Topic 3a: Tuples

Immutable Sequences

Learning Outcomes:

- Understand why tuples exist (immutability)
- Apply tuple syntax and operations
- Distinguish when to use tuples vs lists
- Use tuple unpacking effectively

Prerequisites: Topic 1 (Functions), Topic 2 (Lists)

Opening Problem

You're building a GPS tracking system.

Location coordinates should never change once recorded.

```
# Storing Kuala Lumpur coordinates as a list
kl_location = [3.1390, 101.6869]

print("Original:", kl_location)
```

What could go wrong?

The Problem: Accidental Modification

```
kl_location = [3.1390, 101.6869]

# Somewhere deep in the code...
def process_location(loc):
    loc[0] = 0  # Bug: accidentally modifying the original!
    return loc

process_location(kl_location)

print("After processing:", kl_location)
```

Output:

```
After processing: [0, 101.6869]
```

The original data is corrupted. No error was raised.

Why This Happens

Lists are mutable — they can be changed after creation.

When you pass a list to a function, the function receives a reference to the same list, not a copy.

```
coordinates = [3.1390, 101.6869]
backup = coordinates  # Not a copy — same list!

coordinates[0] = 0
print(backup)  # [0, 101.6869] — backup is also modified!
```

For data that should never change, lists are dangerous.

There must be a better way.

The Solution: Tuples

Tuples are immutable sequences.

```
kl_location = (3.1390, 101.6869)  # Parentheses, not brackets

kl_location[0] = 0  # Attempting to modify
```

Output:

```
TypeError: 'tuple' object does not support item assignment
```

Python prevents the modification. The bug is caught immediately.

Tuples Protect Your Data

```
kl_location = (3.1390, 101.6869)

def process_location(loc):
    loc[0] = 0  # TypeError raised here!
    return loc

process_location(kl_location)
```

The program fails fast — you know exactly where the problem is.

With lists, the bug might go unnoticed until much later.

Where Tuples Fit

Python provides 4 built-in data structures:

Data Structure	Symbol	Mutable?	Ordered?	Primary Use
List	`[ ]`	Yes	Yes	Collections that change
Tuple	`( )`	**No**	Yes	Collections that should not change

Tuple vs List: The Core Difference

Feature	List	Tuple
Symbol	`[1, 2, 3]`	`(1, 2, 3)`
Mutable	Yes	**No**
Can modify elements	Yes	No
Can add/remove elements	Yes	No
Memory usage	More	**Less**
Speed	Slower	**Faster**

Think of it as: Tuple = "Frozen List"

Structure of a Tuple

```
my_tuple = (10, 20, 30, "Python", 3.14)
```

Breaking it down:

```
      tuple name                  elements (comma-separated)
          |                                 |
          ▼                                 ▼
my_tuple = (10, 20, 30, "Python", 3.14)
           |                              |
           └──── parentheses ( ) ─────────┘
```

Component	Value
Tuple name	`my_tuple`
Elements	`10` , `20` , `30` , `"Python"` , `3.14`

Creating Tuples: Multiple Ways

Method 1: Using parentheses

```
coordinates = (3.14, 101.68)
print(coordinates)  # (3.14, 101.68)
```

Method 2: Using the `tuple()` function

```
numbers = tuple([1, 2, 3])  # Convert list to tuple
print(numbers)  # (1, 2, 3)
```

Method 3: Without parentheses (tuple packing)

```
point = 5, 10, 15  # Still a tuple!
print(point)       # (5, 10, 15)
print(type(point)) # <class 'tuple'>
```

Special Cases: Empty and Single-Element Tuples

Empty tuple:

```
empty = ()
# or
empty = tuple()

print(empty)        # ()
print(len(empty))   # 0
```

Single-element tuple — trailing comma required:

```
single = (5,)   # Correct: tuple with one element
print(single)   # (5,)

wrong = (5)     # Wrong: just the integer 5 in parentheses
print(wrong)    # 5
print(type(wrong))  # <class 'int'>
```

The Trailing Comma Rule

Why is `(5)` not a tuple?

Parentheses are used for grouping in math expressions:

```
result = (5 + 3) * 2  # Parentheses for grouping, not tuple
print(result)  # 16
```

Python needs the comma to distinguish:

```
grouping = (5)     # Integer 5
one_tuple = (5,)   # Tuple containing 5

print(type(grouping))  # <class 'int'>
print(type(one_tuple)) # <class 'tuple'>
```

Rule: Single-element tuples require a trailing comma.

