += 1
    return dict(frequency)

text = "The quick brown fox jumps over the lazy dog. The dog barks."
freq = word_frequency(text)
print(freq)
# Output: {'the': 2, 'quick': 1, 'brown': 1, 'fox': 1, 'jumps': 1, 'over': 1,
    'lazy': 1, 'dog.': 1, 'dog': 1, 'barks.': 1}
```

# 5. Graph Representations

- Represent adjacency lists for graph algorithms

- Examples

```
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=' ')
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

print("DFS traversal:")
dfs(graph, 'A')  # Output: A B D E F C
```

# 6. Lookup Tables

- Create efficient mappings for quick data retrieval

- Examples

```python
morse_code = {
    'A': '.-', 'B': '-...', 'C': '-.-.', 'D': '-..', 'E': '.', 'F': '..-.',
    'G': '--.', 'H': '....', 'I': '..', 'J': '.---', 'K': '-.-', 'L': '.-..',
    'M': '--', 'N': '-.', 'O': '---', 'P': '.--.', 'Q': '--.-', 'R': '.-.',
    'S': '...', 'T': '-', 'U': '..-', 'V': '...-', 'W': '.--', 'X': '-..-',
    'Y': '-.--', 'Z': '--..', ' ': ' '
}

def encode_morse(message):
    return ' '.join(morse_code.get(char.upper(), '') for char in message)

message = "Hello World"
encoded = encode_morse(message)
print(encoded)
# Output: .... . .-.. .-.. ---  .-- --- .-. .-.. -..
```

# DICTIONARY

## ADVANCED CONCEPTS

# 1. Default Dictionaries (collections.defaultdict)

- Automatically initializes new keys with a default value

- Examples

```
from collections import defaultdict
fruit_counts = defaultdict(int)
fruit_counts['apple'] += 1  # No KeyError, default value is 0
```

# 2. Ordered Dictionaries (collections.OrderedDict)

- Remembers the order in which keys were inserted

- Examples

```
from collections import OrderedDict

# Creating an OrderedDict
ordered_dict = OrderedDict()
ordered_dict['banana'] = 3
ordered_dict['apple'] = 4
ordered_dict['orange'] = 2

# Iterating over OrderedDict
for key, value in ordered_dict.items():
    print(key, value)
```

Output:

Copy code

```
banana 3
apple 4
orange 2
```

# 3. Counter Dictionaries (collections.Counter)

- Specialized dictionary for counting hashable objects

- Examples

```python
from collections import Counter
word_counts = Counter("mississippi")
print(word_counts)
# Result: Counter({'i': 4, 's': 4, 'p': 2, 'm': 1})
```

# 4. Dictionary Merging (Python 3.9+)

- Use the | operator to merge dictionaries

- Examples

```python
dict1 = {"a": 1, "b": 2}
dict2 = {"b": 3, "c": 4}
merged = dict1 | dict2
print(merged)
# Result: {"a": 1, "b": 3, "c": 4}
```

# DICTIONARY

# COMMON PITFALLS AND BEST PRACTICES

- **Key Errors:** Use get() or setdefault() to avoid KeyError exceptions

- **Mutable Keys:** Don't use mutable objects (like lists) as dictionary keys

- **Memory Usage:** Be cautious with large dictionaries, as they can consume significant memory

- **Default Values:** Use dict.get(key, default) instead of checking if key in dict

- **Copying:** Use dict.copy() for shallow copies or copy.deepcopy() for deep copies

- **Key Existence:** Use in operator to check for key existence instead of exceptions

# DICTIONARY

# DEBUGGING ISSUES

**Common Errors:**

- **KeyError**: Occurs when trying to access a non-existent key
- **TypeError**: Unhashable type (e.g., using a list as a key)

**Debugging Tips:**

- Use print() to inspect dictionary contents
- Utilize the in operator to check for key existence before accessing
- Use pprint module for pretty-printing complex dictionaries

# DICTIONARY

## VS OTHER DATA STRUCTURES

1. **Dictionaries vs Lists**

    - Dictionaries have O(1) average-case lookup, lists have O(n)

    - Dictionaries are unordered (before Python 3.7), lists are ordered

    - Use dictionaries when you need fast lookups by key

2. **Dictionaries vs Sets**

    - Both have fast lookups, but dictionaries store key-value pairs

    - Use sets for membership testing, dictionaries for associating values with keys

### 3. Dictionaries vs Tuples

- Dictionaries are mutable and use keys for access, tuples are immutable and use indices

- Use dictionaries for named access to elements, tuples for fixed collections

# DICTIONARY

# TRY THESE PROJECTS

1. **Word Frequency Analyzer**

    - Build a program that counts word frequencies in a given text using dictionaries

2. **Configuration Parser**

    - Create a program that reads and writes configuration files using dictionaries

3. **Contact Book Application**

    - Create a dictionary-based contact book with functions to add, update, delete, and search contacts

Naresh Saginala

Agile Coach

# Repost

# If you find this helpful

@nareshsaginala