Introduction to Control Structures: Iterations in Python

Part 1: Understanding Control Structures (30 minutes)

What Are Control Structures?

Control structures are programming constructs that determine the order in which statements are executed. They control the "flow" of your program.

The Three Types of Control Structures:

1. Sequential Control Structure

- Default flow of execution
- Code runs line by line, top to bottom

```
# Sequential execution
print("Step 1")
print("Step 2")
print("Step 3")
# Output: Step 1, Step 2, Step 3 (in order)
```

2. Conditional Control Structure (Selection)

- Makes decisions based on conditions
- Uses if, elif, else statements

```
# Conditional execution
age = 18
if age >= 18:
   print("You can vote")
else:
   print("You cannot vote yet")
```

3. Iterative Control Structure (Repetition)

- Repeats a block of code multiple times

- Uses for and while loops

```
# Iterative execution
for i in range(3):
    print("Python is awesome!")
# Output prints the same message 3 times
```

But, our focus is on iteration control structures.

What is Iteration? (30 minutes)

Definition

Iteration is the process of repeating a set of instructions multiple times until a specific condition is met or until all items in a collection have been processed.

Why Do We Need Iteration?

Without Loops (Repetitive and Inefficient):

```
# Printing numbers 1 to 5 without loops
print(1)
print(2)
print(3)
print(4)
print(5)
# Imagine doing this for 1000 numbers! 😱
```

With Loops (Efficient and Clean):

```
# Printing numbers 1 to 5 with a loop
for i in range(1, 6):
    print(i)
# Works for any range - even 1000 numbers!
```

Benefits of Iteration

1. **Eliminates Code Repetition** - Write once, repeat many times
2. **Handles Large Data Sets** - Process thousands of items easily

3. **Automates Repetitive Tasks** - Reduces human error
4. **Makes Code Maintainable** - Changes need to be made in one place
5. **Increases Efficiency** - Faster development and execution

Types of Iteration

Type	When to Use	Loop Type
Definite Iteration	When you know how many times to repeat	for loop
Indefinite Iteration	When you repeat until a condition is met	while loop

---

Part 3: The FOR Loop - Definite Iteration (60 minutes)

Understanding the FOR Loop

A for loop is used when you want to iterate over a sequence or repeat code a specific number of times.

Real-World Analogy: Imagine you have a class roster with 30 student names. You want to call each student's name one by one. You know there are 30 names, so you'll call exactly 30 names - no more, no less.

Basic Syntax

```
for variable in sequence:
    # code block to execute
    # this code runs for each item in sequence
```

Components:

- **for**: Keyword that starts the loop
- **variable**: Temporary variable that holds current item (loop variable)
- **in**: Keyword that separates variable from sequence
- **sequence**: Collection to iterate over (list, string, range, etc.)
- **:**: Colon marks the start of the code block
- **Indentation**: Code block must be indented (4 spaces or 1 tab)

Example 1: Looping Through a List

```
# List of fruits
fruits = ["apple", "banana", "cherry", "date"]

# Loop through each fruit
for fruit in fruits:
    print(f"I like {fruit}")

# Output:
# I like apple
# I like banana
# I like cherry
# I like date
```

How It Works:

1. Loop starts, fruit takes the value "apple"
2. Prints "I like apple"
3. Loop continues, fruit becomes "banana"
4. Prints "I like banana"
5. Continues until all items are processed

Example 2: Looping Through a String

```
# Strings are sequences of characters
word = "PYTHON"

for letter in word:
    print(letter)

# Output:
# P
# Y
# T
# H
# O
# N
```

Example 3: Looping Through a Tuple

```
# Tuple of coordinates
coordinates = (10, 20, 30, 40)

for coord in coordinates:
    print(f"Coordinate: {coord}")

# Output:
# Coordinate: 10
# Coordinate: 20
# Coordinate: 30
# Coordinate: 40
```

Example 4: Looping Through a Dictionary

Dictionaries have three ways to loop:

```
# Sample dictionary
student_scores = {
    "Alice": 95,
    "Bob": 87,
    "Charlie": 92
}

# Method 1: Loop through keys (default)
print("Method 1: Keys only")
for name in student_scores:
    print(name)
# Output: Alice, Bob, Charlie

# Method 2: Loop through values
print("\nMethod 2: Values only")
for score in student_scores.values():
    print(score)
# Output: 95, 87, 92

# Method 3: Loop through both keys and values
print("\nMethod 3: Keys and Values")
for name, score in student_scores.items():
```

```
    print(f"{name} scored {score}")
# Output:
# Alice scored 95
# Bob scored 87
# Charlie scored 92
```

The range() Function (40 minutes)

What is range()?