Exercise 1: Identify Tuple Syntax

Which of these create valid tuples?

```
a = (1, 2, 3)
b = 1, 2, 3
c = (1)
d = (1,)
e = tuple([1, 2, 3])
f = ()
g = tuple()
```

Identify:

1. Which create tuples?
2. What is the type of `c`?
3. How many elements does `d` have?

Tuple Characteristics

Tuples share many characteristics with lists:

Characteristic	List	Tuple
Ordered	Yes	Yes
Indexed	Yes	Yes
Allows duplicates	Yes	Yes
Supports multiple types	Yes	Yes
Iterable	Yes	Yes
Mutable	Yes	**No**

The only difference: **Tuples cannot be modified after creation.**

Indexing: Same as Lists

Important: Even though tuples use `()` for creation, we use `[]` for indexing.

```
fruits = ("apple", "banana", "cherry", "date")  # Created with ()
print(fruits[0])   # Accessed with [] — NOT fruits(0)
```

Positive indexing (left to right, starting at 0):

```
fruits = ("apple", "banana", "cherry", "date")
#           0         1         2        3

print(fruits[0])   # apple
print(fruits[2])   # cherry
```

Negative indexing (right to left, starting at -1):

```
print(fruits[-1])  # date (last element)
print(fruits[-2])  # cherry (second to last)
```

Slicing: Extracting a Range of Elements

Slice syntax: `tuple[start:end]`

- `start` : Index where slice begins (inclusive)
- `end` : Index where slice ends (exclusive — this index is NOT included)

```
numbers = (10, 20, 30, 40, 50, 60)
#           0   1   2   3   4   5
```

Slicing: Basic Examples

```
numbers = (10, 20, 30, 40, 50, 60)
#           0   1   2   3   4   5
```

Extract elements from index 1 to 3:

```
print(numbers[1:4])   # (20, 30, 40)
# Start at index 1, stop BEFORE index 4
```

Omitting start — defaults to 0:

```
print(numbers[:3])    # (10, 20, 30)
# Same as numbers[0:3]
```

Omitting end — goes to the last element:

```
print(numbers[3:])    # (40, 50, 60)
# From index 3 to the end
```

Looping Through Tuples

Using a for loop:

```
colors = ("red", "green", "blue")

for color in colors:
    print(color)
```

Output:

```
red
green
blue
```

Works the same as looping through lists.

CRUD Operations Overview

Operation	List	Tuple
Create	`[1, 2, 3]`	`(1, 2, 3)`
Read	`list[0]`, slicing	`tuple[0]`, slicing
Update	`list[0] = 99`	Not directly possible
Delete	`del list[0]`, `remove()`	Not directly possible

Tuples are read-only after creation.

CREATE: Making Tuples

```
# Empty tuple
empty = ()

# From literal
coordinates = (3.14, 101.68)

# From list conversion
numbers = tuple([1, 2, 3])

# Single element (trailing comma!)
single = (42,)

# Mixed types (allowed but rarely recommended)
mixed = (1, "hello", 3.14, True)

# Nested tuples
nested = ((1, 2), (3, 4), (5, 6))
```

READ: Accessing Tuple Elements

By index:

```
point = (10, 20, 30)
x = point[0]   # 10
y = point[1]   # 20
z = point[2]   # 30
```

By slicing:

```
first_two = point[:2]  # (10, 20)
```

By looping:

```
for value in point:
    print(value)
```

By unpacking (covered next):

```
x, y, z = point  # x=10, y=20, z=30
```

UPDATE: The Immutability Challenge

Direct modification fails:

```
point = (10, 20, 30)
point[0] = 99
```

Output:

```
TypeError: 'tuple' object does not support item assignment
```

UPDATE: Workaround (But Question It!)

Workaround: Convert to list, modify, convert back

```
point = (10, 20, 30)

# Step 1: Convert to list
temp_list = list(point)

# Step 2: Modify the list
temp_list[0] = 99

# Step 3: Convert back to tuple
point = tuple(temp_list)

print(point)  # (99, 20, 30)
```

If you need this workaround often, ask yourself: Do you really need a tuple?

Perhaps a list is more appropriate for your use case.

UPDATE: Using Concatenation

Add elements by creating a new tuple:

```
original = (1, 2, 3)

# Add element at the end
extended = original + (4,)
print(extended)  # (1, 2, 3, 4)

# Add element at the beginning
extended = (0,) + original
print(extended)  # (0, 1, 2, 3)

# Combine tuples
combined = (1, 2) + (3, 4) + (5, 6)
print(combined)  # (1, 2, 3, 4, 5, 6)
```

Note: The original tuple is unchanged. A new tuple is created.

