Python Dictionaries

1. What Is a Dictionary?

A **dictionary** is a Python data structure used to store data in **key-value pairs**.

Think of it like a real dictionary:

- You look for a **word**
- You get its **meaning**

In Python:

- You look for a **key**
- You get its **value**

```
student = {
    "name": "Hafsa",
    "age": 25,
    "course": "Python"
}
```

Here:

```
"name"   → "Hafsa"
"age"    → 25
"course" → "Python"
```

The left side is the **key** and the right side is the **value**.

---

2. Why Do We Need Dictionaries?

Lists are useful when data is mainly a sequence:

```
students = ["Hafsa", "Asim", "Ali"]
```

But what if every student has multiple pieces of information?

```
student = ["Hafsa", 25, "Python", "Beginner"]
```

Now you have to remember:

```
index 0 → name
index 1 → age
index 2 → course
index 3 → level
```

That becomes difficult to understand.

A dictionary makes the meaning explicit:

```
student = {
    "name": "Hafsa",
    "age": 25,
    "course": "Python",
    "level": "Beginner"
}
```

Now the data explains itself.

```
"name"   → name
"age"    → age
"course" → course
"level"  → level
```

Mental model

Think:

```
Dictionary
    ↓
Key → Value
```

For example:

```
"name" → "Hafsa"
```

---

3. Creating a Dictionary

Dictionaries are created using curly braces `{}`.

```
student = {
    "name": "Hafsa",
    "age": 25,
    "course": "Python"
}
```

The general structure is:

```
dictionary = {
    key: value,
    key: value,
    key: value
}
```

Each key-value pair is separated by a comma.

The key and value are separated by a colon `:`.

---

4. An Empty Dictionary

You can create an empty dictionary:

```
student = {}
```

You can add data later:

```
student["name"] = "Hafsa"
student["age"] = 25
```

Now:

```
print(student)
```

Output:

```
{'name': 'Hafsa', 'age': 25}
```

You can also use:

```
student = dict()
```

Both create an empty dictionary.

---

5. Dictionary Keys and Values

Consider:

```
student = {
    "name": "Hafsa",
    "age": 25,
    "is_student": True
}
```

There are three keys:

```
name
age
is_student
```

And three values:

```
Hafsa
25
True
```

The structure is:

```
Key              Value
-----------------------
"name"      →    "Hafsa"
"age"       →    25
"is_student"→    True
```

A dictionary can contain different data types as values.

```
profile = {
    "name": "Hafsa",
    "age": 25,
    "skills": ["Python", "JavaScript"],
    "active": True
}
```

---

6. Accessing Dictionary Values

Unlike lists, dictionaries are accessed using **keys**.

```
student = {
    "name": "Hafsa",
    "age": 25
}

print(student["name"])
```

Output:

```
Hafsa
```

And:

```
print(student["age"])
```

Output:

```
25
```

The basic syntax is:

```
dictionary[key]
```

---

7. Dictionary vs List Access

List:

```
students = ["Hafsa", "Asim", "Ali"]

print(students[0])
```

Dictionary:

```
student = {
    "name": "Hafsa",
    "age": 25
}

print(student["name"])
```

The difference is important:

```
List       → position/index
Dictionary → key
```

---

8. Keys Must Exist

Suppose:

```
student = {
    "name": "Hafsa",
    "age": 25
}
```

This works:

```
print(student["name"])
```

But:

```
print(student["email"])
```

causes:

```
KeyError
```

because `"email"` does not exist.

---

9. Checking Whether a Key Exists

You can use `in`.

```
student = {
    "name": "Hafsa",
    "age": 25
}

print("name" in student)
```

Output:

```
True
```

And:

```
print("email" in student)
```

Output:

```
False
```

This is useful when you are not sure whether data exists.

```
if "email" in student:
    print(student["email"])
```

---

10. Adding New Items

Dictionaries are mutable.

You can add a new key-value pair:

```
student = {
    "name": "Hafsa",
    "age": 25
}

student["course"] = "Python"
```

Now:

```
print(student)
```

Output:

```
{
    'name': 'Hafsa',
    'age': 25,
    'course': 'Python'
}
```

The syntax is:

```
dictionary[new_key] = value
```

---

11. Updating Existing Values

If the key already exists, assigning a new value updates it.

```
student = {
    "name": "Hafsa",
    "age": 25
}

student["age"] = 26
```

Now:

```
print(student["age"])
```

Output:

```
26
```

Python does not create another `"age"` entry.