The range() function generates a sequence of numbers. It's commonly used with for loops when you need to repeat code a specific number of times.

Three Forms of range()

1. range(stop)

Generates numbers from 0 to stop-1

```
# range(5) generates: 0, 1, 2, 3, 4
for i in range(5):
    print(i)

# Output:
# 0
# 1
# 2
# 3
# 4
```

2. range(start, stop)

Generates numbers from start to stop-1

```
# range(2, 7) generates: 2, 3, 4, 5, 6
for i in range(2, 7):
    print(i)
```

```
# Output:
# 2
# 3
# 4
# 5
# 6
```

3. range(start, stop, step)

Generates numbers from start to stop-1, incrementing by step

```
# range(0, 10, 2) generates: 0, 2, 4, 6, 8
for i in range(0, 10, 2):
    print(i)

# Output:
# 0
# 2
# 4
# 6
# 8
```

Practical Examples with range()

Example 1: Countdown Timer

```
# Count down from 10 to 1
for i in range(10, 0, -1):
    print(i)
print("Blast off! 🚀 ")

# Output: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, Blast off! 🚀
```

Example 2: Multiplication Table

```
# Print multiplication table for 5
number = 5
for i in range(1, 11):
    result = number * i
```

```
    print(f"{number} × {i} = {result}")

# Output:
# 5 × 1 = 5
# 5 × 2 = 10
# 5 × 3 = 15
# ... and so on
```

Example 3: Sum of Numbers

```
# Calculate sum of numbers from 1 to 100
total = 0
for num in range(1, 101):
    total += num
print(f"Sum of 1 to 100 is: {total}")

# Output: Sum of 1 to 100 is: 5050
```

Example 4: Using range() with List Indexing

```
# Access list items by index
colors = ["red", "green", "blue", "yellow"]

for i in range(len(colors)):
    print(f"Color {i+1}: {colors[i]}")

# Output:
# Color 1: red
# Color 2: green
# Color 3: blue
# Color 4: yellow
```

Part 5: Important Rules for FOR Loops (30 minutes)

Rule 1: The Object Must Be Iterable

Iterables in Python:

- Lists: [1, 2, 3]
- Strings: "hello"
- Tuples: (1, 2, 3)
- Dictionaries: {"a": 1, "b": 2}
- Sets: {1, 2, 3}
- Ranges: range(10)
- Files: file objects

Non-Iterables:

- Integers: 10
- Floats: 3.14
- Booleans: True, False
- None: None

```
# ✅ This works
for x in [1, 2, 3]:
    print(x)

# ❌ This gives an error
for x in 10:  # TypeError: 'int' object is not iterable
    print(x)

# ✅ Correct way to loop 10 times
for x in range(10):
    print(x)
```

Rule 2: Use Meaningful Variable Names

```
# ❌ Bad: Unclear variable names
for x in ["apple", "banana", "cherry"]:
    print(x)

# ✅ Good: Clear, descriptive names
for fruit in ["apple", "banana", "cherry"]:
    print(fruit)

# ✅ Good: Even better with context
```

```
for student_name in class_roster:
    print(f"Present: {student_name}")
```

Common Naming Conventions:

- Single items from collections: for item in items
- Index counters: for i in range(10) or for idx in range(len(list))
- Key-value pairs: for key, value in dictionary.items()

Rule 3: Indentation is Mandatory

```
# ✅ Correct indentation
for i in range(3):
    print("Inside loop")
    print("Still inside loop")
print("Outside loop")

# ❌ IndentationError
for i in range(3):
print("This will cause an error")

# ❌ Inconsistent indentation
for i in range(3):
    print("4 spaces")
      print("6 spaces - Error!")
```

