Computational Thinking

Lecture 08: Data Structures: List, Tuple, and Dictionary

University of Engineering and Technology

VIETNAM NATIONAL UNIVERSITY HANOI

Outline

- Motivation and Real-life Scenarios
- List, Tuple, Dictionary
- Comparison of Lists, Tuples, and Dictionaries
- Practical Exercises
- Summary & Discussion

Motivation and Real-life Scenarios

Motivation and Real-life Scenarios (1)

Why Do We Need Data Structures?

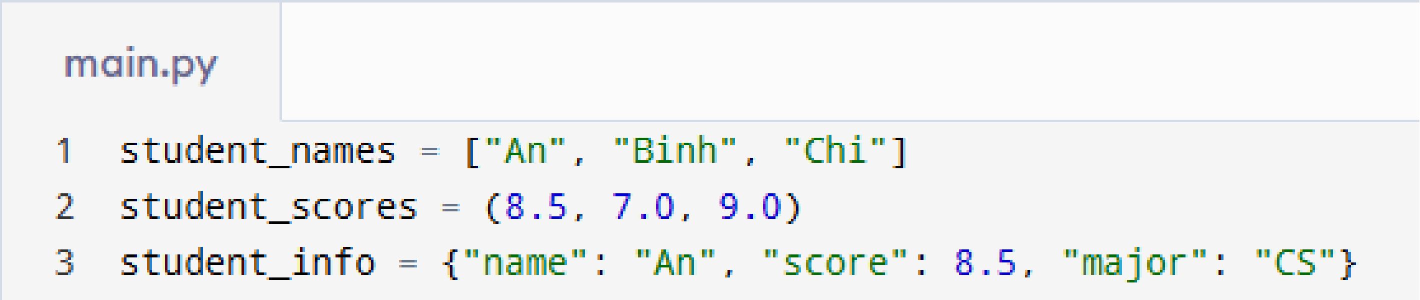
- How can we store and manage data efficiently?
- What happens if we try to keep everything in single variables?
- How do we represent a class of students, or an inventory of products, or a list of exam scores?

Motivation and Real-life Scenarios (2)

Organizing Data Like Organizing Items in Real Life

- List → like a shopping list (items in order)
- Tuple → like a fixed package (items that should not change)
- Dictionary → like a phonebook (key - value pairs)

Example Scenario: Student Management Example



```python
student_names = ["An", "Binh", "Chi"]
student_scores = (8.5, 7.0, 9.0)
student_info = {"name": "An", "score": 8.5, "major": "CS"}
```

- Lists → store multiple items
- Tuples → keep data fixed and ordered
- Dictionaries → store labeled information

Motivation and Real-life Scenarios (3)

Everyday Applications - Where We See These in Real Life

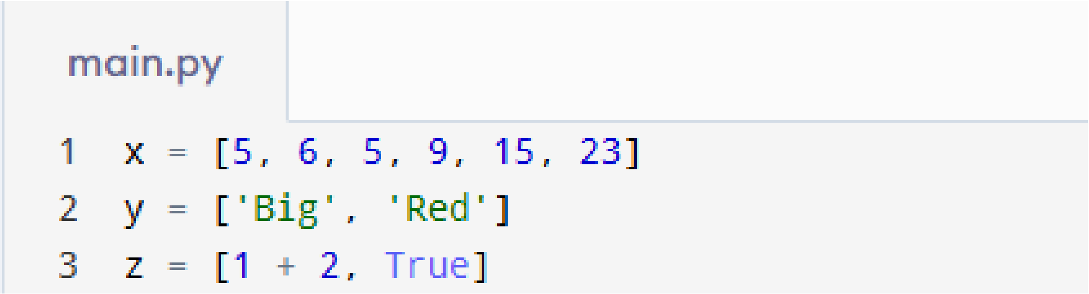
Examples:

- List: playlist of songs, queue of tasks
- Tuple: GPS coordinates (latitude, longitude)
- Dictionary: word meaning in a dictionary app, JSON data in APIs

List

What is a List?

- Key idea: A list is an ordered collection of items, enclosed in [ ]
- Example:



```
x = [5, 6, 5, 9, 15, 23]
y = ['Big', 'Red']
z = [1 + 2, True]
```

- Explain visually:
  - Elements have indexes (starting at 0)
  - A list can be empty ([ ]) or contain mixed types (though usually homogeneous)

