Python Nested Data Structures

1. What Are Nested Data Structures?

A **nested data structure** is a data structure that contains another data structure inside it.

For example, a list can contain another list:

```
students = [
    ["Hafsa", 25],
    ["Asim", 26]
]
```

A dictionary can contain a list:

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

A dictionary can contain another dictionary:

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

You can even combine several structures:

```
students = [
    {
        "name": "Hafsa",
        "skills": ["Python", "JavaScript"]
    },
    {
        "name": "Asim",
        "skills": ["Flutter", "Dart"]
    }
]
```

This is called **nested data**.

---

2. Why Do We Need Nested Data Structures?

Real-world data is rarely completely flat.

Consider a student:

```
Name: Hafsa
Age: 25
Country: Pakistan
Skills:
    Python
    JavaScript
    React
```

A simple dictionary can store the basic information:

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

But skills contain multiple values.

We can use a list inside the dictionary:

```
student = {
    "name": "Hafsa",
    "age": 25,
    "country": "Pakistan",
    "skills": ["Python", "JavaScript", "React"]
}
```

Now the structure matches the real-world data.

Mental model

Think of nested structures like **containers inside containers**:

```
Dictionary
│
├── name → "Hafsa"
├── age → 25
└── skills
      │
      ├── "Python"
```

```
       ├── "JavaScript"
       └── "React"
```

---

3. Types of Nested Data Structures

Python allows many combinations.

List inside a list

```
numbers = [
    [1, 2, 3],
    [4, 5, 6]
]
```

Dictionary inside a dictionary

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

List inside a dictionary

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

Dictionary inside a list

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

List and dictionary combined

```
courses = {
    "Python": ["Beginner", "Intermediate"],
    "JavaScript": ["Beginner", "Advanced"]
}
```

These combinations appear constantly in real programs.

---

4. Nested Lists

A list can contain other lists.

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

Think of it like a table:

```
1  2  3
4  5  6
7  8  9
```

Each inner list represents a row.

---

5. Accessing a Nested List

To access an item inside a nested list, use multiple indexes.

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

print(matrix[0])
```

Output:

```
[1, 2, 3]
```

Now:

```
print(matrix[0][1])
```

Output:

```
2
```

Why?

```
matrix[0]
→ first inner list
```

```
→ [1, 2, 3]

matrix[0][1]
→ second item of that list
→ 2
```

Mental model

Read the indexes from left to right:

```
matrix[0][1]
       ↑  ↑
       │  └── item inside the row
       └───── row
```

---

6. More Nested List Access

```
matrix = [
    [10, 20, 30],
    [40, 50, 60]
]
```

```
print(matrix[1][0])
```

Output:

```
40
```

```
print(matrix[1][2])
```

Output:

```
60
```

The first index selects the inner list.

The second index selects the item inside that list.

---

7. Updating Nested Lists

Nested items can be changed.

```
matrix = [
    [10, 20, 30],
    [40, 50, 60]
]
```

```
matrix[1][2] = 100
```

Now:

```
print(matrix)
```

Output:

```
[[10, 20, 30], [40, 50, 100]]
```

The same indexing technique is used to modify the nested value.

---

8. Looping Through Nested Lists

Suppose:

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

You can use nested loops:

```
for row in matrix:
    for number in row:
        print(number)
```

Output:

```
1
2
3
4
5
6
7
8
9
```

The outer loop handles each inner list.

The inner loop handles each item.

---

9. Printing a Matrix

If you want each row on its own line:

```
for row in matrix:
    print(row)
```

Output:

```
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
```

Or:

```
for row in matrix:
    for number in row:
        print(number, end=" ")
    print()
```

Output:

```
1 2 3
4 5 6
7 8 9
```

---

10. Dictionary Inside a List

This is one of the most important real-world structures.

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

The outer structure is a list.

Each item inside the list is a dictionary.

Think:

```
List
|
├── Dictionary
|     ├── name
|     └── age
|
└── Dictionary
      ├── name
      └── age
```

---

11. Accessing a Dictionary Inside a List

First access the list item:

```
students[0]
```

This gives:

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

Then access the dictionary key:

```
students[0]["name"]
```

Output:

```
Hafsa
```

Another example:

```
students[1]["age"]
```

Output:

```
26
```

Mental model

```
students[0]["name"]
        |      |
        |      └── dictionary key
        └─────────── list index
```

---

12. Updating a Dictionary Inside a List

You can update nested data:

```
students[0]["age"] = 26
```

Now the first student's age is `26`.

You can also add a new key:

```
students[0]["course"] = "Python"
```

Now:

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

contains:

```
{
    "name": "Hafsa",
    "age": 26,
    "course": "Python"
}
```

---

13. Looping Through a List of Dictionaries

This pattern is extremely common.

