Creating Dictionaries & Accessing Values

AiCE Software Engineering Foundations — Python Programming

A list stores items by position. A dictionary stores items by name. This seemingly small difference changes everything about how you organize and retrieve data. This lesson opens Python's dictionary — the data structure that makes Python feel like a real database.

1. What Is a Dictionary?

A dictionary is an unordered collection of key-value pairs. Each piece of data has two parts: a key, the label you use to look it up, and a value, the actual data stored under that label. The analogy is precise — a physical dictionary maps words (keys) to their definitions (values); a Python dictionary maps any key you define to any value you want to store.

Consider storing information about a student. With a list, you access everything by position:

```
# List approach — access by position only
student = ["Amara Diallo", 22, "Senegal", 3.85]

# To get the GPA, you must remember it is at index 3
print(student[3])
```

This works, but it's fragile — add a field between name and age, and every index after it shifts. With a dictionary:

```
# Dictionary approach — access by meaningful key
student = {
    "name"    : "Amara Diallo",
    "age"     : 22,
    "country" : "Senegal",
    "gpa"     : 3.85
}
```

```
# To get the GPA, you use its name
print(student["gpa"])
```

> **Self-Documenting Data**
> *student["gpa"] tells you exactly what you're retrieving — no need to remember positions, and adding new keys never breaks existing code. This is why dictionaries are the preferred way to represent structured records in Python.*

2. Creating a Dictionary

Dictionaries are written with curly braces {}. Each key-value pair is written as key: value, separated by commas:

```
# Creating a dictionary — AiCE founder profile
founder = {
    "name"        : "Adesina Olagunju",
    "title"       : "Founder",
    "organisation": "AiCE",
    "country"     : "Nigeria",
    "year_founded": 2025
}

print(founder)
print(type(founder))
print(f"Number of fields: {len(founder)}")
```

This produces:

```
{'name': 'Adesina Olagunju', 'title': 'Founder',
'organisation': 'AiCE', ...}
<class 'dict'>
Number of fields: 5
```

Dictionary Keys Must Be Unique

Within a single dictionary, every key must be unique. Defining the same key twice silently overwrites the first — no error is raised. Keys are almost always strings, though other immutable types can be used too. Values can be anything: strings, numbers, booleans, lists, or even other dictionaries.

3. Accessing Values by Key

You retrieve a value using its key inside square brackets — similar to a list index, but using a key name instead of a position:

```
student = {
    "name"    : "Ngozi Adeyemi",
    "age"     : 23,
    "country" : "Nigeria",
    "gpa"     : 3.72,
}

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

# Using values in f-strings
print(f"{student["name"]} from {student["country"]}")
print(f"GPA: {student["gpa"]:.2f}")
```

Keys Are Case-Sensitive

The key must match exactly, including case. student["Name"] would raise a KeyError because the key is "name", not "Name" — the same case-sensitivity rule that applies everywhere else in Python.

Using .get() — Safe Access

Accessing a missing key with square brackets raises a KeyError and crashes your program. .get() is a safer alternative — it returns None by default if the key isn't found, or a default value you specify:

```
student = {"name": "Kofi", "age": 21, "country": "Ghana"}

# Square bracket — raises KeyError if key missing
# print(student["gpa"])

# .get() — returns None if key missing
print(student.get("gpa"))              # None

# .get() with a default value
print(student.get("gpa", "Not set"))     # Not set
print(student.get("name", "Unknown"))    # Kofi
```

> **Prefer .get() Whenever a Key Might Be Missing**
> *In production code, .get() is preferred over direct bracket access whenever there's any chance a key might not exist — which is often, especially with data from APIs, files, or user input.*

Key Takeaways

A dictionary stores key-value pairs inside curly braces, retrieved by a meaningful key name rather than a numeric position — making it self-documenting in a way lists aren't. Keys must be unique and are case-sensitive, and .get() is the safer way to access a value that might not exist, returning None or a chosen default instead of crashing. The next lesson covers adding, updating, removing, and looping over everything a dictionary contains.