Why Lists Are Useful

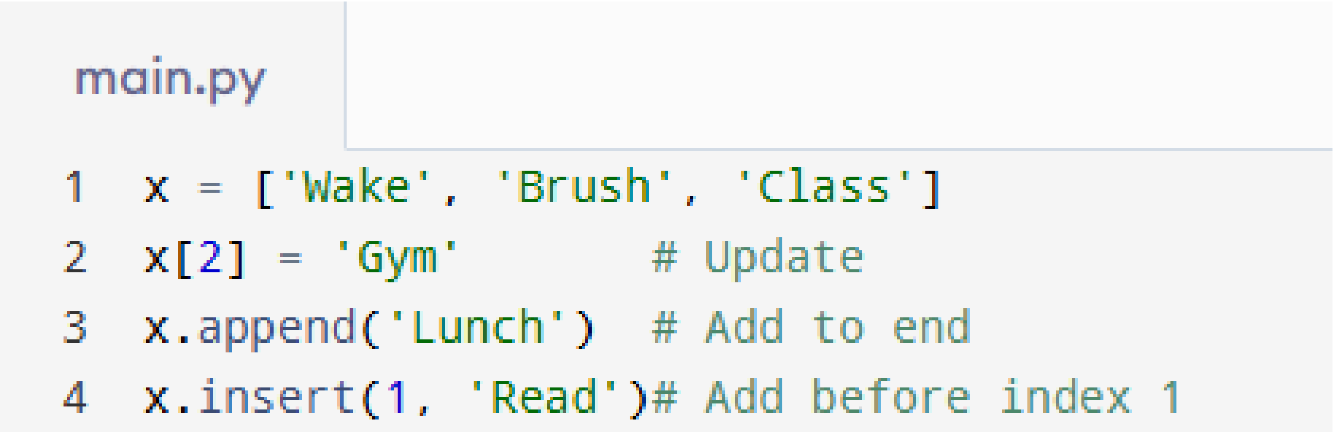
- Aggregate related data (e.g., student names, grades, tasks)
- Support flexible operations: add, remove, sort, iterate
- Foundation for more complex data types (matrix, dataset, record)

Basic Operations

Operation	**Example**	**Result**
Length	len(x)	6
Indexing	x[0]	5
Slicing	x[2:4]	[5, 9]
Concatenate	x + y	New list
Check item	'Red' in y	True

Slicing creates a new list (not modify original)

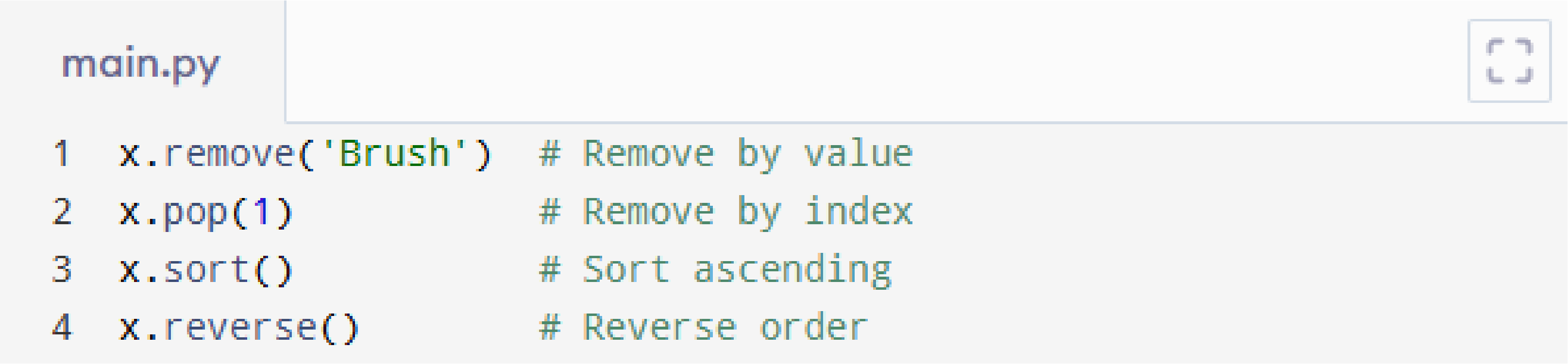
Updating & Adding Elements



```python
x = ['Wake', 'Brush', 'Class']
x[2] = 'Gym'       # Update
x.append('Lunch')  # Add to end
x.insert(1, 'Read')# Add before index 1
```

- Lists are mutable (can be modified)
- Operations happen in place (ID doesn’t change)

Removing & Reordering



main.py

```python
x.remove('Brush')  # Remove by value
x.pop(1)           # Remove by index
x.sort()           # Sort ascending
x.reverse()        # Reverse order
```

Visual tip: show list before/after operations

Mini Exercises

Create a list grades = [9.5, 8.0, 7.0, 10.0]

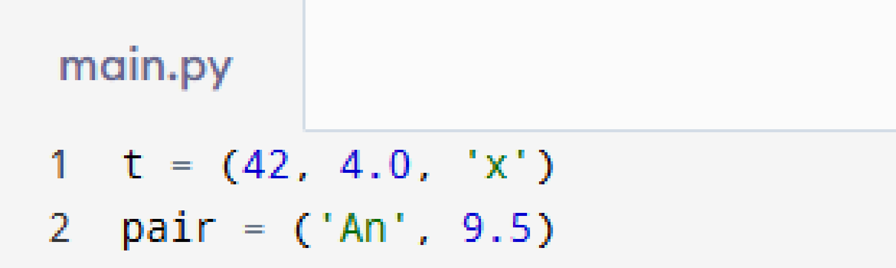
1. Print the first and last grade
2. Add a new grade 9.0
3. Sort the list descending

Tuple

What is a Tuple?