DELETE: Removing Elements

Direct deletion of elements fails:

```
point = (10, 20, 30)
del point[0]
```

Output:

```
TypeError: 'tuple' object doesn't support item deletion
```

DELETE: Workaround (But Question It!)

Workaround: Convert to list, remove, convert back

```
data = (1, 2, 3, 4, 5)

temp_list = list(data)
temp_list.remove(3)  # Remove value 3
data = tuple(temp_list)

print(data)  # (1, 2, 4, 5)
```

If you need this workaround often, ask yourself: Do you really need a tuple?

Delete entire tuple:

```
data = (1, 2, 3)
del data  # Deletes the variable
# print(data)  # NameError: name 'data' is not defined
```

Exercise 2: CRUD Operations

Given this tuple:

```
scores = (85, 92, 78, 88, 95)
```

Tasks:

1. Access the third element
2. Get the last two elements using slicing
3. Change the first element to 90 (show the workaround)
4. Add 100 to the end of the tuple

Tuple Unpacking

Assign tuple elements to individual variables:

```
coordinates = (3.14, 101.68)

# Traditional way
lat = coordinates[0]
lon = coordinates[1]

# Unpacking way (cleaner)
lat, lon = coordinates

print(lat)  # 3.14
print(lon)  # 101.68
```

The number of variables must match the number of elements.

Unpacking Error: Mismatched Count

Too few variables:

```
point = (10, 20, 30)
x, y = point
```

Output:

```
ValueError: too many values to unpack (expected 2)
```

Too many variables:

```
point = (10, 20)
x, y, z = point
```

Output:

Tuple Unpacking: More Examples

With three elements:

```
rgb = (255, 128, 0)
red, green, blue = rgb

print(f"Red: {red}, Green: {green}, Blue: {blue}")
# Red: 255, Green: 128, Blue: 0
```

Swapping values:

```
a = 5
b = 10

a, b = b, a  # Swap using tuple unpacking

print(a)  # 10
print(b)  # 5
```

Functions and Tuples

Returning multiple values as a tuple:

```
def get_min_max(numbers):
    return min(numbers), max(numbers)  # Returns a tuple

data = [5, 2, 8, 1, 9, 3]
result = get_min_max(data)
print(result)       # (1, 9)
print(type(result)) # <class 'tuple'>

# Unpack at the call site
minimum, maximum = get_min_max(data)
print(f"Min: {minimum}, Max: {maximum}")
# Min: 1, Max: 9
```

Functions and Tuples: More Examples

Calculate statistics:

```
def calculate_stats(numbers):
    total = sum(numbers)
    count = len(numbers)
    average = total / count
    return total, count, average

scores = [85, 92, 78, 88, 95]
total, count, avg = calculate_stats(scores)

print(f"Total: {total}")    # Total: 438
print(f"Count: {count}")    # Count: 5
print(f"Average: {avg}")    # Average: 87.6
```

This pattern is cleaner than returning a list or dictionary.

Tuples as Function Parameters

Accept tuple as parameter:

```
def calculate_distance(point1, point2):
    x1, y1 = point1  # Unpack first point
    x2, y2 = point2  # Unpack second point

    distance = ((x2 - x1)**2 + (y2 - y1)**2) ** 0.5
    return distance

start = (0, 0)
end = (3, 4)

dist = calculate_distance(start, end)
print(f"Distance: {dist}")  # Distance: 5.0
```

Exercise 3: Tuple Unpacking

What will this code print?

```
def get_student_info(student_tuple):
    name, age, gpa = student_tuple
    return name, gpa

student = ("Ali", 20, 3.85)
student_name, student_gpa = get_student_info(student)

print(f"{student_name} has GPA {student_gpa}")
```

Tuple Comparison

Tuples are compared element by element:

```
print((1, 2, 3) == (1, 2, 3))  # True
print((1, 2, 3) == (1, 2, 4))  # False

print((1, 2, 3) < (1, 2, 4))   # True (compares element by element)
print((1, 2, 3) < (2, 0, 0))   # True (first element decides)
```

Comparison follows lexicographic order:

1. Compare first elements
2. If equal, compare second elements
3. Continue until a difference is found

Membership Testing

Check if element exists:

```
colors = ("red", "green", "blue")

print("red" in colors)      # True
print("yellow" in colors)   # False
print("yellow" not in colors)  # True
```

Works the same as lists.