```
students = [
    {"name": "Hafsa", "age": 25},
    {"name": "Asim", "age": 26},
    {"name": "Ali", "age": 24}
]

for student in students:
    print(student["name"])
```

Output:

```
Hafsa
Asim
Ali
```

Each loop iteration gives you one dictionary.

---

14. Processing Multiple Fields

```
for student in students:
    print(f"Name: {student['name']}")
    print(f"Age: {student['age']}")
    print()
```

Output:

```
Name: Hafsa
Age: 25

Name: Asim
Age: 26

Name: Ali
Age: 24
```

This pattern is common when processing API data or database records.

---

15. Dictionary Inside a Dictionary

A dictionary can contain another dictionary.

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

The outer dictionary contains:

```
name
profile
```

The `"profile"` value is another dictionary.

---

16. Accessing Nested Dictionaries

To access the age:

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

Output:

```
25
```

Break it down:

```
user
 ↓
["profile"]
 ↓
["age"]
 ↓
25
```

The syntax:

```
dictionary["outer_key"]["inner_key"]
```

---

17. Updating Nested Dictionaries

You can update the inner value:

```
user["profile"]["age"] = 26
```

You can also add a new value:

```
user["profile"]["city"] = "Gujranwala"
```

Now:

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

contains:

```
{
    "age": 26,
    "country": "Pakistan",
    "city": "Gujranwala"
}
```

---

18. Dictionary With Lists

A dictionary can contain a list.

```
developer = {
    "name": "Hafsa",
    "skills": [
```

```
        "Python",
        "JavaScript",
        "React"
    ]
}
```

Access the entire list:

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

Access one skill:

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

Output:

```
Python
```

---

19. Updating a List Inside a Dictionary

You can change a list item:

```
developer["skills"][1] = "TypeScript"
```

Now:

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

Output:

```
['Python', 'TypeScript', 'React']
```

You can also add a new skill:

```
developer["skills"].append("Node.js")
```

Now:

```
['Python', 'TypeScript', 'React', 'Node.js']
```

---

20. Looping Through a List Inside a Dictionary

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

```
for skill in developer["skills"]:
    print(skill)
```

Output:

```
Python
JavaScript
React
```

The dictionary gives you the list.

The loop processes the list.

---

21. Dictionary With Multiple Nested Structures

Real applications often combine everything.

```
developer = {
    "name": "Hafsa",
    "profile": {
        "age": 25,
        "country": "Pakistan"
    },
    "skills": [
        "Python",
        "JavaScript",
        "React"
    ]
}
```

Here:

```
developer
│
├── name → string
│
├── profile → dictionary
│      ├── age
│      └── country
│
└── skills → list
       ├── Python
       ├── JavaScript
       └── React
```

This is a realistic nested structure.

---

22. Accessing Different Levels

Get the name:

```
developer["name"]
```

Get the country:

```
developer["profile"]["country"]
```

Get the first skill:

```
developer["skills"][0]
```

The type of the nested value determines what you do next.

```
Dictionary
→ use a key

List
→ use an index
```

---

23. The Most Important Rule

When working with nested data:

Look at the current container before deciding how to access the next level.

For example:

```
developer["profile"]
```

returns a dictionary.

So you use another key:

```
developer["profile"]["age"]
```

But:

```
developer["skills"]
```

returns a list.

So you use an index:

```
developer["skills"][0]
```

Mental model

```
If current value is a dictionary
→ use ["key"]

If current value is a list
→ use [index]
```

---

24. List of Dictionaries With Lists

You can combine three levels.