A tuple is an ordered, immutable collection of elements enclosed in parentheses ( )

- Example:



```python
t = (42, 4.0, 'x')
pair = ('An', 9.5)
```

  - Ordered like a list
  - Can store different data types
  - Cannot be changed after creation

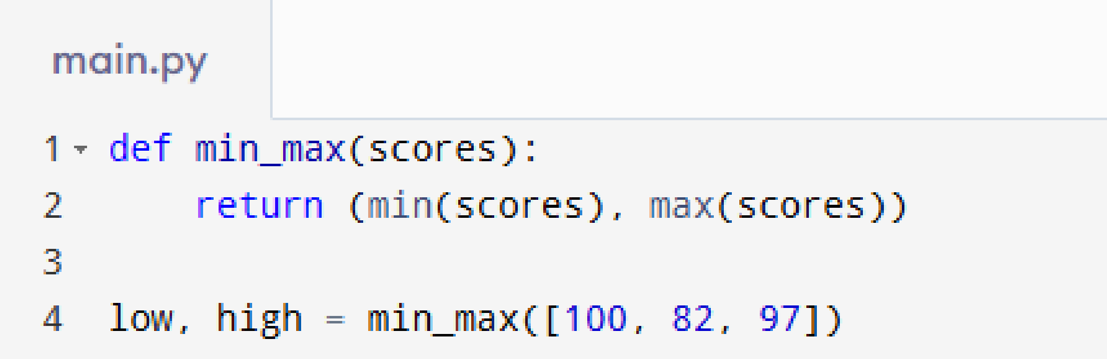
Why Tuple?

Sometimes, a list is too flexible - we know exactly how many values are needed

⇨ Use tuple when:

- The number of elements is fixed
- The data should not change

Example – Return multiple values from a function:



main.py

```python
def min_max(scores):
    return (min(scores), max(scores))

low, high = min_max([100, 82, 97])
```

Motivation: function returns **two values**, not a resizable list

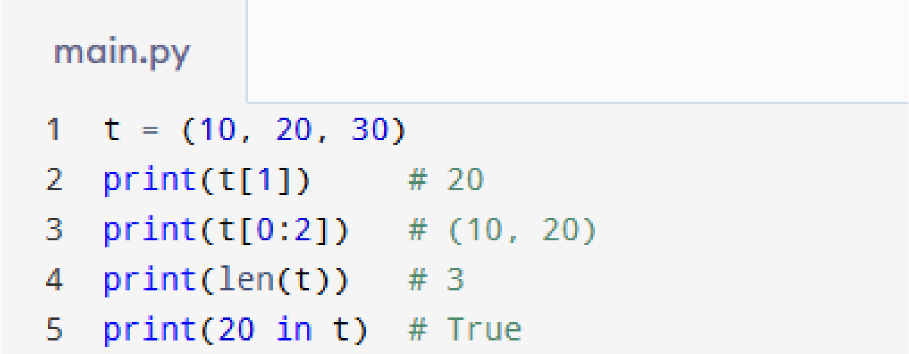
Tuple versus List

Feature	List	Tuple
Syntax	[ ]	( )
Mutability	Mutable	Immutable
Typical size	Variable	Fixed (2–3 items)
Use cases	Dynamic data	Static, grouped data
Example	[1, 2, 3]	(1, 2, 3)

Use a tuple for fixed-size data like coordinates, RGB colors, or paired values

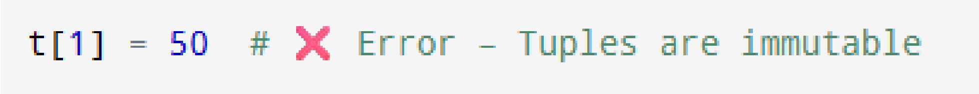
Tuple Operations

Tuples behave like sequences:



```python
t = (10, 20, 30)
print(t[1])      # 20
print(t[0:2])    # (10, 20)
print(len(t))    # 3
print(20 in t)   # True
```