Built-in Functions for Tuples

Function	Description	Example
`len(t)`	Number of elements	`len((1,2,3))` → `3`
`min(t)`	Smallest element	`min((5,2,8))` → `2`
`max(t)`	Largest element	`max((5,2,8))` → `8`
`sum(t)`	Sum of elements	`sum((1,2,3))` → `6`
`tuple(iterable)`	Convert to tuple	`tuple([1,2,3])` → `(1,2,3)`

Using `len()`, `min()`, `max()`, `sum()`

```
scores = (85, 92, 78, 88, 95)

print(f"Count: {len(scores)}")      # 5
print(f"Minimum: {min(scores)}")    # 78
print(f"Maximum: {max(scores)}")    # 95
print(f"Total: {sum(scores)}")      # 438
print(f"Average: {sum(scores)/len(scores)}")  # 87.6
```

Same functions work on lists and tuples.

Converting Between Types

```
# List to tuple
my_list = [1, 2, 3]
my_tuple = tuple(my_list)
print(my_tuple)  # (1, 2, 3)

# Tuple to list
my_tuple = (1, 2, 3)
my_list = list(my_tuple)
print(my_list)  # [1, 2, 3]

# String to tuple
my_string = "hello"
char_tuple = tuple(my_string)
print(char_tuple)  # ('h', 'e', 'l', 'l', 'o')
```

Exercise 4: Built-in Functions

Given this tuple:

```
temperatures = (28.5, 30.2, 27.8, 32.1, 29.5, 31.0, 28.8)
```

Write code to:

1. Find the number of readings
2. Find the lowest temperature
3. Find the highest temperature
4. Calculate the average temperature

The Principled Approach

4 principles for using tuples properly:

1. Unpack Properly with Meaningful Names
2. Unpack Before Operating
3. Question Workarounds
4. Keep Elements the Same Type

Principle 1: Unpack Properly with Meaningful Names

The Wrong Way:

```
point = (10, 20, 30)
x, y = point  # ValueError! Wrong count
```

Output:

```
ValueError: too many values to unpack (expected 2)
```

Principle 1: The Wrong Way (Continued)

Also wrong — meaningless names:

```
data = (3.14, 101.68)
a, b = data  # What are a and b?

result = a + b  # Meaningless code
```

Principle 1: The Correct Way

Step 1: Match the count

Step 2: Use descriptive names

```
coordinates = (3.14, 101.68)

# Check count if unsure
print(len(coordinates))  # 2

# Unpack with meaningful names
latitude, longitude = coordinates

print(f"Location: {latitude}, {longitude}")
```

Rule: Match the element count AND use names that describe the data.

Principle 1: More Examples

```
# Bad: Generic names
p = (255, 128, 0)
a, b, c = p

# Good: Descriptive names
rgb_color = (255, 128, 0)
red, green, blue = rgb_color

# Bad: Wrong count AND bad names
point = (10, 20, 30)
x, y = point  # Error!

# Good: Correct count AND good names
point = (10, 20, 30)
x, y, z = point
```

Principle 2: Unpack Before Operating

The Wrong Way:

```
def process_point(point):
    # Directly using index in calculation
    result = point[0] * 2 + point[1] * 3
    return result
```

Hard to read. What is index 0? What is index 1?

Principle 2: The Correct Way

Unpack first, then operate:

```
def process_point(point):
    # Step 1: Unpack into named variables
    x, y = point

    # Step 2: Use the named variables
    result = x * 2 + y * 3
    return result

point = (10, 20)
print(process_point(point))  # 80
```

Rule: Unpack tuple elements into named variables before using them in operations.

Principle 2: Another Example

```
# Bad: Indices everywhere
def calculate_area(rectangle):
    return rectangle[0] * rectangle[1]

# Good: Unpack first
def calculate_area(rectangle):
    width, height = rectangle
    return width * height

rect = (10, 5)
print(calculate_area(rect))  # 50
```

Principle 3: Question Workarounds

If you find yourself doing this often:

```
data = (1, 2, 3)

# Convert to list
temp = list(data)
temp.append(4)
data = tuple(temp)
```

Stop and ask: Do I really need a tuple here?

Principle 3: Make the Right Choice

If data needs to change → Use a list from the start

```
# Wrong: Tuple that keeps getting converted
scores = (85, 90)
temp = list(scores)
temp.append(78)
scores = tuple(temp)  # Wasteful!

# Right: List when data changes
scores = [85, 90]
scores.append(78)  # Simple and efficient
```

Rule: If you need workarounds frequently, reconsider your data structure choice.