It updates the existing value.

---

12. Adding vs Updating

These two operations use the same syntax.

```
student["course"] = "Python"
```

If `"course"` does not exist:

```
Add
```

If `"course"` already exists:

```
Update
```

For example:

```
student = {
    "name": "Hafsa"
}

student["course"] = "Python"
```

`course` is added.

Then:

```
student["course"] = "JavaScript"
```

`course` is updated.

---

13. Dictionary Length

Use `len()` to find the number of key-value pairs.

```
student = {
    "name": "Hafsa",
    "age": 25,
    "course": "Python"
}

print(len(student))
```

Output:

```
3
```

There are three key-value pairs.

---

14. Dictionary Values Can Be Different Types

A dictionary does not require every value to have the same type.

```
user = {
    "name": "Hafsa",
    "age": 25,
    "skills": ["Python", "JavaScript"],
    "is_active": True,
    "experience": 2.5
}
```

Here:

```
name       → string
age        → integer
skills     → list
is_active  → boolean
experience → float
```

This makes dictionaries extremely useful for representing real-world objects.

---

15. Dictionaries Can Store Lists

For example:

```
developer = {
    "name": "Hafsa",
    "skills": ["Python", "JavaScript", "React"]
}
```

You can access the list:

```
print(developer["skills"])
```

Output:

```
['Python', 'JavaScript', 'React']
```

And then access an individual list item:

```
print(developer["skills"][0])
```

Output:

```
Python
```

The first access gets the dictionary value.

The second access gets the list item.

---

16. Dictionaries Can Store Other Dictionaries

A dictionary can contain another dictionary.

```
user = {
    "name": "Hafsa",
    "profile": {
        "age": 25,
```

```
        "country": "Pakistan"
    }
}
```

You can access the nested value:

```
print(user["profile"]["age"])
```

Output:

```
25
```

Nested dictionaries will be explored in detail in a separate topic.

---

17. Looping Through a Dictionary

You can loop through a dictionary.

```
student = {
    "name": "Hafsa",
    "age": 25,
    "course": "Python"
}

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

Output:

```
name
age
course
```

By default, looping over a dictionary gives you its **keys**.

---

18. Looping Through Keys

You can also explicitly use `.keys()`.

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

Output:

```
name
age
```

```
course
```

`.keys()` gives access to the dictionary's keys.

---

19. Looping Through Values

Use `.values()` when you only need values.

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

Output:

```
Hafsa
25
Python
```

This is useful when the keys are not important.

---

20. Looping Through Keys and Values

Often you need both.

Use `.items()`:

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

Output:

```
name Hafsa
age 25
course Python
```

You can make the output clearer:

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

Output:

```
name: Hafsa
age: 25
course: Python
```

Important mental model

```
.keys()   → keys
.values() → values
.items()  → keys + values
```

---

21. Removing Items with `pop()`

Use `pop()` to remove a specific key.

```
student = {
    "name": "Hafsa",
    "age": 25,
    "course": "Python"
}

student.pop("age")
```

Now:

```
print(student)
```

Output:

```
{'name': 'Hafsa', 'course': 'Python'}
```

`pop()` also returns the removed value.

```
age = student.pop("age")
```

---

22. Removing the Last Item with `popitem()`

`popitem()` removes and returns the last inserted key-value pair.

```
student = {
    "name": "Hafsa",
    "age": 25,
    "course": "Python"
}

item = student.popitem()

print(item)
```

Output:

```
('course', 'Python')
```

---

23. Removing Everything with `clear()`

Use `clear()` to remove all items.

```
student = {
    "name": "Hafsa",
    "age": 25
}

student.clear()

print(student)
```

Output:

```
{}
```

The dictionary still exists, but it is empty.

---

24. Deleting a Dictionary Item with `del`

You can also use `del`.