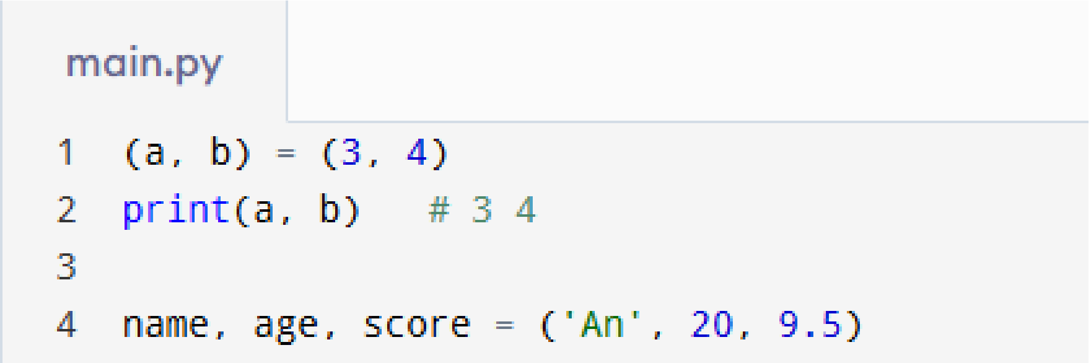
But remember:



```python
t[1] = 50  # ❌ Error – Tuples are immutable
```

Tuple Assignment (Unpacking)

Tuple unpacking makes code concise:



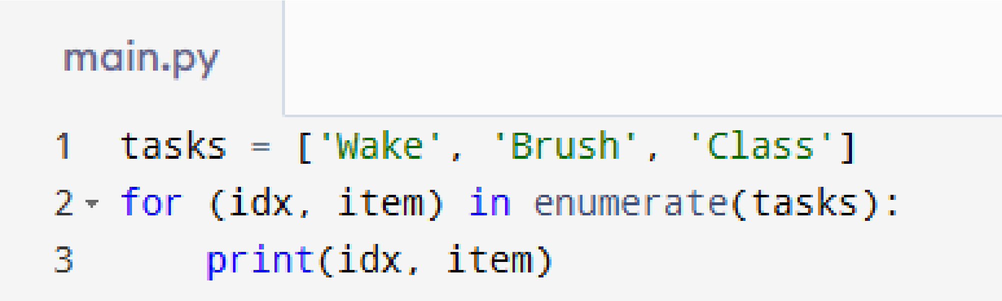
```python
(a, b) = (3, 4)
print(a, b)    # 3 4

name, age, score = ('An', 20, 9.5)
```

Use case: quickly assign multiple variables at once

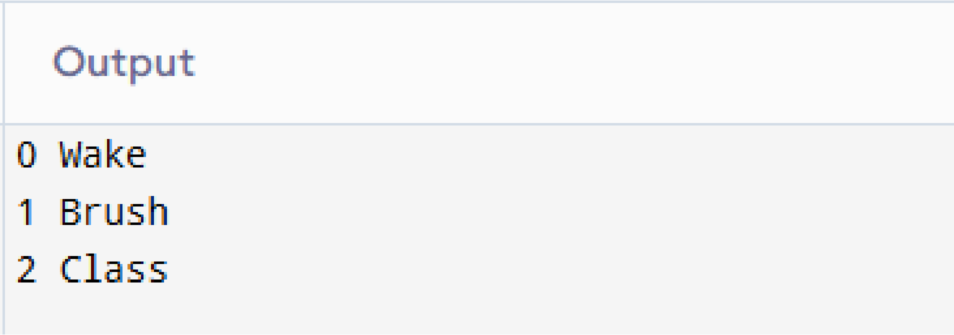
Enumerate Example

Tuples often appear in **loops** with enumerate():



```python
tasks = ['Wake', 'Brush', 'Class']
for (idx, item) in enumerate(tasks):
    print(idx, item)
```

Output:



```
0 Wake
1 Brush
2 Class
```

Mini Exercises

- Create a tuple point = (3, 4) and access its elements
- Convert list [1, 2, 3] → tuple using tuple()
- Write a function returning (sum, average) of a list.