Principle 4: Group Data That Belongs Together

Tuples are for data that forms a logical unit.

Good examples:

```
# Coordinates — x and y belong together as one point
point = (10, 20)

# Database connection — all needed to connect
db_config = ("localhost", 3306, "admin", "secret123")
#             host       port   user    password

# RGB color — three values that define one color
color = (255, 128, 0)
```

Principle 4: The Key Question

Ask: "Do these values belong together as one thing?"

```
# Good: Values that form a logical unit
coordinates = (3.14, 101.68)  # One location
rgb = (255, 128, 0)           # One color
date = (2024, 1, 15)          # One date

# Bad: Unrelated values thrown together
random_stuff = (42, "hello", True, 3.14)  # No logical connection
```

Principle 4: Unpacking Connection Data

```
# Database connection tuple
db_config = ("localhost", 3306, "admin", "secret123")

# Unpack with meaningful names
host, port, username, password = db_config

# Now use them
print(f"Connecting to {host}:{port} as {username}")
# Connecting to localhost:3306 as admin
```

The values are different types, but they belong together.

Principle 4: The Rule

> "Use tuples for data that belongs together as a logical unit."

Good Use (Cohesive)	Bad Use (Random)
`(x, y)` coordinates	`(42, "hello", True)`
`(host, port, user, pass)`	Unrelated values
`(year, month, day)`	Data with no connection
`(red, green, blue)`	

Rule: If the values form a single concept together, use a tuple.

Principles Summary

#	Principle	How To Apply
1	Unpack Properly	Match count + use meaningful names
2	Unpack Before Operating	Unpack first, then calculate
3	Question Workarounds	If converting often, use a list
4	Group Related Data	Use tuples for data that belongs together

Common Mistakes

1. Forgetting trailing comma for single element:

```
single = (5)   # Wrong: integer 5
single = (5,)  # Correct: tuple with one element
```

2. Trying to modify tuple elements:

```
point = (10, 20)
point[0] = 99  # TypeError!
```

Common Mistakes (Continued)

3. Expecting `sorted()` to return a tuple:

```
numbers = (5, 2, 8)
result = sorted(numbers)  # Returns a list, not tuple!
print(type(result))  # <class 'list'>
```

4. Mismatched unpacking:

```
point = (10, 20, 30)
x, y = point  # ValueError: too many values to unpack
```

Common Mistakes (Continued)

5. Using list when tuple is more appropriate:

```
# Bad: coordinates can be accidentally modified
location = [3.14, 101.68]

# Good: coordinates are protected
location = (3.14, 101.68)
```

6. Confusing tuple creation with function calls:

```
result = function(1, 2, 3)  # Function call
my_tuple = (1, 2, 3)        # Tuple creation
```

Summary

What we covered:

1. **Tuples are immutable sequences** — cannot be modified after creation
2. **Syntax:** Parentheses `()`, comma-separated elements
3. **Single element:** Requires trailing comma `(5,)`
4. **CRUD Operations:** Read works, Update/Delete require conversion
5. **Tuple Unpacking:** Clean way to access elements
6. **Functions:** Tuples are ideal for returning multiple values
7. **Built-in tools:** `len()`, `min()`, `max()`, `sum()`, `count()`, `index()`
8. **Hashable:** Can be used as dictionary keys and set elements

Tuples vs Lists: Quick Reference

Aspect	List	Tuple
Syntax	`[1, 2, 3]`	`(1, 2, 3)`
Mutable	Yes	No
Methods	11+	2
Memory	More	Less
Speed	Slower	Faster
As dict key	No	Yes
Use case	Variable data	Fixed data

Principles Quick Reference

#	Principle	Rule
1	Immutability by Design	If it shouldn't change, make it a tuple
2	Fixed vs Variable	Fixed structure → Tuple
3	Unpacking	Unpack into named variables
4	Multiple Returns	Return as tuples
5	Hashable Keys	Tuples can be dict keys

What's Next

Topic 3b: Sets

- Tuples can be **set elements** because they are immutable
- Lists **cannot** be set elements

```
# Valid: set of tuples
coordinates = {(0, 0), (1, 0), (0, 1)}

# Invalid: set of lists
# coordinates = {[0, 0], [1, 0]}  # TypeError!
```

Understanding tuple immutability is key to understanding sets.