```
developers = [
    {
        "name": "Hafsa",
        "skills": ["Python", "React"]
    },
    {
        "name": "Asim",
        "skills": ["Dart", "Flutter"]
    }
]
```

Get Hafsa's first skill:

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

Output:

```
Python
```

Break it down:

```
developers
    ↓
[0]
    ↓
first dictionary
    ↓
["skills"]
    ↓
skills list
    ↓
[0]
```

```
    ↓
"Python"
```

---

25. Nested Data From APIs

APIs commonly return nested data.

For example:

```
response = {
    "user": {
        "name": "Hafsa",
        "profile": {
            "city": "Gujranwala"
        }
    }
}
```

To access the city:

```
city = response["user"]["profile"]["city"]
```

This type of nested structure appears frequently when working with web APIs.

---

26. Nested Data and JSON

JSON uses a structure very similar to Python dictionaries and lists.

Example JSON-like data:

```
data = {
    "product": {
        "name": "Laptop",
        "price": 150000,
        "features": [
            "16GB RAM",
            "512GB SSD"
        ]
    }
}
```

Access the product name:

```
data["product"]["name"]
```

Access the first feature:

```
data["product"]["features"][0]
```

Understanding nested Python structures makes working with JSON much easier.

---

27. Nested Data and `get()`

Nested dictionaries can sometimes contain missing keys.

For example:

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

This could fail:

```
user["profile"]["city"]
```

because `"profile"` doesn't exist.

A safer approach is:

```
profile = user.get("profile", {})
city = profile.get("city", "Unknown")
```

Now the program can continue even when the data is incomplete.

---

28. A Safer Nested Access Pattern

Suppose:

```
user = {
    "profile": {
        "name": "Hafsa"
    }
}
```

You can write:

```
city = user.get("profile", {}).get("city", "Unknown")
```

If `"city"` is missing, the result is:

```
Unknown
```

The pattern is:

```
dictionary.get("outer_key", {}).get("inner_key", default)
```

This is useful when working with optional API data.

---

29. Nested Loops

Nested structures often require nested loops.

Example:

```
classes = [
    ["Hafsa", "Asim"],
    ["Ali", "Sara"]
]
```

Loop through every student:

```
for classroom in classes:
    for student in classroom:
        print(student)
```

Output:

```
Hafsa
Asim
Ali
Sara
```

The outer loop processes the groups.

The inner loop processes items inside each group.

---

30. Nested List of Dictionaries

Consider:

```
courses = [
    {
        "name": "Python",
        "students": ["Hafsa", "Asim"]
    },
    {
        "name": "JavaScript",
        "students": ["Ali", "Sara"]
    }
]
```

You can loop:

```
for course in courses:
    print(course["name"])

    for student in course["students"]:
        print(student)
```

Output:

```
Python
Hafsa
Asim
JavaScript
Ali
Sara
```

This is a realistic structure for representing courses and their students.

---

31. Filtering Nested Data

Suppose:

```
students = [
    {"name": "Hafsa", "marks": 92},
    {"name": "Asim", "marks": 75},
    {"name": "Ali", "marks": 88}
]
```

Find students with marks above 80:

```
for student in students:
    if student["marks"] >= 80:
        print(student["name"])
```

Output:

```
Hafsa
Ali
```

The pattern is:

```
list
 ↓
dictionary
 ↓
key
```

```
 ↓
value
 ↓
condition
```

---

32. Modifying Nested Data in a Loop

You can update nested dictionaries.

```
students = [
    {"name": "Hafsa", "marks": 92},
    {"name": "Asim", "marks": 75}
]

for student in students:
    if student["marks"] >= 80:
        student["result"] = "Pass"
```

Now Hafsa has:

```
{
    "name": "Hafsa",
    "marks": 92,
    "result": "Pass"
}
```

while Asim does not receive the new key.

---

33. Nested Data for haas.dev Resources

Imagine haas.dev resources are represented like this:

```
resources = [
    {
        "title": "Python Dictionaries",
        "category": "Python",
        "topics": [
            "Keys",
            "Values",
            "Methods"
        ]
    },
    {
```

```
        "title": "Python Sets",
        "category": "Python",
        "topics": [
            "Unique Values",
            "Set Operations"
        ]
    }
]
```

Get the first resource title:

```
print(resources[0]["title"])
```

Get its first topic:

```
print(resources[0]["topics"][0])
```

Output:

```
Keys
```

---

34. Searching Nested Resources

Suppose you want to find resources in the Python category:

```
for resource in resources:
    if resource["category"] == "Python":
        print(resource["title"])
```

Output:

```
Python Dictionaries
Python Sets
```

You can also search topics:

```
for resource in resources:
    if "Methods" in resource["topics"]:
        print(resource["title"])
```

Output:

```
Python Dictionaries
```

This is a practical example of processing nested application data.

---

35. Adding Data to Nested Structures

Suppose:

```
resource = {
    "title": "Python Dictionaries",
    "topics": ["Keys", "Values"]
}
```

Add a topic:

```
resource["topics"].append("Methods")
```

Now:

```
print(resource["topics"])
```

Output:

```
['Keys', 'Values', 'Methods']
```

The dictionary itself did not need to be replaced.

We accessed the nested list and modified it.

---

36. Removing Nested Data

You can remove a nested list item:

```
resource["topics"].remove("Values")
```

Or by index:

```
resource["topics"].pop(0)
```

You can remove an entire nested key:

```
resource.pop("topics")
```

The operation depends on which level you want to modify.

---

37. Nested Data and Slicing

Nested lists can also be sliced.

```
developer = {
    "skills": [
        "Python",
```

```
        "JavaScript",
        "React",
        "Node.js"
    ]
}
```

Get the first two skills:

```
print(developer["skills"][:2])
```

Output:

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

The first operation accesses the dictionary key.

The second operation slices the list.

---

38. Nested Data and `len()`

You can use `len()` at different levels.

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

Number of dictionary fields:

```
print(len(developer))
```

Output:

```
2
```

Number of skills:

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

Output:

```
3
```

Always identify which level you are measuring.

---

39. Common Mistake: Wrong Index

Given:

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

This:

```
students["name"]
```

is incorrect.

Why?

Because `students` is a **list**.

You need an index first:

```
students[0]["name"]
```

---

40. Common Mistake: Using an Index on a Dictionary

Given:

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

This is incorrect:

```
student[0]
```

because `student` is a dictionary.

Use:

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

---

41. Common Mistake: Forgetting the Structure

Consider:

```
data = {
    "students": [
        {"name": "Hafsa"},
        {"name": "Asim"}
    ]
}
```

To access Asim:

```
data["students"][1]["name"]
```

Read it step by step:

```
data
 ↓
"students"
 ↓
list
 ↓
[1]
 ↓
dictionary
 ↓
"name"
 ↓
"Asim"
```

Never try to guess the syntax.

Follow the structure level by level.

---

42. Common Mistake: Too Many Levels

Nested structures can become difficult to understand when they become excessively deep.

For example:

```
data["user"]["profile"]["account"]["settings"]["preferences"]["theme"]
```

This works, but deeply nested data can become hard to maintain.

If your data regularly needs many levels, consider whether the structure can be simplified.

Good data modeling is not about making data as nested as possible.

It is about making the structure match the problem clearly.

---

43. A Practical Debugging Technique

When nested access doesn't work, inspect one level at a time.

Instead of:

```
print(data["user"]["profile"]["settings"]["theme"])
```

start with:

```
print(data)
```

Then:

```
print(data["user"])
```

Then:

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

Then:

```
print(data["user"]["profile"]["settings"])
```

Finally:

```
print(data["user"]["profile"]["settings"]["theme"])
```

This makes it easier to identify where the problem is.

---

44. Understanding the Data Type

Use `type()` while debugging:

```
print(type(data))
```

Then:

```
print(type(data["user"]))
```

And:

```
print(type(data["user"]["skills"]))
```

You might discover:

```
dict
dict
list
```

This immediately tells you:

```
dict → use keys
list → use indexes
```

---

45. Nested Data Problem-Solving Framework

When you receive nested data, follow these steps.

Step 1: Identify the outer structure

Is it:

```
list?
dictionary?
```

Step 2: Access one level

```
data["users"]
```

or:

```
data[0]
```

Step 3: Check the resulting type

```
type(data["users"])
```

Step 4: Choose the next operation

If it is a dictionary:

```
["key"]
```

If it is a list:

```
[index]
```

Step 5: Repeat

Continue until you reach the value you need.

Mental model

```
Identify
   ↓
Access
   ↓
Inspect
   ↓
Access again
   ↓
Repeat
   ↓
Final value
```

---

46. Real World Structure: Online Store

An online store might have:

```
store = {
    "name": "Tech Store",
    "products": [
        {
            "name": "Laptop",
            "price": 150000,
            "features": ["16GB RAM", "512GB SSD"]
        },
        {
            "name": "Phone",
            "price": 80000,
            "features": ["128GB Storage", "5G"]
        }
    ]
}
```

Get the first product:

```
store["products"][0]
```

Get its name:

```
store["products"][0]["name"]
```

Get its first feature:

```
store["products"][0]["features"][0]
```

The same access logic keeps repeating.

---

47. Real World Structure: User Account

```
account = {
    "username": "hafsa",
    "profile": {
        "name": "Hafsa",
        "location": {
            "city": "Gujranwala",
            "country": "Pakistan"
        }
```

```
    },
    "skills": [
        "Python",
        "JavaScript"
    ]
}
```

Get city:

```
account["profile"]["location"]["city"]
```

Get second skill:

```
account["skills"][1]
```

This is a good example of why understanding the type at every level matters.

---

48. Real World Structure: Application Configuration

```
config = {
    "app": {
        "name": "haas.dev",
        "version": "1.0"
    },
    "database": {
        "host": "localhost",
        "port": 5432
    },
    "features": [
        "search",
        "resources",
        "quizzes"
    ]
}
```

Get the app name:

```
config["app"]["name"]
```

Get the database port:

```
config["database"]["port"]
```

Get the first feature:

```
config["features"][0]
```

Nested structures allow related settings to stay grouped together.

---

49. Mini Project: Student Management System

Create:

```
students = [
    {
        "name": "Hafsa",
        "age": 25,
        "subjects": {
            "Python": 92,
            "JavaScript": 88
        }
    },
    {
        "name": "Asim",
        "age": 26,
        "subjects": {
            "Python": 85,
            "JavaScript": 79
        }
    }
]
```

Step 1: Print the first student's name

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

Step 2: Print Hafsa's Python marks

```
print(students[0]["subjects"]["Python"])
```

Step 3: Print every student's name

```
for student in students:
    print(student["name"])
```

Step 4: Print every student's Python marks

```
for student in students:
    print(student["name"], student["subjects"]["Python"])
```

Step 5: Add a new subject

```
students[0]["subjects"]["React"] = 90
```

Step 6: Find students with Python marks above 80

```
for student in students:
    if student["subjects"]["Python"] > 80:
        print(student["name"])
```

This single project combines:

```
list
dictionary
nested dictionary
loop
condition
indexing
updating
```

---

50. 5 Minute Challenge

Create a nested structure for a developer:

```
developer = {
    "name": "Hafsa",
    "profile": {
        "role": "Software Engineer",
        "experience": 2
    },
    "skills": [
        "Python",
        "JavaScript",
        "React"
    ]
}
```

Now:

1. Print the developer's name.
2. Print the role.
3. Print the experience.
4. Print the first skill.

5. Add `"TypeScript"` to the skills list.
6. Change the role.
7. Add a `"city"` field inside `profile`.
8. Loop through all skills.
9. Print the entire profile.
10. Check the type of `developer["skills"]`.

---

51. Practice Exercises

Exercise 1: Nested List

Create:

```
numbers = [
    [10, 20, 30],
    [40, 50, 60],
    [70, 80, 90]
]
```

Print:

- `20`
- `60`
- `70`
- `90`

Exercise 2: List of Dictionaries

Create:

```
students = [
    {"name": "Hafsa", "marks": 92},
    {"name": "Asim", "marks": 85},
    {"name": "Ali", "marks": 78}
]
```

Print every student's name and marks.

Exercise 3: Dictionary With a List

Create:

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

Add `"React"` to the skills list.

Then print all skills.

Exercise 4: Nested Dictionary

Create:

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

Print the age and country.

Then add a city.

Exercise 5: Nested Loop

Create:

```
groups = [
    ["Hafsa", "Asim"],
    ["Ali", "Sara"]
]
```

Use nested loops to print every name.

Exercise 6: Filtering

Given:

```
students = [
    {"name": "Hafsa", "marks": 92},
    {"name": "Asim", "marks": 75},
    {"name": "Ali", "marks": 88}
]
```

Print only students whose marks are greater than or equal to `80`.

Exercise 7: Nested Resource Data

Create:

```
resources = [
    {
        "title": "Python Dictionaries",
        "category": "Python",
        "topics": ["Keys", "Values", "Methods"]
    },
    {
        "title": "Python Sets",
        "category": "Python",
        "topics": ["Unique Values", "Set Operations"]
    }
]
```

Then:

1. Print every resource title.
2. Print the first topic of each resource.
3. Find resources containing `"Methods"`.

---

52. Mini Quiz

Question 1

What is nested data?

A. Data stored only in lists

B. A data structure containing another data structure

C. Data stored only in variables

D. Data without keys

Answer: B

Question 2

Given:

```
students = [
    {"name": "Hafsa"},
```

```
    {"name": "Asim"}
]
```

How do you access `"Asim"`?

A.