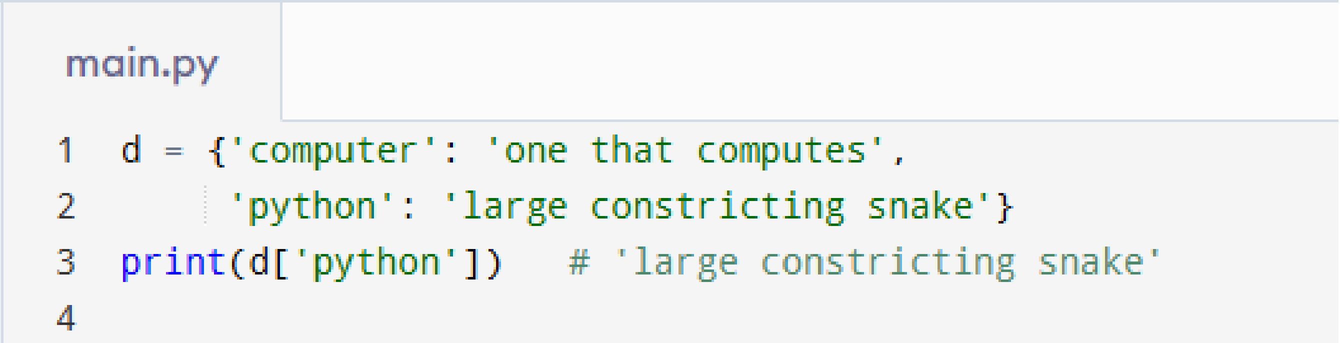
Dictionary

What is a Dictionary?

A **dictionary** is an **unordered collection of key–value pairs**.

- Each *key* maps to a *value*, and every key must be unique
- Syntax → {key: value, …}

Example:



```
d = {'computer': 'one that computes',
     'python': 'large constricting snake'}
print(d['python'])   # 'large constricting snake'

```

Analogy: like a **real dictionary** or a **contacts app** — you look up a *word/name* (key) to get its *definition/details*

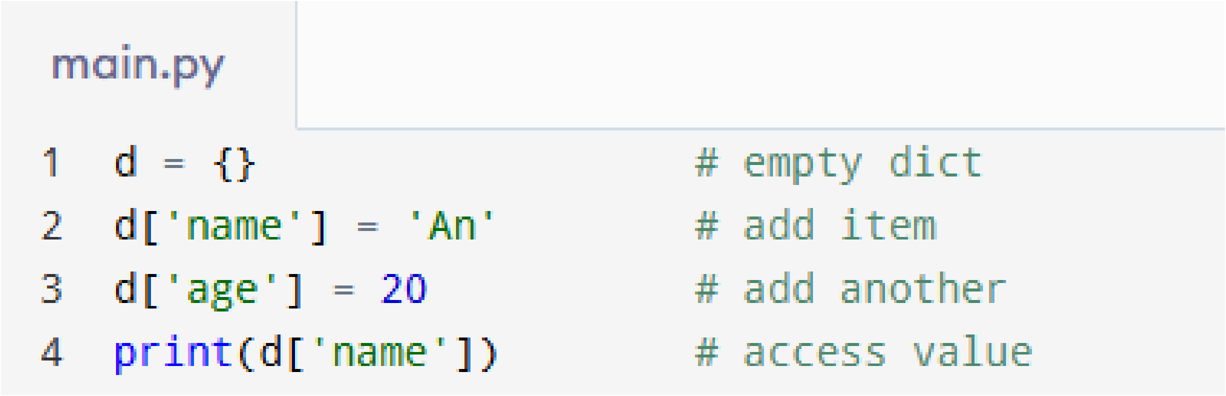
Why Dictionary?

- Let you label data with meaningful names instead of numeric indices
- Provide fast lookup by key
- Useful when you have structured or paired data.

Examples:

- Student → score
- Username → profile
- infoWord → definition

Creating and Accessing



main.py

```python
d = {}                  # empty dict
d['name'] = 'An'        # add item
d['age'] = 20           # add another
print(d['name'])        # access value
```

- A **KeyError** occurs if the key doesn’t exist (d['grade’])
- Keys must be **hashable** (immutable types → str, int, tuple).