Rule 4: Don't Modify the Collection While Iterating

```
# ❌ BAD: Modifying list during iteration
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num % 2 == 0:
        numbers.remove(num)  # Dangerous!
print(numbers)  # Unpredictable results

# ✅ GOOD: Create a copy
numbers = [1, 2, 3, 4, 5]
for num in numbers[:]:  # [:] creates a copy
```

```
    if num % 2 == 0:
        numbers.remove(num)
print(numbers)  # [1, 3, 5]

# ✅ BETTER: Create a new list
numbers = [1, 2, 3, 4, 5]
odd_numbers = [num for num in numbers if num % 2 != 0]
print(odd_numbers)  # [1, 3, 5]
```

Rule 5: Understand Loop Scope

```
# Loop variable persists after loop ends
for i in range(5):
    pass

print(i)  # Output: 4 (last value)

# Be careful with variable naming
x = 100
for x in range(5):
    print(x)  # Uses loop variable

print(x)  # Output: 4 (original value overwritten!)
```

WHILE LOOPS & LOOP CONTROL (3 Hours)

Part 1: The WHILE Loop - Indefinite Iteration (45 minutes)

Understanding the WHILE Loop

A while loop repeats code as long as a condition remains True. Unlike a for loop, you don't always know in advance how many times it will run.

Real-World Analogy: Imagine filling a water tank. You keep filling while the tank is not full. You don't know exactly how long it will take - you just keep going until the condition "tank is full" becomes true.

Basic Syntax

```
while condition:
   # code block to execute
   # update condition variable
```

Components:

- **while**: Keyword that starts the loop
- **condition**: Boolean expression (evaluates to True or False)
- **:**: Colon marks the start of code block
- **Indentation**: Code block must be indented
- **Update**: You must change something to eventually make condition False

Flow of Execution

1. Check condition
2. If True, execute code block
3. Return to step 1
4. If False, exit loop and continue with rest of program

Example 1: Basic Counter

```
# Count from 1 to 5
count = 1

while count <= 5:
   print(f"Count is: {count}")
   count += 1  # Critical: Update the condition variable

print("Done counting!")

# Output:
# Count is: 1
# Count is: 2
# Count is: 3
# Count is: 4
# Count is: 5
# Done counting!
```

Step-by-Step Execution:

1. count = 1, check 1 <= 5 → True → print and increment
2. count = 2, check 2 <= 5 → True → print and increment
3. count = 3, check 3 <= 5 → True → print and increment
4. count = 4, check 4 <= 5 → True → print and increment
5. count = 5, check 5 <= 5 → True → print and increment
6. count = 6, check 6 <= 5 → False → exit loop

Example 2: User Input Validation

```
# Keep asking until valid input is received
password = ""

while password != "python123":
    password = input("Enter password: ")
    if password != "python123":
        print("Incorrect! Try again.")
    else:
        print("Access granted!")

# Loop continues until correct password is entered
```

Example 3: Sum Until Threshold

```
# Add numbers until sum exceeds 50
total = 0
number = 1

while total <= 50:
    total += number
    print(f"Added {number}, total is now {total}")
    number += 1

print(f"Final total: {total}")
```

Example 4: Menu System

```
# Simple menu that runs until user quits
choice = ""
```

```
while choice != "q":
    print("\n--- Menu ---")
    print("1. Say Hello")
    print("2. Say Goodbye")
    print("q. Quit")

    choice = input("Enter your choice: ")

    if choice == "1":
        print("Hello, User!")
    elif choice == "2":
        print("Goodbye, User!")
    elif choice == "q":
        print("Exiting program...")
    else:
        print("Invalid choice!")
```

Part 2: Critical Rules for WHILE Loops (30 minutes)

Rule 1: Condition Must Eventually Become False

```
# ❌ INFINITE LOOP - Never stops!
x = 0
while x < 5:
    print(x)
    # x never changes, so x < 5 is always True

# ✅ CORRECT - Condition eventually becomes False
x = 0
while x < 5:
    print(x)
    x += 1  # x increases, will eventually be >= 5
```

Rule 2: Initialize Variables Before the Loop

```
# ❌ BAD - Variable not initialized
while count < 10:  # NameError: name 'count' is not defined
```

```
    print(count)
    count += 1

# ✅ GOOD - Variable initialized
count = 0
while count < 10:
    print(count)
    count += 1
```