```
student = {
    "name": "Hafsa",
    "age": 25
}

del student["age"]
```

Now:

```
print(student)
```

Output:

```
{'name': 'Hafsa'}
```

You can also delete the entire dictionary:

```
del student
```

After this, `student` no longer exists.

---

25. `get()` for Safe Access

Instead of:

```
student["email"]
```

you can use:

```
student.get("email")
```

If the key does not exist, `get()` returns:

```
None
```

instead of raising a `KeyError`.

Example:

```
student = {
    "name": "Hafsa"
}

print(student.get("email"))
```

Output:

```
None
```

---

26. `get()` With a Default Value

You can provide a default value.

```
email = student.get("email", "Email not provided")

print(email)
```

Output:

```
Email not provided
```

This is useful when working with data where some fields may be missing.

---

27. `update()`

The `update()` method can add or update multiple items.

```
student = {
    "name": "Hafsa",
    "age": 25
```

```
}

student.update({
    "course": "Python",
    "level": "Beginner"
})
```

Now:

```
print(student)
```

Output:

```
{
    'name': 'Hafsa',
    'age': 25,
    'course': 'Python',
    'level': 'Beginner'
}
```

It can also update existing keys:

```
student.update({
    "age": 26
})
```

---

28. Dictionary Keys Must Be Hashable

Dictionary keys need to be types that Python can hash.

Common valid keys include:

```
{
    "name": "Hafsa",
    1: "One",
    2: "Two",
    (1, 2): "Coordinates"
}
```

Strings, integers, and tuples can be used as keys when their contents are hashable.

A list cannot be used as a dictionary key:

```
{
    ["a", "b"]: "value"
}
```

This causes a `TypeError`.

For beginners, the most common choice is simply:

```
string keys
```

such as:

```
"user_id"
"name"
"email"
"course"
```

---

29. Duplicate Keys

Dictionary keys should be unique.

Consider:

```
student = {
    "name": "Hafsa",
    "name": "Asim"
}
```

Python keeps the last value:

```
print(student)
```

Output:

```
{'name': 'Asim'}
```

So avoid duplicate keys when you want to preserve multiple pieces of data.

---

30. Dictionary Insertion Order

Modern Python dictionaries preserve insertion order.

```
student = {}

student["name"] = "Hafsa"
student["age"] = 25
student["course"] = "Python"
```

When you iterate:

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

the keys appear in the order they were inserted.

But remember:

A dictionary should primarily be thought of as key-value data, not as a replacement for a list.

---

31. Dictionary and List Comparison

List

```
student = ["Hafsa", 25, "Python"]
```

Access:

```
student[0]
```

Meaning depends on position.

Dictionary

```
student = {
    "name": "Hafsa",
    "age": 25,
    "course": "Python"
}
```

Access:

```
student["name"]
```

Meaning is explicit.

Mental model

```
List
Position → Value

Dictionary
Key → Value
```

---

32. When Should You Use a Dictionary?

Use a dictionary when data has **named properties**.

Good examples:

```
student = {
    "name": "Hafsa",
    "age": 25
}
```

```
product = {
    "name": "Laptop",
    "price": 150000,
    "stock": 10
}
```

```
user = {
    "username": "hafsa",
    "email": "user@example.com"
}
```

```
settings = {
    "theme": "dark",
    "notifications": True
}
```

---

33. Real World Example: Student Record

```
student = {
    "name": "Hafsa",
    "age": 25,
    "course": "Python",
    "level": "Beginner"
}

print(student["name"])
print(student["course"])
```

Output:

```
Hafsa
Python
```

This is much easier to understand than:

```
student = ["Hafsa", 25, "Python", "Beginner"]
```

---

34. Real World Example: Product

Imagine an online store.

```
product = {
    "name": "Laptop",
    "price": 150000,
    "brand": "Example",
    "in_stock": True
}
```

You can check:

```
if product["in_stock"]:
    print("Product is available")
```

You can update:

```
product["price"] = 145000
```

And add:

```
product["category"] = "Electronics"
```

---

35. Real World Example: User Permissions

A software application might store permissions like this:

```
permissions = {
    "read": True,
    "write": True,
    "delete": False
}
```

Then:

```
if permissions["delete"]:
    print("Delete allowed")
else:
    print("Delete not allowed")
```

Output:

```
Delete not allowed
```

This pattern appears frequently in real applications.

---

36. Real World Example: haas.dev Resource

A resource could be represented as:

```
resource = {
    "title": "Python Dictionaries",
    "category": "Python",
    "level": "Beginner",
    "free": True
}
```

You can access:

```
print(resource["title"])
print(resource["category"])
```

Output:

```
Python Dictionaries
Python
```

This is similar to how applications represent structured information about content.

---

37. Dictionary Data From User Input

You can build a dictionary from user input:

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))

student = {
    "name": name,
    "age": age
}

print(student)
```

If the user enters:

```
Hafsa
25
```

the dictionary becomes:

```
{
    "name": "Hafsa",
```

```
    "age": 25
}
```

---

38. Checking User Data

Dictionaries become especially useful with conditions.

```
user = {
    "name": "Hafsa",
    "is_admin": True
}

if user["is_admin"]:
    print("Admin dashboard")
else:
    print("User dashboard")
```

The dictionary stores the data.

The condition uses that data to make a decision.

---

39. Counting With Dictionaries

Dictionaries are commonly used for counting.

Suppose:

```
letters = ["a", "b", "a", "c", "a", "b"]
```

We want:

```
a → 3
b → 2
c → 1
```

We can build:

```
counts = {}

for letter in letters:
    if letter in counts:
        counts[letter] += 1
    else:
        counts[letter] = 1

print(counts)
```

Output:

```
{'a': 3, 'b': 2, 'c': 1}
```

This is one of the most important dictionary patterns.

Mental model

```
If key exists:
    increase value

If key does not exist:
    create key with 1
```

---

40. Dictionary as a Lookup Table

Dictionaries are excellent for quickly finding a value based on a key.

```
prices = {
    "apple": 200,
    "banana": 150,
    "mango": 300
}
```

Now:

```
print(prices["mango"])
```

Output:

```
300
```

Instead of searching through a list manually, the key directly identifies the value.

---

41. A Simple Grade System

```
grades = {
    "Hafsa": 92,
    "Asim": 85,
    "Ali": 78
}

print(grades["Hafsa"])
```

Output:

```
92
```

You can also update a grade:

```
grades["Ali"] = 82
```

Or add another student:

```
grades["Sara"] = 90
```

---

42. Dictionary Inside a List

Real applications often combine data structures.

For example:

```
students = [
    {
        "name": "Hafsa",
        "age": 25
    },
    {
        "name": "Asim",
        "age": 26
    }
]
```

Now:

```
print(students[0]["name"])
```

Output:

```
Hafsa
```

This pattern is extremely common when working with structured application data.

---

43. Common Mistake: Using Indexes

Beginners sometimes try:

```
student = {
    "name": "Hafsa",
    "age": 25
}
```

```
print(student[0])
```

This is incorrect because dictionaries are accessed using keys.

Use:

```
print(student["name"])
```

---

44. Common Mistake: Forgetting Quotes Around String Keys

This is incorrect:

```
student = {
    name: "Hafsa"
}
```

unless `name` is already a variable.

Normally write:

```
student = {
    "name": "Hafsa"
}
```

---

45. Common Mistake: Confusing `[]` and `{}`

List:

```
students = ["Hafsa", "Asim"]
```

Dictionary:

```
student = {
    "name": "Hafsa",
    "age": 25
}
```

Remember:

```
[] → list
{} → dictionary
```

An empty `{}` creates a dictionary.

An empty list is:

```
[]
```

---

46. Common Mistake: Accessing a Missing Key

This can fail:

```
email = student["email"]
```

if `"email"` does not exist.

Use:

```
email = student.get("email")
```

when missing data is possible.

---

47. Common Mistake: Modifying While Iterating

Avoid directly changing the dictionary's size while looping over it.

Problematic pattern:

```
for key in student:
    del student[key]
```

This can raise:

```
RuntimeError
```

A safer approach is to first create a separate list of keys:

```
for key in list(student.keys()):
    del student[key]
```

Or simply use:

```
student.clear()
```

when you want to remove everything.

---

48. Problem Solving With Dictionaries

When you see a problem, ask:

Step 1: What is the thing I am storing?

For example:

```
student
product
user
resource
```

Step 2: What properties does it have?