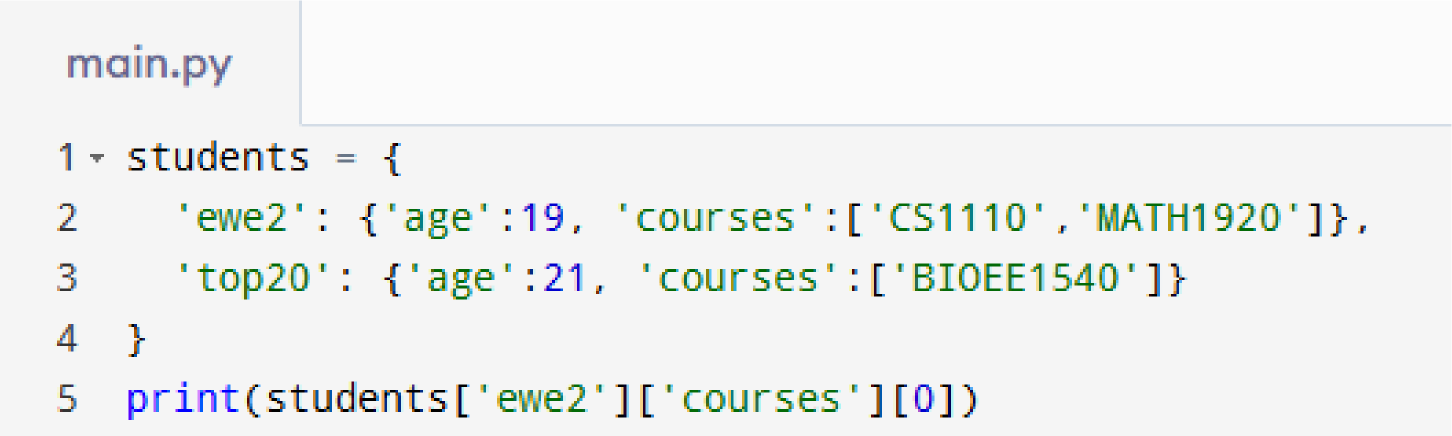
Basic Operations

Operation	Example	Result
Length	len(d)	number of items
Membership	'name' in d	True
Add/Update	d['city']='Hanoi'	add or replace
Delete	d.pop('age')	removes and returns value

Reminder: Assignment to a *new key* automatically **creates** it

Nested Structures

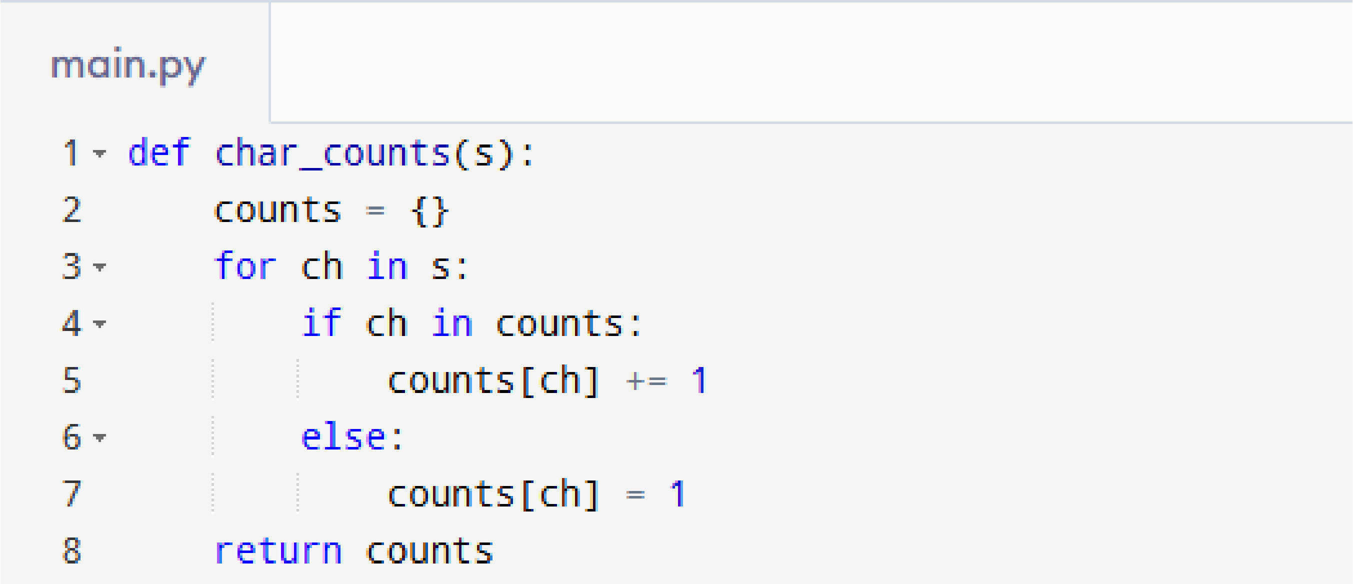
Dictionaries can store **lists** or **other dictionaries** as values



```python
students = {
    'ewe2': {'age':19, 'courses':['CS1110','MATH1920']},
    'top20': {'age':21, 'courses':['BIOEE1540']}
}
print(students['ewe2']['courses'][0])
```

Dictionary as Accumulator

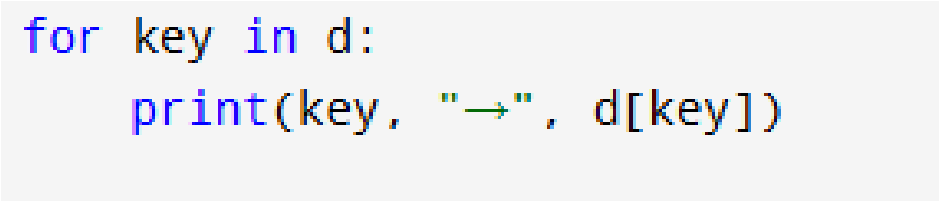
Count occurrences efficiently:



```python
def char_counts(s):
    counts = {}
    for ch in s:
        if ch in counts:
            counts[ch] += 1
        else:
            counts[ch] = 1
    return counts
```

