Nested Dictionaries & Real Data Patterns

AiCE Software Engineering Foundations — Python Programming

This closing lesson on dictionaries — and on the entire Python Programming section — covers nesting dictionaries inside one another, the powerful list-of-dictionaries pattern used throughout real software, and building dictionaries dynamically. It closes with a complete intelligence report that brings every data structure from this section together.

1. Nested Dictionaries

A dictionary's value can itself be a dictionary, letting you represent complex, hierarchical data — exactly the kind found in real databases, APIs, and configuration files:

```
# AiCE Multi-Campus Directory
campuses = {
    "Lagos": {
        "country" : "Nigeria",
        "director": "Chukwuemeka Obi",
        "students": 420,
        "courses" : ["Python", "JavaScript", "SQL"]
    },
    "Accra": {
        "country" : "Ghana",
        "director": "Ama Owusu",
        "students": 310,
        "courses" : ["Python", "Data Science", "DevOps"]
    },
}

# Accessing nested values
print(campuses["Lagos"]["director"])
print(campuses["Accra"]["students"])
print(campuses["Lagos"]["courses"][1])    # Second course
```

Nested dictionary access uses chained square brackets: campuses["Lagos"]["director"] first retrieves the Lagos dictionary, then retrieves the director from within it.

campuses["Lagos"]["courses"][1] goes three levels deep — dictionary, inner dictionary, then list index.

Looping through a nested structure and accumulating a total works naturally with .items():

```
total_students = 0

for campus, info in campuses.items():
    print(f"{campus}, {info["country"]}: {info["students"]}
students")
    total_students += info["students"]

print(f"Total across all campuses: {total_students}")
```

> **This Is What JSON Looks Like**
> *This nested structure is exactly how JSON data is organized — the format used by almost every web API in existence. Once you start working with real APIs later in this program, everything will look exactly like what you just built here.*

2. Lists of Dictionaries

One of the most powerful and commonly used data patterns in Python is a list of dictionaries — each dictionary represents one record, and the list holds all records. This is the Python equivalent of a database table:

```
students = [
    {"name": "Amara Diallo", "country": "Senegal", "gpa": 3.8,
"paid": True},
    {"name": "Kofi Mensah",  "country": "Ghana",   "gpa": 2.9,
"paid": False},
    {"name": "Ngozi Adeyemi","country": "Nigeria", "gpa": 3.5,
"paid": True},
]

for student in students:
    name = student["name"]
    paid = "Yes" if student["paid"] else "No"
```

```
    print(f"{name:<18} | Paid: {paid}")
```

> **A Quick Preview — List Comprehensions & Lambda**
> *You may occasionally see compact expressions like [s for s in students if s["paid"]] (a list comprehension — builds a new list of only the paid students) or max(students, key=lambda s: s["gpa"]) (finds the record with the highest GPA). These are more advanced techniques covered properly later in this program. For now, just read them conceptually: a list of dictionaries, filtered or searched by key. The pattern — not the shorthand syntax — is what matters right now.*

3. Building Dictionaries Dynamically

Just as lists are built with .append(), dictionaries are commonly built one key-value pair at a time — especially from user input:

```
# Build a student record from user input
student = {}    # Start with an empty dictionary

student["name"]    = input("Full name: ").strip().title()
student["country"] = input("Country  : ").strip().title()
student["age"]     = int(input("Age      : "))
student["gpa"]     = float(input("GPA      : "))

# Compute derived fields
student["eligible"] = student["gpa"] >= 2.0 and student["age"]
>= 16

for key, value in student.items():
    print(f"{key.title():<12}: {value}")
```

The program builds the dictionary field by field, computes a derived value (eligible) from the fields already collected, and displays the entire record with .items().

4. Dictionary Methods — Complete Reference

Method	What It Does	Returns
dict[key]	Access value by key — raises KeyError if missing	Value
dict.get(key, default)	Access value — returns default (or None) if missing	Value or default
dict[key] = value	Add a new key or update an existing one	Nothing
dict.pop(key)	Remove and return the value for a key	Value
del dict[key]	Remove a key-value pair	Nothing
dict.clear()	Remove all key-value pairs	Nothing
dict.keys() / .values() / .items()	Return all keys / values / key-value pairs	View objects
key in dict	Check if a key exists	True or False
len(dict)	Return the number of key-value pairs	Integer

5. Lists vs. Dictionaries — When to Use Which

Situation	Use	Reason
An ordered sequence of similar items	List	Order matters; access by position
One record with multiple named fields	Dictionary	Access by meaningful key name
Multiple records of the same structure	List of dicts	Each dict is a record; list holds them all
Looking up a value by a known name	Dictionary	Key lookup is instant regardless of size

6. Bringing It All Together — The AiCE Country Intelligence Report

A capstone combining nested dictionaries, loops, and dictionary operations into a genuinely useful report:

```
ecosystems = {
    "Nigeria": {"capital": "Abuja", "developers": 85000,
"unicorns": 3, "aice_students": 420},
    "Kenya"  : {"capital": "Nairobi", "developers": 62000,
"unicorns": 2, "aice_students": 385},
    "Ghana"  : {"capital": "Accra", "developers": 41000,
"unicorns": 1, "aice_students": 310},
}

total_devs     = sum(e["developers"] for e in
ecosystems.values())
total_students = sum(e["aice_students"] for e in
ecosystems.values())
top_country    = max(ecosystems, key=lambda c:
ecosystems[c]["developers"])

for country, data in ecosystems.items():
    print(f"{country} — Capital: {data["capital"]},
Developers: {data["developers"]:,}")

print(f"Total developers : {total_devs:,}")
print(f"Total students   : {total_students:,}")
print(f"Leading ecosystem: {top_country}")
```

This report pulls together nested dictionaries, loop-based iteration, .values(), .items(), and sum() and max() with key expressions previewed earlier in this lesson — all in one coherent program producing genuinely useful output about the African tech landscape.

Key Takeaways

Nested dictionaries — dictionaries whose values are themselves dictionaries — represent hierarchical data exactly the way JSON and real APIs do. A list of dictionaries is the Python equivalent of a database table: one dictionary per record, one list holding them all.

Dictionaries, like lists, are typically built dynamically, one key at a time, rather than written out in full. Between variables, strings, numbers, conditionals, loops, lists, and dictionaries, you now have every foundational building block of Python — the same six or seven concepts that every professional Python program, no matter how large, is ultimately built from.