MODULE 1: CORE PYTHON & DATA

WEEK: 1 LECTURE: 5

DATE: 21/08/2025

INSTRUCTOR: ORANGZAIB RAJPOOT

PYTHON'S COLLECTIONS (PART 2): DICTIONARIES & SETS

Last time, we explored sequences like lists and tuples, where data is stored in an ordered line, accessed by a numerical index. Today, we unlock a new dimension of data organization with **Dictionaries**, which store data based on a custom key, and **Sets**, which are optimized for storing unique items. These structures are fundamental to everything from web development to data science.

TODAY'S AGENDA

- **The Ultimate Filing Cabinet - Dictionaries**
  - Beyond the Index: The Power of Key-Value Pairs
  - Creating and Understanding Dictionaries
  - CRUD Operations: Create, Read, Update, Delete
  - Safe Access with .get() to Avoid Errors
- **Advanced Dictionaries & Introduction to Sets**
  - Iterating Over Dictionaries: .keys(), .values(), .items()
  - Nesting: Storing Lists and Dictionaries inside Dictionaries
  - **Introduction to Sets: The Uniqueness Enforcer**
  - Creating Sets and Key Properties (Unordered, Unique)
  - *Interactive Exercises on Iteration and Nesting*
- **Set Operations & The Hands-On Lab**
  - Practical Set Operations: Union, Intersection, Difference
  - Use Cases: When to Use a Set vs. a List
  - **Hands-On Lab: The Contact Book Application**
  - Q&A and Wrap-up

THE ULTIMATE FILING CABINET - DICTIONARIES

BEYOND THE INDEX: KEY-VALUE PAIRS

Imagine a filing cabinet. You don't find a file by knowing it's the "5th file from the front." You look for it by its label, like "Project Alpha" or "Client XYZ."

A Python **dictionary** works the same way. It's an **unordered** collection of **key-value pairs**.

- **Key:** The unique label you use to look up data. Keys must be **immutable** (strings, numbers, and tuples are great keys; lists are not).
- **Value:** The data associated with a key. The value can be of **any data type**—a string, a number, a list, even another dictionary.

Dictionaries are enclosed in curly braces {}.

```
# A simple dictionary representing a user
user = {
    "username": "ada_lovelace",
    "email": "ada@example.com",
    "id": 1815,
    "is_active": True
}
```

CREATE / UPDATE (THEY ARE THE SAME OPERATION!)

You can add a new key-value pair or update an existing one using square bracket assignment.

```
# Create a new key-value pair
user["last_login"] = "2025-08-20"

# Update an existing value
user["email"] = "ada.lovelace@newdomain.com"

print(user)
```

READ / ACCESS

You access a value by its key.

```
print(user["username"]) # Output: ada_lovelace
```

Danger Zone: If you try to access a key that doesn't exist, you'll get a KeyError, which will crash your program.

SAFE ACCESS WITH .GET()

The .get() method is the preferred way to access data safely. It returns the value if the key exists, or None (a special null value) if it doesn't. You can also provide a default value.

```
# Safe access
location = user.get("location")
print(location) # Output: None

# Safe access with a default value
location = user.get("location", "Location not specified")
print(location) # Output: Location not specified
```

DELETE

Use the del keyword or the .pop() method.

```
# Using del
del user["is_active"]

# Using .pop() - it removes the pair and returns the value
user_id = user.pop("id")
print(f"Removed user with ID: {user_id}")
```

IN-CLASS EXERCISE: CAR INVENTORY

- Create a dictionary car with keys "make", "model", and "year".
- Add a new key "color" with a value of your choice.
- Update the "year" to the current year.
- Safely get the value for the key "engine_type", providing a default value of "unknown".
- Print the final dictionary.

ADVANCED DICTIONARIES

How do you loop through all the data? Python gives you three clear ways.

```
student = {"name": "Alan Turing", "major": "Computer Science", "id": 1912}


# Method 1: Looping over keys (the default)
print("--- KEYS ---")
for key in student:
    print(f"{key}: {student[key]}")
```

```
# Method 2: Looping over values
print("\n--- VALUES ---")
for value in student.values():
    print(value)

# Method 3: Looping over key-value pairs (THE BEST WAY!)
# .items() returns a view object of (key, value) tuples
print("\n--- ITEMS ---")
for key, value in student.items(): # This uses tuple unpacking!
    print(f"The student's {key} is {value}.")
```

NESTING: BUILDING COMPLEX STRUCTURES

The real power of dictionaries shines when you nest them. The value of a key can be a list or even another dictionary. This is how complex data, like JSON from a web API, is structured.

```
# A dictionary where a value is a list
course = {
    "name": "Intro to AI",
    "course_code": "CS101",
    "students": ["Alice", "Bob", "Charlie"]
}
# A dictionary where a value is another dictionary
users = {
    "user_123": {"name": "Alice", "email": "alice@example.com"},
    "user_456": {"name": "Bob", "email": "bob@example.com"}}
```

```
# Accessing nested data
first_student = course["students"][0]
#Access the list, then index it

bob_email = users["user_456"]["email"]
# Access the outer dict, then the inner one
```

IN-CLASS EXERCISE: COURSE ROSTER

Given the course dictionary:

```
course = {
    "name": "Intro to AI",
    "course_code": "CS101",
    "students": ["Alice", "Bob", "Charlie"]
}
```

- Add a new student, "Diana", to the students list.
- Add a new key-value pair: "topics": ["Machine Learning", "Search Algorithms", "Logic"].
- Write a loop that prints each topic from the new topics list.

