Topic 3c: Dictionaries

Key-Value Pairs

Learning Outcomes:

- Understand the purpose of dictionaries (key-value mapping)
- Apply dictionary syntax and operations
- Perform CRUD operations on dictionaries
- Distinguish when to use dictionaries vs lists vs tuples vs sets

Prerequisites: Topic 2 (Lists), Topic 3a (Tuples), Topic 3b (Sets)

Opening Problem

You're building a student record system.

You need to store multiple pieces of information about each student.

```python
# Using separate variables
student_name = "Ali"
student_age = 20
student_course = "Computer Science"

print("Name:", student_name)
print("Age:", student_age)
print("Course:", student_course)
```

What could go wrong?

The Problem: Scattered Data

```python
# Student 1
student1_name = "Ali"
student1_age = 20
student1_course = "Computer Science"

# Student 2
student2_name = "Sara"
student2_age = 19
student2_course = "Mathematics"

# Student 3
student3_name = "Ahmad"
student3_age = 21
student3_course = "Physics"

# Variables are scattered. Hard to manage!
```

Problems:

Why Not Use a List?

```python
# Trying with a list
student = ["Ali", 20, "Computer Science"]

print(student[0])  # Ali
print(student[1])  # 20
print(student[2])  # Computer Science
```

Problems:

- What is index 0? Index 1? Index 2?
- You must remember the positions
- If order changes, code breaks silently

There must be a better way.

The Solution: Dictionaries

Dictionaries use descriptive keys instead of numeric indices.

```python
student = {
    "name": "Ali",
    "age": 20,
    "course": "Computer Science"
}

print(student["name"])    # Ali
print(student["age"])     # 20
print(student["course"])  # Computer Science
```

Keys describe what each value represents. No more guessing!

Dictionaries Group Related Data

```python
student = {
    "name": "Ali",
    "age": 20,
    "course": "Computer Science"
}

# Pass entire student to a function
def display_student(stu):
    print("Name:", stu["name"])
    print("Age:", stu["age"])
    print("Course:", stu["course"])

display_student(student)
```

All data about one entity is in one place.

Where Dictionaries Fit

Python provides 4 built-in data structures:

Type	Symbol	Mutable?	Ordered?	Duplicates?
List	`[ ]`	Yes	Yes	Yes
Tuple	`( )`	No	Yes	Yes
Set	`{ }`	Yes	No	No
Dictionary	`{k: v}`	**Yes**	**Yes**	**No (keys)**

Key difference: Dictionaries store key-value pairs, not just values.

Dictionary vs List vs Tuple vs Set

Need	Use
Ordered collection that can change	List
Ordered collection that cannot change	Tuple
Unique items, order doesn't matter	Set
Named access to values	**Dictionary**

```python
# List: access by position
scores = [85, 92, 78]
print(scores[0])  # What is this score for?

# Dictionary: access by name
scores = {"midterm": 85, "final": 92, "assignment": 78}
print(scores["midterm"])  # Clear!
```

Structure of a Dictionary

```python
student = {"name": "Ali", "age": 20, "course": "CS"}
```

Breaking it down:

```
     dict name    key    value  key  value
         |         |       |     |     |
         ▼         ▼       ▼     ▼     ▼
     student = {"name": "Ali", "age": 20}
               │                        │
               └─── curly braces { } ───┘
```

Component	Description
Dictionary name	`student`
Keys	`"name"`, `"age"`, `"course"`

Dictionary Rules

Keys:

- Must be **unique** (no duplicate keys)
- Must be **immutable** (strings, numbers, tuples)
- Cannot be lists or dictionaries

Values:

- Can be **any type** (strings, numbers, lists, even other dictionaries)
- Can be **duplicated**

Exercise 1: Identify the Parts

Given this dictionary:

```
product = {"id": 101, "name": "Laptop", "price": 2500.00}
```

Identify:

1. What are the keys?
2. What are the values?
3. What is the value associated with key `"price"`?

How Dictionaries Work: Hashing

Why are dictionaries so fast?

When you access `student["name"]`, Python doesn't search through all keys.

Instead, it uses **hashing** to find the value instantly.

Visualizing: Buckets

Think of a dictionary as a row of buckets:

```
Bucket:   [0]     [1]     [2]     [3]     [4]     [5]     [6]     [7]
           │       │       │       │       │       │       │       │
           ▼       ▼       ▼       ▼       ▼       ▼       ▼       ▼
         empty   empty    20    empty   empty   "Ali"  empty   empty
                          ▲                       ▲
                          │                       │
                        "age"                  "name"
```

Each key is placed in a bucket based on its **hash value**.