Rule 3: Update Inside the Loop

```
# ❌ BAD - Update outside the loop (infinite loop)
num = 1
while num <= 5:
    print(num)
num += 1  # This is outside, loop never ends!

# ✅ GOOD - Update inside the loop
num = 1
while num <= 5:
    print(num)
    num += 1  # Inside the loop, properly indented
```

Rule 4: Be Careful with Logical Operators

```
# Example: Multiple conditions
age = 15
has_permission = False

# Both conditions must be met to enter
while age < 18 and not has_permission:
    print("You need permission or to be 18+")
    age = int(input("Enter your age: "))
    if age < 18:
        response = input("Do you have permission? (yes/no): ")
        has_permission = (response.lower() == "yes")

print("Access granted!")
```

Part 3: Loop Control Statements (45 minutes)

The BREAK Statement

Purpose: Exit the loop immediately, regardless of the condition.

```
# Example 1: Search and stop when found
numbers = [10, 25, 30, 45, 50, 60]
target = 45

for num in numbers:
    if num == target:
        print(f"Found {target}!")
        break  # Exit loop immediately
    print(f"Checking {num}...")

# Output:
# Checking 10...
# Checking 25...
# Checking 30...
# Found 45!
# Example 2: Break in while loop
attempts = 0
max_attempts = 3

while True:  # Infinite loop
    password = input("Enter password: ")
    attempts += 1

    if password == "secret":
        print("Access granted!")
        break  # Exit on success

    if attempts >= max_attempts:
        print("Too many attempts!")
        break  # Exit on too many failures
```

```
    print(f"Wrong! {max_attempts - attempts} attempts left.")
```

The CONTINUE Statement

Purpose: Skip the rest of the current iteration and move to the next one.

```
# Example 1: Skip even numbers
for i in range(1, 11):
    if i % 2 == 0:
        continue  # Skip even numbers
    print(i)

# Output: 1, 3, 5, 7, 9 (only odd numbers)
# Example 2: Skip specific items
fruits = ["apple", "banana", "cherry", "date"]

for fruit in fruits:
    if fruit == "banana":
        continue  # Skip banana
    print(f"I like {fruit}")

# Output:
# I like apple
# I like cherry
# I like date
# Example 3: Continue in while loop
count = 0

while count < 10:
    count += 1

    if count % 3 == 0:
        continue  # Skip multiples of 3

    print(count)

# Output: 1, 2, 4, 5, 7, 8, 10
```

The ELSE Clause with Loops

Purpose: Code in else runs when loop completes normally (not interrupted by break).

```
# Example 1: else with for loop
for i in range(5):
    print(i)
else:
    print("Loop completed successfully!")

# Output:
# 0
# 1
# 2
# 3
# 4
# Loop completed successfully!
# Example 2: else with break
numbers = [1, 3, 5, 7, 9]
target = 4

for num in numbers:
    if num == target:
        print(f"Found {target}!")
        break
else:
    print(f"{target} not found in the list")

# Output: 4 not found in the list
# Example 3: else with while loop
count = 0

while count < 5:
    print(count)
    count += 1
else:
    print("While loop finished!")
```

```
# Output:
# 0
# 1
# 2
# 3
# 4
# While loop finished!
```

Comparison: break vs continue vs else

```
# Demonstration of all three
print("=== Example with BREAK ===")
for i in range(5):
    if i == 3:
        break
    print(i)
else:
    print("Completed")  # Won't print
# Output: 0, 1, 2

print("\n=== Example with CONTINUE ===")
for i in range(5):
    if i == 3:
        continue
    print(i)
else:
    print("Completed")  # Will print
# Output: 0, 1, 2, 4, Completed

print("\n=== Example with neither ===")
for i in range(5):
    print(i)
else:
    print("Completed")  # Will print
# Output: 0, 1, 2, 3, 4, Completed
```

Part 4: FOR vs WHILE - When to Use Which? (30 minutes)

Decision Guide

Scenario	Use FOR Loop	Use WHILE Loop
Known number of iterations	✅ Yes	❌ No
Iterating over collection	✅ Yes	❌ No
Unknown iterations	❌ No	✅ Yes
Condition-based repetition	❌ No	✅ Yes
Using range()	✅ Yes	⚠️ Possible but awkward
User input validation	❌ No	✅ Yes
Counting with step	✅ Yes	⚠️ Possible but verbose