For a student:

```
name
age
course
level
```

Step 3: Give every property a key.

```
student = {
    "name": "Hafsa",
    "age": 25,
    "course": "Python",
    "level": "Beginner"
}
```

Step 4: Decide what operation you need.

```
Read       → dictionary[key]
Add        → dictionary[key] = value
Update     → dictionary[key] = new_value
Safe read  → dictionary.get(key)
Delete     → pop() / del
Loop       → keys() / values() / items()
```

This turns a vague problem into a structured process.

---

49. Dictionary Methods Cheat Sheet

Method	Purpose
`get()`	Safely retrieve a value

`keys()`	Get all keys
`values()`	Get all values
`items()`	Get key-value pairs
`update()`	Add/update multiple items
`pop()`	Remove a specific key
`popitem()`	Remove the last item
`clear()`	Remove all items

Example:

```
student.get("name")
student.keys()
student.values()
student.items()
student.update({"level": "Intermediate"})
student.pop("age")
student.popitem()
student.clear()
```

---

50. Dictionary Operations Cheat Sheet

Create

```
student = {
    "name": "Hafsa"
}
```

Access

```
student["name"]
```

Safe access

```
student.get("name")
```

Add

```
student["age"] = 25
```

Update

```
student["age"] = 26
```

Check key

```
"name" in student
```

Delete

```
del student["age"]
```

Remove and return

```
student.pop("age")
```

Loop keys

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

Loop values

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

Loop both

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

---

51. Mini Project: Student Information System

Create a simple student record.

```
student = {
    "name": "Hafsa",
    "age": 25,
    "course": "Python",
    "marks": 92
}

print("Student:", student["name"])
print("Age:", student["age"])
print("Course:", student["course"])
print("Marks:", student["marks"])
```

Now add a result:

```
student["result"] = "Pass"
```

Update the marks:

```
student["marks"] = 95
```

Finally:

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

The goal is to practice:

```
Creating
Accessing
Adding
Updating
Looping
```

---

52. 5 Minute Challenge

Create a dictionary called `developer`.

It should contain:

```
name
experience
```

```
skills
is_available
```

Example structure:

```
developer = {
    "name": "Hafsa",
    "experience": 2,
    "skills": ["Python", "JavaScript"],
    "is_available": True
}
```

Then:

1. Print the developer's name.
2. Print the skills.
3. Add a `location` key.
4. Update `experience`.
5. Check whether `email` exists.
6. Use `.get()` to safely retrieve `email`.
7. Loop through all key-value pairs.

---

53. Practice Exercises

Exercise 1: Create a Profile

Create a dictionary containing:

```
name
age
city
profession
```

Print every value.

Exercise 2: Update Data

Create:

```
product = {
    "name": "Keyboard",
    "price": 5000
}
```

Change the price to `4500`.

Exercise 3: Add Data

Create a dictionary containing a student's name.

Then add:

```
age
course
marks
```

Exercise 4: Safe Access

Create a dictionary without an email.

Use `.get()` to print:

```
Email not available
```

when the key is missing.

Exercise 5: Loop Through Data

Create a dictionary:

```
skills = {
    "Python": "Intermediate",
    "JavaScript": "Advanced",
    "React": "Beginner"
}
```

Print:

```
Python → Intermediate
JavaScript → Advanced
React → Beginner
```

Exercise 6: Count Items

Given:

```
items = ["apple", "banana", "apple", "orange", "banana", "apple"]
```

Create a dictionary containing the number of times each item appears.

Expected:

```
{
    "apple": 3,
```

```
    "banana": 2,
    "orange": 1
}
```

Exercise 7: Student Grades

Create:

```
grades = {
    "Hafsa": 92,
    "Asim": 85,
    "Ali": 78
}
```

Add another student.

Update Ali's grade.

Print all students and their grades.

Exercise 8: Resource Data

Create a dictionary representing a haas.dev resource:

```
title
category
level
free
```

Print the information in a readable format.

---

54. Mini Quiz

Question 1

What is the main purpose of a dictionary?

A. Store only numbers

B. Store data using key-value pairs

C. Store only strings

D. Repeat code

Answer: B

Question 2

How do you access the `"name"` value?

```
student = {
    "name": "Hafsa"
}
```

A.

```
student[0]
```

B.

```
student["name"]
```

C.

```
student.name
```

D.

```
student(name)
```

Answer: B

Question 3

Which method safely retrieves a value when a key may not exist?