Step 1: Adding Entries

When you add a key-value pair, Python calculates a hash for the key:

```
student = {"name": "Ali", "age": 20}

Key         Hash         Bucket

"name"  →  hash = 5  →  bucket 5  →  stores "Ali"
"age"   →  hash = 2  →  bucket 2  →  stores 20
```

Each key gets a unique "address" (bucket) based on its hash value.

Step 2: Accessing Values

When you access a value, Python uses the hash to find the bucket directly:

```
student["name"]?

"name"  →  hash = 5  →  go to bucket 5

Bucket:   [0]     [1]     [2]     [3]     [4]     [5]     [6]     [7]
           |       |       |       |       |       |       |       |
           ▼       ▼       ▼       ▼       ▼       ▼       ▼       ▼
         empty   empty    20    empty   empty   "Ali"  empty   empty
                                                  ▲
                                                  |
                                           Found "Ali"!
```

No need to check every bucket. Python goes directly to the correct one.

This is why dictionary lookups are fast, regardless of size.

Why Keys Must Be Immutable

What if a key could change after being added?

```
Step 1: Add [1, 2] as key (hash = 7, placed in bucket 7)

Bucket:   [0]     [1]     [2]     [3]     [4]     [5]     [6]     [7]
           │       │       │       │       │       │       │       │
           ▼       ▼       ▼       ▼       ▼       ▼       ▼       ▼
         empty   empty   empty   empty   empty   empty   empty  "value"
                                                                  ▲
                                                                  │
                                                                [1, 2]
```

Why Keys Must Be Immutable (Continued)

```
Step 2: Key [1, 2] changes to [1, 2, 3] (new hash = 9)

Bucket:   [0]     [1]     [2]   ...   [7]     [8]     [9]
           |       |       |           |       |       |
           ▼       ▼       ▼           ▼       ▼       ▼
         empty   empty   empty      "value"  empty   empty
                                      ▲               ▲
                                      |               |
                              still here!      Python looks here!

Python looks in bucket 9, but the value is still in bucket 7!
```

This is why Python only allows immutable keys (strings, numbers, tuples).

The TypeError

Trying to use a mutable key:

```
my_dict = {}
my_dict[[1, 2]] = "value"  # List as key
```

Output:

```
TypeError: unhashable type: 'list'
```

"Unhashable" means Python cannot calculate a stable hash for this type.

Dictionary Characteristics

3 key characteristics:

1. **Ordered** (Python 3.7+) — Maintains insertion order
2. **Mutable** — Can add, update, delete entries
3. **Heterogeneous Values** — Values can be different types

Characteristic 1: Ordered

Dictionaries maintain the order in which items were added.

```
student = {"name": "Ali", "age": 20, "course": "CS"}
print(student)
```

Output:

```
{'name': 'Ali', 'age': 20, 'course': 'CS'}
```

The order is preserved: name, then age, then course.

Characteristic 2: Mutable

You can modify dictionaries after creation.

```python
student = {"name": "Ali", "age": 20}

# Update existing value
student["age"] = 21

# Add new key-value pair
student["grade"] = "A"

print(student)
```

Output:

```
{'name': 'Ali', 'age': 21, 'grade': 'A'}
```

Characteristic 3: Heterogeneous Values

Values can be different data types:

```python
data = {
    "name": "Ali",           # String
    "scores": [95, 88, 92],  # List
    "details": {"city": "KL", "age": 21}  # Nested dictionary
}

print(data["scores"][1])        # 88
print(data["details"]["city"])  # KL
```

Characteristics Summary

Characteristic	Description
Ordered (Python 3.7+)	Maintains insertion order
Mutable	Can add, update, delete entries
Heterogeneous Values	Values can be any data type

CRUD Operations Overview

Operation	Description
Create	Make a new dictionary or add entries
Read	Access values by key
Update	Modify existing values
Delete	Remove entries

CREATE: Making Dictionaries

Method 1: Empty dictionary

```
my_dict = {}
# or
my_dict = dict()
```

Method 2: With initial values

```
student = {"name": "Ali", "age": 20, "grade": "A"}
```

Method 3: Using dict() constructor

```
student = dict(name="Ali", age=20, grade="A")
print(student)  # {'name': 'Ali', 'age': 20, 'grade': 'A'}
```

CREATE: Adding New Entries

Syntax:

```
dictionary_name[key] = value
```

Example:

```
student = {"name": "Ali", "age": 20}

# Add new key-value pair
student["course"] = "Computer Science"

print(student)
```

Output:

```
{'name': 'Ali', 'age': 20, 'course': 'Computer Science'}
```

CREATE: Using update()

Add multiple entries at once:

```
student = {"name": "Ali", "age": 20}

student.update({"course": "CS", "grade": "A"})

print(student)
```

Output:

```
{'name': 'Ali', 'age': 20, 'course': 'CS', 'grade': 'A'}
```

Exercise 2: Create a Dictionary

Create a dictionary called `book` with:

- title: "Python Basics"
- author: "John Doe"
- price: 50

Then add a new key:

- pages: 300

Print the final dictionary.