```
students["name"][1]
```

B.

```
students[1]["name"]
```

C.

```
students["Asim"]
```

D.

```
students[0]["name"]
```

Answer: B

Question 3

Given:

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

How do you access `"React"`?

A.

```
developer["skills"]["React"]
```

B.

```
developer[0][1]
```

C.

```
developer["skills"][1]
```

D.

```
developer[1]
```

Answer: C

---

53. Interview Questions

Beginner

1. What is a nested data structure?
2. Why are nested structures useful?
3. Can a list contain another list?
4. Can a dictionary contain another dictionary?
5. How do you access a list inside a dictionary?
6. How do you access a dictionary inside a list?
7. What is the difference between a dictionary key and a list index?
8. How do nested loops work with nested lists?

Intermediate

9. How do you access a value inside a dictionary nested inside a list?
10. How do you modify a nested dictionary value?
11. How do you modify an item inside a list stored in a dictionary?
12. How can `.get()` help with nested dictionaries?
13. How would you process a list of dictionaries?
14. How would you filter a list of dictionaries?
15. How would you search for a value inside nested data?
16. How do APIs commonly use nested structures?
17. What is the relationship between Python dictionaries/lists and JSON?
18. How can `type()` help debug nested structures?

Problem Solving

19. How would you represent multiple students and their subjects?
20. How would you represent products with multiple features?
21. How would you search nested resource data?
22. How would you update a value several levels deep?
23. How would you safely access optional nested information?

---

54. Nested Data Cheat Sheet

List inside list

```
data[0][1]
```

Dictionary inside list

```
data[0]["name"]
```

List inside dictionary

```
data["skills"][0]
```

Dictionary inside dictionary

```
data["profile"]["age"]
```

List inside dictionary inside list

```
data[0]["skills"][0]
```

Dictionary inside dictionary inside dictionary

```
data["user"]["profile"]["location"]
```

Safe nested dictionary access

```
data.get("profile", {}).get("city", "Unknown")
```

Loop through list of dictionaries

```
for item in data:
    print(item["name"])
```

Loop through nested lists

```
for group in data:
    for item in group:
        print(item)
```

---

55. Final Mental Model

Nested data becomes much easier when you stop seeing it as one complicated object.

See it as a series of containers:

```
OUTER CONTAINER
      ↓
ACCESS ONE LEVEL
      ↓
LOOK AT THE NEW CONTAINER
```

```
      ↓
CHOOSE THE CORRECT ACCESS METHOD
      ↓
REPEAT
```

Remember:

```
Dictionary
    ↓
use a key

List
    ↓
use an index
```

For example:

```
data[0]["profile"]["skills"][1]
```

Read it from left to right:

```
data
 ↓
[0]
 ↓
dictionary
 ↓
["profile"]
 ↓
dictionary
 ↓
["skills"]
 ↓
list
 ↓
[1]
 ↓
final value
```

That is the core skill behind working with nested data.

Key Takeaways

- Nested data structures contain other data structures.

- Lists can contain lists, dictionaries, or other structures.
- Dictionaries can contain lists, dictionaries, or other structures.
- A **list is accessed by index**.
- A **dictionary is accessed by key**.
- Real-world applications frequently use combinations such as:
  - list of dictionaries
  - dictionary containing lists
  - dictionary containing dictionaries
- Nested loops are useful for processing nested lists.
- APIs and JSON commonly use nested structures.
- Always inspect the current data type before deciding how to access the next level.
- When nested access becomes confusing, break it into individual steps.
- `type()` is useful for debugging nested structures.
- `.get()` can make nested dictionary access safer when data may be missing.
- Good nested structures represent relationships between pieces of data clearly.

The most important rule:

```
At every level ask:

"What type of data am I looking at?"

dict → key
list → index
```

Once you understand that rule, even deeply nested data becomes manageable.

Visit haas.dev for more resources and guides.

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

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