Example: Same Task, Different Loops

Task: Print numbers 0 to 4

```
# Using FOR loop (preferred for this)
print("Using FOR:")
for i in range(5):
    print(i)

# Using WHILE loop (more verbose)
print("\nUsing WHILE:")
i = 0
while i < 5:
    print(i)
    i += 1
```

Task: Keep asking until valid input

```
# Using WHILE loop (preferred for this)
print("Using WHILE:")
age = -1
while age < 0 or age > 120:
    age = int(input("Enter valid age (0-120): "))
```

```
print(f"Age: {age}")

# Using FOR loop (awkward, not recommended)
print("\nUsing FOR:")
for _ in range(1000):  # Arbitrary large number
    age = int(input("Enter valid age (0-120): "))
    if 0 <= age <= 120:
        break
print(f"Age: {age}")
```

Converting Between FOR and WHILE

```
# FOR loop
for i in range(1, 6):
    print(i)

# Equivalent WHILE loop
i = 1
while i < 6:
    print(i)
    i += 1
```

Common Pitfalls and Debugging (30 minutes)

Pitfall 1: The Infinite Loop

```
# ❌ Problem: Infinite loop
x = 0
while x < 10:
    print(x)
    # Forgot to increment x!

# ✅ Solution: Always update condition variable
x = 0
while x < 10:
    print(x)
    x += 1
```

How to Stop Infinite Loops:

- Press Ctrl + C in terminal
- Click "Stop" button in IDE
- Close the Python window (last resort)

Pitfall 2: Off-by-One Errors

```
# ❌ Problem: Misses last element
items = ["a", "b", "c", "d", "e"]
for i in range(len(items) - 1):  # Goes 0 to 3
    print(items[i])
# Output: a, b, c, d (missing 'e')

# ✅ Solution: Use correct range
items = ["a", "b", "c", "d", "e"]
for i in range(len(items)):  # Goes 0 to 4
    print(items[i])
# Output: a, b, c, d, e

# ✅ Better: Iterate directly
for item in items:
    print(item)
```

Pitfall 3: Modifying List During Iteration

```
# ❌ Problem: Unpredictable behavior
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num % 2 == 0:
        numbers.remove(num)
print(numbers)  # May not remove all even numbers!

# ✅ Solution 1: Iterate over copy
numbers = [1, 2, 3, 4, 5]
for num in numbers[:]:
    if num % 2 == 0:
        numbers.remove(num)
```

```
print(numbers)  # [1, 3, 5]

# ✅ Solution 2: Build new list
numbers = [1, 2, 3, 4, 5]
odd_numbers = [num for num in numbers if num % 2 != 0]
print(odd_numbers)  # [1, 3, 5]
```

Pitfall 4: Wrong Indentation

```
# ❌ Problem: Code outside loop runs every iteration
for i in range(3):
    print("Inside loop")
print("This should be outside")  # Wrong indentation!

# ✅ Solution: Proper indentation
for i in range(3):
    print("Inside loop")
print("This is outside")  # Correct!
```

Pitfall 5: Using = Instead of ==

```
# ❌ Problem: Assignment instead of comparison
x = 5
while x = 10:  # SyntaxError
    print(x)

# ✅ Solution: Use comparison operator
x = 5
while x == 10:  # Comparison
    print(x)
```

Nested Loops (60 minutes)

What Are Nested Loops?

A nested loop is a loop inside another loop. The inner loop completes all its iterations for each iteration of the outer loop.

Real-World Analogy: Think of a clock:

- Outer loop: Hours (1-12)
- Inner loop: Minutes (0-59)

For each hour, all 60 minutes must pass.

Basic Structure

```
for outer_variable in outer_sequence:
    # Outer loop code
    for inner_variable in inner_sequence:
        # Inner loop code
        # This runs multiple times per outer iteration
    # Back to outer loop
```

Example 1: Multiplication Table

```
# Print multiplication table from 1 to 5
for i in range(1, 6):  # Outer loop: multiplicand
    for j in range(1, 6):  # Inner loop: multiplier
        result = i * j
        print(f"{i} × {j} = {result:2d}", end="  ")
    print()  # New line after each row

# Output:
# 1 × 1 =  1  1 × 2 =  2  1 × 3 =  3  1 × 4 =  4  1 × 5 =  5
# 2 × 1 =  2  2 × 2 =  4  2 × 3 =  6  2 × 4 =  8  2 × 5 = 10
# 3 × 1 =  3  3 × 2 =  6  3 × 3 =  9  3 × 4 = 12  3 × 5 = 15
# ... and so on
```