READ: Accessing Values

Method 1: Using square brackets

```
student = {"name": "Ali", "age": 20, "grade": "A"}

print(student["name"])   # Ali
print(student["age"])    # 20
```

Warning: If key doesn't exist, raises KeyError!

```
print(student["height"])  # KeyError: 'height'
```

READ: Using get() — Safe Access

The get() method returns a default value if key not found:

```python
student = {"name": "Ali", "age": 20, "grade": "A"}

# Key exists
print(student.get("name"))          # Ali

# Key doesn't exist – returns None
print(student.get("height"))        # None

# Key doesn't exist – returns custom default
print(student.get("height", "N/A"))  # N/A
```

No error is raised!

READ: Getting All Keys, Values, Items

```
student = {"name": "Ali", "age": 20, "grade": "A"}

# All keys
print(student.keys())    # dict_keys(['name', 'age', 'grade'])

# All values
print(student.values())  # dict_values(['Ali', 20, 'A'])

# All key-value pairs
print(student.items())   # dict_items([('name', 'Ali'), ('age', 20), ('grade', 'A')])
```

READ: Looping Over Keys

```
student = {"name": "Ali", "age": 20, "grade": "A"}

for key in student:
    print(key)
```

Output:

```
name
age
grade
```

READ: Looping Over Values

```
student = {"name": "Ali", "age": 20, "grade": "A"}

for value in student.values():
    print(value)
```

Output:

```
Ali
20
A
```

READ: Looping Over Key-Value Pairs

```
student = {"name": "Ali", "age": 20, "grade": "A"}

for key, value in student.items():
    print(key, ":", value)
```

Output:

```
name : Ali
age : 20
grade : A
```

Exercise 3: Read from Dictionary

Given:

```python
student = {"name": "Ali", "age": 20, "grade": "A"}
```

Tasks:

1. Access the value of `"name"` using square brackets
2. Access the value of `"height"` using get() with default "Unknown"
3. Print all keys
4. Loop through and print all key-value pairs

UPDATE: Modifying Existing Values

Syntax:

```
dictionary_name[existing_key] = new_value
```

Example:

```
student = {"name": "Ali", "age": 20, "grade": "B"}

# Update existing value
student["grade"] = "A"

print(student)
```

Output:

```
{'name': 'Ali', 'age': 20, 'grade': 'A'}
```

UPDATE: Using update()

Update multiple values at once:

```
student = {"name": "Ali", "age": 20, "grade": "B"}

student.update({"age": 21, "grade": "A"})

print(student)
```

Output:

```
{'name': 'Ali', 'age': 21, 'grade': 'A'}
```

Note: `update()` can both add new keys and modify existing ones.

Exercise 4: Update a Dictionary

Given:

```python
book = {"title": "Python Basics", "author": "John Doe", "price": 50}
```

Tasks:

1. Update the price to 55
2. Add a new key "edition" with value 2
3. Print the updated dictionary

DELETE: Removing Entries

Method 1: Using del

```
student = {"name": "Ali", "age": 20, "grade": "A"}

del student["age"]

print(student)  # {'name': 'Ali', 'grade': 'A'}
```

Warning: Raises KeyError if key doesn't exist!

DELETE: Using pop()

Remove and return the value:

```
student = {"name": "Ali", "age": 20, "grade": "A"}

removed_value = student.pop("age")

print(removed_value)  # 20
print(student)        # {'name': 'Ali', 'grade': 'A'}
```

With default value (no error if key missing):

```
removed = student.pop("height", "Not found")
print(removed)  # Not found
```

DELETE: Using clear()

Remove all entries:

```
student = {"name": "Ali", "age": 20, "grade": "A"}

student.clear()

print(student)  # {}
```

DELETE Summary

Method	Behavior
`del dict[key]`	Remove key. Raises KeyError if not found
`dict.pop(key)`	Remove and return value. Raises KeyError if not found
`dict.pop(key, default)`	Remove and return value. Returns default if not found
`dict.clear()`	Remove all entries

Exercise 5: Delete from Dictionary

Given:

```
employee = {"name": "Sara", "department": "IT", "salary": 5000}
```

Tasks:

1. Remove the "salary" key using del
2. Print the dictionary
3. Try to remove "position" using pop() with a default value
4. Clear the entire dictionary

Checking Key Existence

Use the `in` keyword:

```
student = {"name": "Ali", "age": 20, "grade": "A"}

if "name" in student:
    print("Key exists!")
else:
    print("Key does not exist!")
```

Output:

```
Key exists!
```

Safe Access Pattern

Always check before accessing with square brackets:

```
student = {"name": "Ali", "age": 20}

key = "height"

if key in student:
    print(student[key])
else:
    print(f"{key} not found")
```