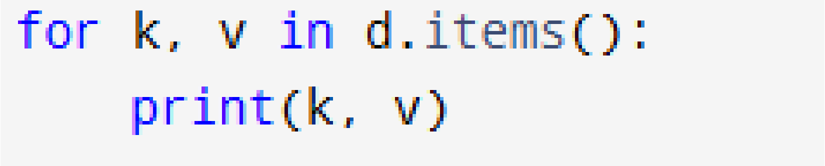
Maps each character → its count ({'a': 2, 'b': 2, '!’: 1})

Iteration with Dictionaries



```
for key in d:
    print(key, "→", d[key])
```

- The loop iterates over keys by default
- You can also use:



```
for k, v in d.items():
    print(k, v)
```

Dictionaries are *not sequences* → no numeric indexing

Mini Exercises

- Create profile = {'name' : 'An', 'major' : 'CS'} → add 'year' : 3
- Print each key - value pair
- Write a function that counts word frequency in a sentence.

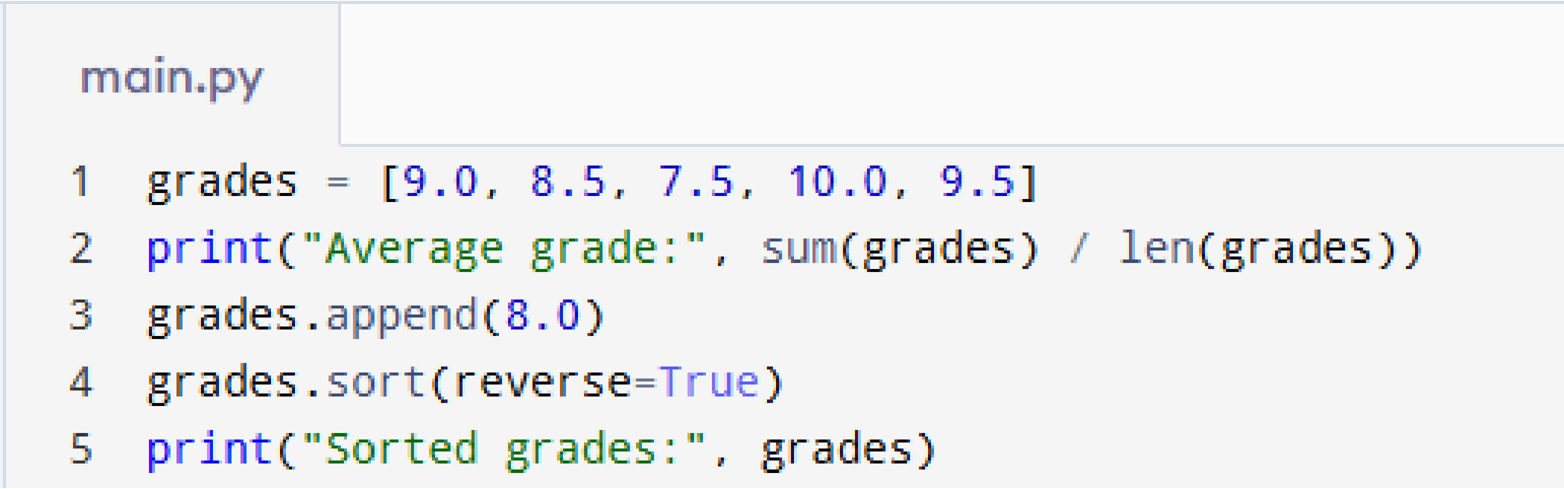
Comparison of List, Tuple, and Dictionary

Property	List	Tuple	Dictionary
Syntax	[ ]	( )	{key: value}
Order maintained?	✅ Yes	✅ Yes	⚠ Yes (insertion order, since Python 3.7)
Mutable?	✅ Yes	❌ No	✅ Yes
Accessed by	Index	Index	Key
Allow duplicates?	✅ Yes	✅ Yes	❌ Keys must be unique
Typical use	Variable-size, homogeneous data collections	Fixed-size, grouped records	Key–value mappings (lookup tables)
Example	[1, 2, 3]	(1, 2, 3)	{'a': 1, 'b': 2}