Example 2: Pattern Printing

```
# Print a right triangle pattern
rows = 5

for i in range(1, rows + 1):  # Outer loop: rows
    for j in range(i):  # Inner loop: stars in each row
        print("*", end="")
```

```
    print()  # New line after each row

# Output:
# *
# **
# ***
# ****
# *****
# Print a pyramid pattern
rows = 5

for i in range(1, rows + 1):
    # Print spaces
    for j in range(rows - i):
        print(" ", end="")

    # Print stars
    for k in range(2 * i - 1):
        print("*", end="")

    print()  # New line

# Output:
#     *
#    ***
#   *****
#  *******
# *********
```

Understanding Execution Flow

```
# Demonstrating nested loop execution
print("Execution flow:")
for i in range(1, 4):  # Outer: 1, 2, 3
    print(f"\nOuter loop: i = {i}")
    for j in range(1, 3):  # Inner: 1, 2
        print(f"  Inner loop: j = {j}")
```

```
        print(f"  Processing: i={i}, j={j}")

# Output:
# Outer loop: i = 1
#   Inner loop: j = 1
#   Processing: i=1, j=1
#   Inner loop: j = 2
#   Processing: i=1, j=2
# Outer loop: i = 2
#   Inner loop: j = 1
#   Processing: i=2, j=1
#   Inner loop: j = 2
#   Processing: i=2, j=2
# Outer loop: i = 3
#   Inner loop: j = 1
#   Processing: i=3, j=1
#   Inner loop: j = 2
#   Processing: i=3, j=2
```

Break and Continue in Nested Loops

```
# Break only exits the innermost loop
for i in range(1, 4):
    print(f"\nOuter loop: {i}")
    for j in range(1, 4):
        if j == 2:
            break  # Only exits inner loop
        print(f"  Inner: {j}")

# Output:
# Outer loop: 1
#   Inner: 1
# Outer loop: 2
#   Inner: 1
# Outer loop: 3
#   Inner: 1
# Using flag to break outer loop
found = False
```

```
for i in range(1, 6):
    for j in range(1, 6):
        if i * j == 12:
            print(f"Found: {i} × {j} = 12")
            found = True
            break
    if found:
        break  # Exit outer loop too
```

Loop Optimization Techniques (45 minutes)

Technique 1: Early Exit with Break

```
# Without optimization - checks all items
numbers = [1, 5, 10, 15, 20, 25, 30]
target = 15

for num in numbers:
    if num == target:
        print(f"Found {target}")
    # Continues even after finding target

# With optimization - exits early
for num in numbers:
    if num == target:
        print(f"Found {target}")
        break  # Stop searching once found
```

Technique 2: Skip Unnecessary Iterations with Continue

```
# Process only valid data
data = [10, -5, 20, 0, -15, 30, 45]

# Without optimization
total = 0
for num in data:
    if num > 0:
```

```
        total += num
print(f"Sum of positives: {total}")

# With optimization using continue
total = 0
for num in data:
    if num <= 0:
        continue  # Skip negative and zero
    total += num
print(f"Sum of positives: {total}")
```

Technique 3: Avoid Redundant Operations

```
# Without optimization - calculates length every iteration
my_list = list(range(1000))
for i in range(len(my_list)):  # len() called once
    print(my_list[i])

# Better - direct iteration
for item in my_list:
    print(item)

# With optimization - store length if needed
length = len(my_list)
for i in range(length):
    print(my_list[i])
```

Technique 4: List Comprehensions (Advanced)

```
# Traditional loop
squares = []
for i in range(10):
    squares.append(i ** 2)

# Optimized with list comprehension
squares = [i ** 2 for i in range(10)]

# With condition
```

```
even_squares = [i ** 2 for i in range(10) if i % 2 == 0]
```

Zip for Parallel Iteration

```
# Without zip - using indices
names = ["Alice", "Bob", "Charlie"]
scores = [85, 92, 78]

for i in range(len(names)):
    print(f"{names[i]}: {scores[i]}")

# With zip - cleaner and more efficient
for name, score in zip(names, scores):
    print(f"{name}: {score}")
```