A. `find()`

B. `search()`

C. `get()`

D. `read()`

Answer: C

---

55. Interview Questions

Beginner

1. What is a dictionary in Python?
2. What is a key-value pair?
3. How do you create a dictionary?
4. How do you access a dictionary value?
5. How do you add a new item?
6. How do you update a value?
7. How do you delete an item?
8. What does `len()` return for a dictionary?

9. How do you check whether a key exists?
10. What is the difference between a list and a dictionary?

Intermediate

11. What does `get()` do?
12. What is the difference between `get()` and `[]` access?
13. What does `.keys()` return?
14. What does `.values()` return?
15. What does `.items()` return?
16. What does `update()` do?
17. What is the difference between `pop()` and `popitem()`?
18. What does `clear()` do?
19. Can dictionary values have different data types?
20. Can a dictionary contain another dictionary or list?

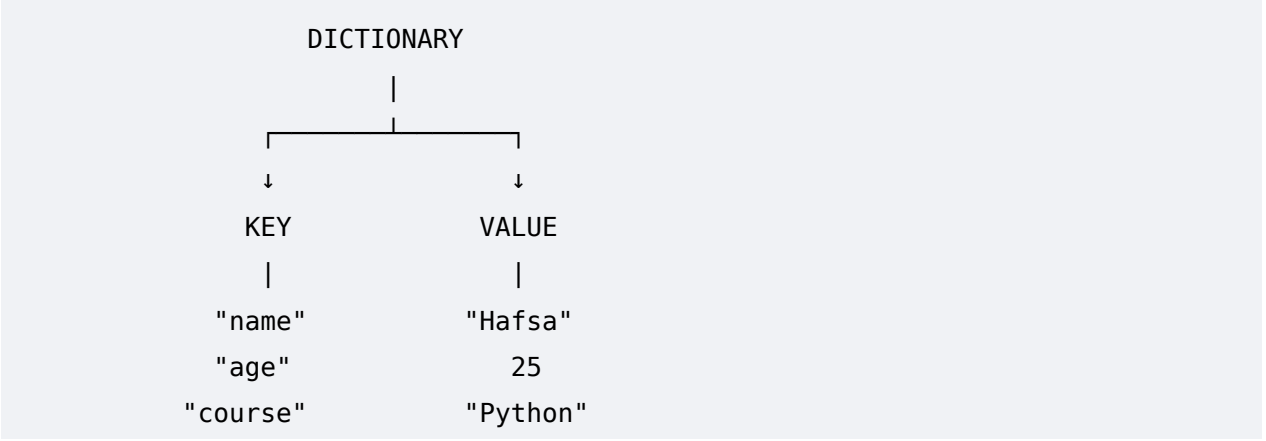
Conceptual

21. Why are dictionaries useful for structured data?
22. Why can duplicate keys cause unexpected results?
23. Why should you use `.get()` when data may be missing?
24. When would you choose a dictionary instead of a list?
25. How can dictionaries be used for counting frequencies?

---

56. Final Mental Model

Remember dictionaries using this simple picture:



The most important syntax:

```
student["name"]
```

The most important operations:

```
# Create
student = {"name": "Hafsa"}

# Read
student["name"]

# Safe read
student.get("name")

# Add
student["age"] = 25

# Update
student["age"] = 26

# Check
"name" in student

# Delete
student.pop("age")

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

The core idea is:

```
LIST
position → value

DICTIONARY
key → value
```

Once you understand that difference, dictionaries become much easier to use.

Key Takeaways

- A dictionary stores data as **key-value pairs**.
- Dictionaries use `{}`.
- Keys are used to access values.
- Dictionary values can contain different data types.

- Dictionaries are mutable.
- You can add and update values using the same syntax.
- `get()` provides safer access when a key may be missing.
- `keys()` gives keys.
- `values()` gives values.
- `items()` gives key-value pairs.
- `pop()` removes a specific item.
- `update()` can add or update multiple items.
- Dictionaries are excellent for structured real-world data.
- Dictionaries are commonly used for lookup tables, records, configuration, permissions, and counting.

Next concept: Dictionary Indexing and Methods will go deeper into accessing, retrieving, modifying, and working with dictionary data efficiently.

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app

https://dev-roast-app.vercel.app/resources