Or simply use get():

```
print(student.get("height", "Not found"))
```

Exercise 6: Check Key Existence

Given:

```python
person = {"name": "John", "age": 30, "city": "New York"}
```

Write code to:

1. Check if "age" exists in the dictionary
2. Check if "country" exists in the dictionary
3. Print appropriate messages for each

The Principled Approach

2 principles for using dictionaries properly:

1. Use Descriptive String Keys
2. Group Related Data in One Dictionary

Principle 1: Use Descriptive String Keys

The Problem:

Keys can technically be any immutable type — integers, floats, tuples. But what do they mean?

```python
# What do these keys represent?
data = {1: "Ali", 2: 20, 3: 85}

# Keys are strings but not descriptive
data = {"a": "Ali", "b": 20, "c": 85}

# Inconsistent naming style
student = {"Name": "Ali", "AGE": 20, "student_score": 85}
```

Principle 1: The Solution

Use descriptive strings with consistent naming:

```
student = {
    "name": "Ali",
    "age": 20,
    "score": 85
}
```

Why strings?

- Self-documenting: `"name"` tells you the value is a name
- Readable: Code reads like English
- Consistent: Follow a pattern (lowercase with underscores)

The Rule: Use descriptive string keys that explain what the value represents. Follow a consistent naming pattern.

Principle 2: Group Related Data in One Dictionary

The Problem:

```python
# Scattered variables about one entity
student_name = "Ali"
student_age = 20
student_score = 85
student_course = "CP125"

# To pass to a function, need 4 parameters!
def display(name, age, score, course):
    print(name, age, score, course)
```

Principle 2: The Solution

```python
# One dictionary groups all related data
student = {
    "name": "Ali",
    "age": 20,
    "score": 85,
    "course": "CP125"
}

# Pass one parameter
def display(stu):
    print(stu["name"], stu["age"], stu["score"], stu["course"])
```

The Rule: If multiple pieces of data describe one entity, put them in a single dictionary.

Principles Summary

#	Principle	Rule
1	Descriptive String Keys	Strings describe values, consistent naming
2	Group Related Data	One entity = one dictionary

Dictionary vs Tuple vs Set

Feature	Dictionary	Tuple	Set
Mutable	Yes	No	Yes
Ordered	Yes (3.7+)	Yes	No
Duplicates	No (keys)	Yes	No
Access by	Key	Index	Iteration only
Syntax	`{k: v}`	`(a, b)`	`{a, b}`

When to Use Each

Use Case	Best Structure
Fixed, unchangeable data	Tuple
Unique items, order doesn't matter	Set
Named attributes for an entity	**Dictionary**
Ordered collection that changes	List

```
# Tuple: Coordinates (fixed)
point = (10, 20)

# Set: Unique visitors
visitors = {101, 102, 103}

# Dictionary: Student record (named attributes)
student = {"name": "Ali", "age": 20}
```

Common Mistakes

1. Accessing non-existent key without get():

```
student = {"name": "Ali"}
print(student["age"])  # KeyError!
```

2. Using mutable types as keys:

```
my_dict = {[1, 2]: "value"}  # TypeError!
```

Common Mistakes (Continued)

3. Confusing empty dict with empty set:

```python
empty = {}          # This is a dictionary, not a set!
empty = set()       # This is a set
```

4. Forgetting that keys must be unique:

```python
data = {"a": 1, "a": 2}
print(data)  # {'a': 2} — first value overwritten!
```

Summary

What we covered:

1. **Dictionaries store key-value pairs** — named access to data
2. **Syntax:** Curly braces `{key: value}`
3. **Characteristics:** Ordered, mutable, heterogeneous values
4. **CRUD Operations:** Create, Read, Update, Delete
5. **Key existence:** Use `in` keyword or `.get()`
6. **Keys:** Must be immutable (prefer descriptive strings)
7. **Principles:** Descriptive keys, group related data

Dictionary Quick Reference

Operation	Syntax
Create	`d = {"key": "value"}`
Access	`d["key"]` or `d.get("key")`
Add/Update	`d["key"] = value`
Delete	`del d["key"]` or `d.pop("key")`
Check exists	`"key" in d`
All keys	`d.keys()`
All values	`d.values()`
All pairs	`d.items()`
Loop	`for k, v in d.items():`

Principles Quick Reference

#	Principle	Rule
1	Descriptive String Keys	Keys explain values, consistent style
2	Group Related Data	One entity = one dictionary

What's Next

You have now learned all 4 Python data structures:

Structure	Use Case
List	Ordered, changeable collection
Tuple	Ordered, unchangeable collection
Set	Unique items, fast membership
Dictionary	Key-value mapping

Next topics will build on these foundations.