Tuples, Immutability & Unpacking

AiCE Software Engineering Foundations — Python Programming (Intermediate)

Python gives you four fundamental data structures: lists, dictionaries, tuples, and sets. You have mastered the first two. This lesson introduces tuples — ordered, immutable sequences that fill a role lists and dictionaries cannot: representing data that should never change.

1. What Is a Tuple?

A tuple is an ordered, immutable sequence of items. Immutable means it cannot be changed after creation — no adding, removing, or updating items. Once a tuple is defined, it is fixed.

Tuples are written with parentheses, items separated by commas:

```
coordinates = (6.5244, 3.3792)    # Lagos latitude and
longitude
rgb_green = (26, 107, 58)         # AiCE green in RGB values
student = ("Amara Diallo", 22, "Senegal", 3.85)
```

At first glance, a tuple looks like a list with parentheses instead of square brackets. The critical difference is that a list can be modified after creation — a tuple cannot. This immutability is not a limitation. It is a feature. It signals to anyone reading your code: this data is fixed. It will not change. You can rely on it.

> **A Sealed Envelope**
> *Think of a tuple as a sealed envelope. Once the contents are placed inside and the envelope is sealed, nothing goes in or comes out. The contents are protected by design.*

2. Creating Tuples

```
# Creating tuples
coordinates = (6.5244, 3.3792)
rgb_color = (26, 107, 58)
student_info = ("Ngozi Adeyemi", 23, "Nigeria", 3.72)
empty_tuple = ()

# A single-item tuple MUST have a trailing comma
single = ("Lagos",)      # Correct — this is a tuple
not_a_tuple = ("Lagos")  # Wrong — this is just a string in
brackets

print(type(single))
print(type(not_a_tuple))

# Tuples without parentheses (packing)
packed = "Accra", "Ghana", 2025
print(packed)
print(type(packed))
```

This produces:

```
<class 'tuple'>
<class 'str'>
('Accra', 'Ghana', 2025)
<class 'tuple'>
```

The single-item tuple requires a trailing comma — without it, Python treats the parentheses as grouping, not as a tuple constructor. Tuples can also be created without parentheses, just comma-separated values; Python recognizes them as tuples automatically. This is called tuple packing.

3. Accessing Tuple Items

Tuples use the same zero-based indexing and slicing as lists. Everything you learned about list indexing applies directly:

```
student = ("Amara Diallo", 22, "Senegal", 3.85)

# Indexing
print(student[0])     # Amara Diallo
print(student[1])     # 22
print(student[-1])    # 3.85 — last item

# Slicing
print(student[1:3])   # (22, "Senegal")

# Iterating
for item in student:
    print(item)

# Membership testing
print("Senegal" in student)    # True
print("Ghana" in student)      # False

# Length
print(len(student))    # 4
```

Tuple Unpacking — The Most Important Tuple Feature

Tuple unpacking assigns each item of a tuple to a separate variable in a single line. You have already seen this when working with functions that return multiple values and with .items() in dictionary iteration — this is where those patterns come from.

```
# Tuple unpacking
student = ("Amara Diallo", 22, "Senegal", 3.85)

# Unpack all four values
name, age, country, gpa = student
print(f"Name   : {name}")
print(f"Age    : {age}")
print(f"Country: {country}")
print(f"GPA    : {gpa}")

# Swap two variables using tuple unpacking
a = 10
b = 20
```

```
print(f"Before swap: a={a}, b={b}")
a, b = b, a
print(f"After swap : a={a}, b={b}")

# Unpack coordinates
lagos_coords = (6.5244, 3.3792)
latitude, longitude = lagos_coords
print(f"Lagos — Lat: {latitude}, Long: {longitude}")

# Extended unpacking with *
first, *middle, last = (10, 20, 30, 40, 50)
print(f"First : {first}")
print(f"Middle: {middle}")
print(f"Last  : {last}")
```

The variable swap — a, b = b, a — is a classic Python idiom. Python evaluates the right side first as a tuple (b, a), then unpacks it into a and b simultaneously — no temporary variable needed. The extended unpacking with * collects all remaining items into a list; here, middle becomes [20, 30, 40].

4. Tuples Are Immutable — What This Means in Practice

Attempting to modify a tuple raises an error:

```
student = ("Amara", 22, "Senegal")

# This will raise TypeError
student[0] = "Ngozi"
# TypeError: 'tuple' object does not support item assignment

# This will also raise an error
student.append("PhD")
# AttributeError: 'tuple' object has no attribute 'append'
```

Immutability makes tuples useful in specific situations:

When data should not change — geographic coordinates, RGB color codes, database primary keys, configuration constants.

As dictionary keys — lists cannot be used as dictionary keys since they're mutable, but tuples can.

As function return values — returning multiple values from a function always creates a tuple.

For performance — tuples are slightly faster than lists for iteration and use less memory.

```
# Tuples as dictionary keys (lists cannot do this)
locations = {
    (6.5244, 3.3792)  : "Lagos",
    (-1.2921, 36.8219): "Nairobi",
    (5.6037, -0.1870) : "Accra",
}

for coords, city in locations.items():
    lat, lon = coords
    print(f"{city}: ({lat}, {lon})")
```

5. Named Tuples — Readable Immutable Records

Python's collections module provides namedtuple — a tuple subclass where each position has a name. This gives you the immutability of a tuple with the readability of a dictionary:

```
from collections import namedtuple

# Define the named tuple structure
Student = namedtuple("Student", ["name", "country", "score",
"grade"])

# Create instances
amara = Student("Amara Diallo", "Senegal", 88, "A")
kofi = Student("Kofi Mensah", "Ghana", 45, "F")

# Access by name (readable)
print(amara.name)
print(amara.score)
```

```
# Access by index (still works)
print(kofi[0])

# Unpack like a regular tuple
name, country, score, grade = amara
print(f"{name} from {country}: {score} ({grade})")

# Still immutable
# amara.score = 95    # AttributeError
```

Named tuples are ideal for representing fixed records — student profiles, database rows, API responses — where the data should not change and you want named access rather than positional indexing.

> **Tuples vs. Lists — The Decision Rule**
> *Use a tuple when the data represents a fixed collection of related items that belong together and should not change — a coordinate pair, an RGB color, a record row. Use a list when the collection is dynamic — items will be added, removed, or updated. When in doubt, ask: should this data ever change? If no, tuple. If yes, list.*

Key Takeaways

A tuple is an ordered, immutable sequence written with parentheses — once created, it can never be changed, which is exactly why it's trusted as a dictionary key, a function's multi-value return, and a representation of fixed records. Tuple unpacking, including the a, b = b, a swap idiom and extended unpacking with *, is the single most useful tuple feature. namedtuple adds named field access on top of a regular tuple's guarantees. The next lesson introduces sets — Python's fourth fundamental data structure, built for uniqueness and fast membership testing.