INTRODUCTION TO SETS: THE UNIQUENESS ENFORCER

What if you have a list of user IDs, and you only want to know how many *unique* users there are? Or you want to see what items two lists have in common? This is where sets excel.

A **set** is an **unordered** collection of **unique** items, enclosed in curly braces {}.

- **Unordered:** Items have no index or position. You cannot slice them.
- **Unique:** Sets automatically discard duplicate values.

```
# Creating a set from a list automatically removes duplicates
numbers_list = [1, 2, 2, 3, 4, 4, 4, 5]
unique_numbers = set(numbers_list)
print(unique_numbers) # Output: {1, 2, 3, 4, 5}
```

```
# Creating a set directly
tags = {"python", "ai", "data", "python"} # The duplicate "python" is ignored
print(tags) # Output:
{'data', 'python', 'ai'} (order is not guaranteed)
```

```
# An empty set is created with set(), not {} which creates an empty dict
empty_s = set()
```

SET OPERATIONS

```
dev_team_A = {"Alice", "Bob", "Charlie", "David"}

dev_team_B = {"Charlie", "David", "Eve", "Frank"}
```

```
# Union (|): All unique members from both sets

all_devs = dev_team_A | dev_team_B

# -> {'Frank', 'David', 'Alice', 'Eve', 'Charlie', 'Bob'}
```

```
# Intersection (&): Members who are in BOTH sets

overlapping_devs = dev_team_A & dev_team_B

# -> {'Charlie', 'David'}
```

```
# Difference (-): Members in the first set but NOT in the second

only_in_A = dev_team_A - dev_team_B

# -> {'Alice', 'Bob'}
```

USE CASES: WHEN TO USE A SET VS. A LIST

- **Use a List when:** Order matters, you need to store duplicates, or you need to access items by a numerical index. (e.g., a to-do list, steps in a recipe).
- **Use a Set when:** You only care about the presence/absence of an item, you need to ensure all items are unique, or you need to perform membership tests or mathematical operations (e.g., unique tags on a blog post, finding common friends).

IN-CLASS EXERCISE: SKILL COMPARISON

```
job_A_skills = {"Python", "SQL", "Tableau", "Git"}
job_B_skills = {"Python", "Java", "AWS", "Git"}
```

- Find the skills required for **both** jobs.
- Find the skills required for job A but **not** job B.
- Create a set of all unique skills required for either job.

HANDS-ON LAB: THE CONTACT BOOK

- Create a new file contacts.py. You will build a simple contact manager using a dictionary.

PART 1: SETUP

- Create a dictionary called contact_book.
- The **keys** will be people's names (strings).
- The **values** will be their phone numbers (strings).
- Pre-populate it with 2-3 contacts.

```
contact_book = {
    "Alice": "555-1234",
    "Bob": "555-5678"
}
```

PART 2: VIEWING ALL CONTACTS

- Write code that iterates through the contact_book using .items().
- Print each contact's name and phone number in a user-friendly format, like Contact: Alice | Phone: 555-1234.

PART 3: ADDING OR UPDATING A CONTACT

- Prompt the user for a name.
- Prompt the user for a phone_number.
- Add the new name and number to the contact_book.
- Print a confirmation. Note that if the name already exists, this will update their number.

PART 4: SEARCHING FOR A CONTACT

- Prompt the user for a name to search for.
- Use the .get() method to look up the name in the contact_book.
- If the contact is found, print their name and number.
- If the contact is not found, print a message like "Sorry, 'John Doe' was not found in your contacts.".

PART 5: DELETING A CONTACT

- Prompt the user for a name to delete.
- **First, check if the name exists as a key in the dictionary.**
- If it exists, use del or .pop() to remove it and print a confirmation.
- If it does not exist, print a message indicating the contact was not found.

CHALLENGE / BONUS FEATURES:

- **Complex Data:** Modify your contact_book so that the value for each name is another **dictionary**. This inner dictionary should store multiple pieces of information, like {"phone": "555-1234", "email": "alice@example.com"}. Update all your functions (view, add, search) to work with this new structure.
- **Unique Groups:** Add a list of tags (e.g., "work", "family") to each contact's inner dictionary. Then, write a function that finds all unique tags used across all your contacts. (This will require iterating and using a **set**!).