Practical Exercises

List Practice

Task: Create a program to store and analyze students' grades



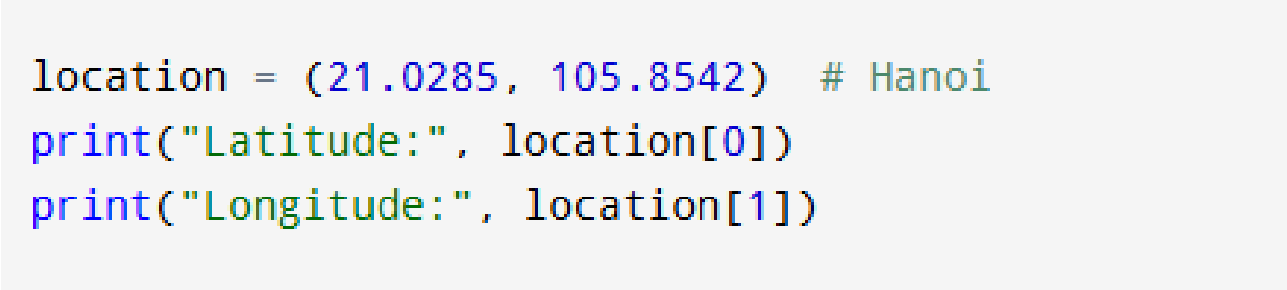
main.py

```python
grades = [9.0, 8.5, 7.5, 10.0, 9.5]
print("Average grade:", sum(grades) / len(grades))
grades.append(8.0)
grades.sort(reverse=True)
print("Sorted grades:", grades)
```

Question: What happens if we use grades[5] = 6.5? Why?

Tuple Practice

Task: Represent geographic coordinates



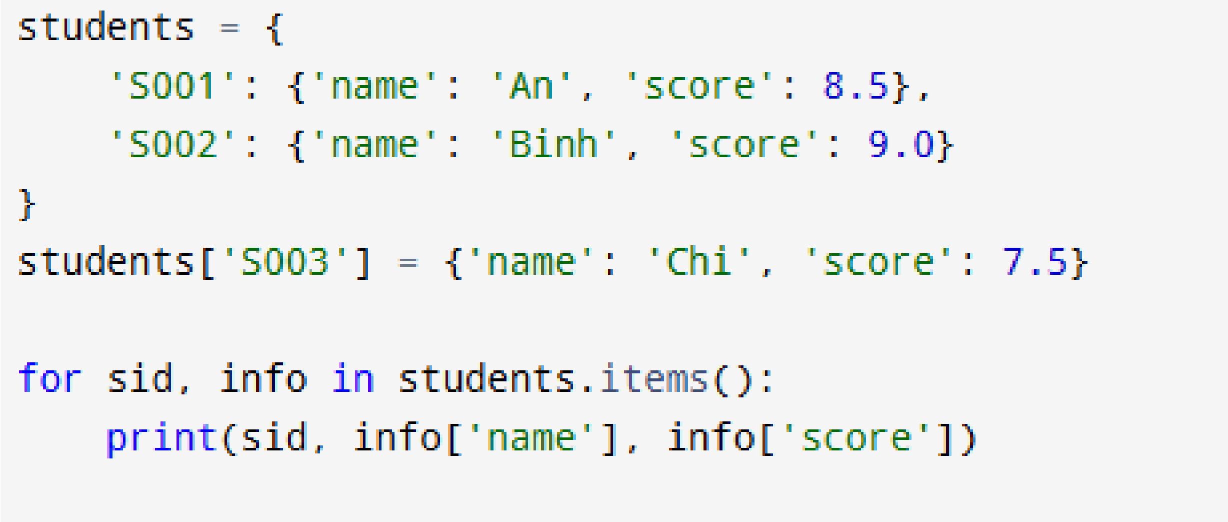
```
location = (21.0285, 105.8542)  # Hanoi
print("Latitude:", location[0])
print("Longitude:", location[1])
```

- Add a tuple for another city (e.g., Ho Chi Minh City)
- Print both coordinates using one print statement.

Why tuples are used for coordinates rather than lists?

Dictionary Practice

Task: Manage a mini student directory



```
students = {
    'S001': {'name': 'An', 'score': 8.5},
    'S002': {'name': 'Binh', 'score': 9.0}
}
students['S003'] = {'name': 'Chi', 'score': 7.5}

for sid, info in students.items():
    print(sid, info['name'], info['score'])
```

Question: How would you print the average score of all students?

Integrated Challenge

Mini Project: "Classroom Data Manager"

Write a short program that:

1. Uses a **list of dictionaries** to store student records
2. Allows the user to:
   - Add a new student
   - Search by student ID
   - Display all scores

Hint: Combine knowledge of **loops, lists, and dictionaries**.

Summary & Discussion

Computational Thinking Link

Each structure supports a key aspect of computational thinking:

- Decomposition: break data into smaller, manageable components
- Pattern recognition: identify similar data items → use lists/tuples
- Abstraction: represent real-world data (e.g., student info → dictionary)
- Algorithmic design: decide the right data type to make algorithms efficient

Common Mistakes

- Mixing up mutable vs. immutable types
- Using lists when dictionary lookups are faster
- Confusing parentheses ( ) and brackets [ ]