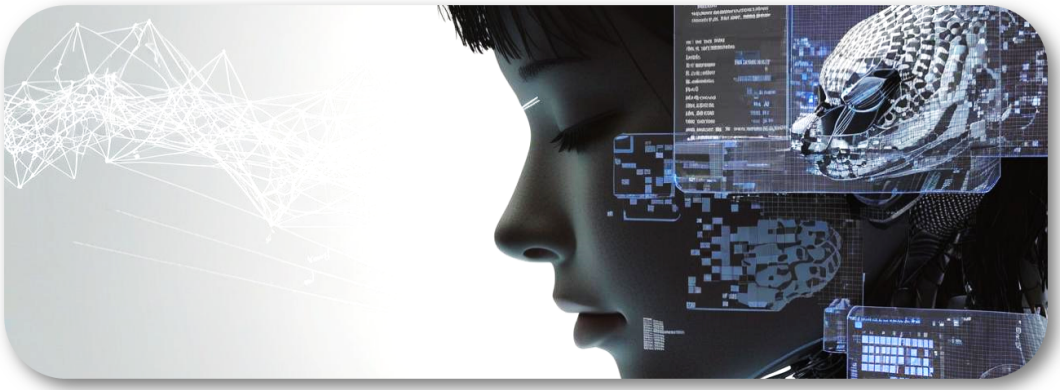

COMP 1023 Introduction to Python Programming

Tutorial 5: Looping

COMP 1023 Teaching Team

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology, Hong Kong SAR, China

Why Looping (or Iteration)?

- Many programming tasks are repetitive: we need to perform the same operation multiple times with only small changes (e.g. a counter, an index, an input value).
- Instead of writing the same statement 100 times, we use a loop construct to tell Python: "repeat this block a controlled number of times".
- In Python there are two primary loop statements:
  - `while` loop / iterative statement (repeat while a condition stays `True`)
  - `for` loop / iterative statement (iterate over a sequence / iterable of values)
- Both allow optional use of `break` (stop early) and `continue` (skip to next iteration), and can have an `else` clause in Python for special post-loop logic.

Why Looping (or Iteration)? (Cont'd)

- Loops make code shorter, more maintainable, and less error-prone.
- In the lecture you saw an example printing something 100 times. Manually duplicating the line would be tedious and brittle; a loop expresses the intention clearly.

Manual vs Loop Example

Manual (not recommended):

```
print("Best COMP course!")
print("Best COMP course!")
... (repeated 98 more times) ...
```

Using a loop:

```
for i in range(100):
    print("Best COMP course!")
```

Warm-up: iPRS Question 1

There are two categories of loops introduced in class: Counter-controlled loops and Sentinel-controlled loops. Which of the following statements is CORRECT?

A. Counter-controlled loops repeat until a specific condition is met, while Sentinel-controlled loops repeat a fixed number of times.

B. Counter-controlled loops repeat a fixed number of times, while Sentinel-controlled loops repeat until a specific condition is met.

C. Counter-controlled loops are implemented using `while` loops, while Sentinel-controlled loops are implemented using `for` loops.

D. None of the above.

Warm-up: iPRS Question 1 - Answer

A. Counter-controlled loops repeat until a specific condition is met, while Sentinel-controlled loops repeat a fixed number of times.

B. Counter-controlled loops repeat a fixed number of times, while Sentinel-controlled loops repeat until a specific condition is met.

C. Counter-controlled loops are implemented using `while` loops, while Sentinel-controlled loops are implemented using `for` loops.

D. None of the above.

Answer: B

- Refer to definitions in lecture slides.
- Note there are no restrictions on which loop type (`for` or `while`) to use for either category. Even `for` loop can be used for sentinel-controlled loops. E.g. iterating until a sentinel value is entered by user (you will learn this in later lectures):

```
for s in iter(lambda: input("Enter value (END to stop): "), "END"):
    print(f"Got: {s}")
```

Demo 1: For-Else with Break and Continue

Recap:

- `for-else` structure allows an `else` block after the loop. Similarly, there are `while-else` structures.
- `break` exits the innermost loop immediately.
- `continue` skips the rest of the current loop iteration and proceeds to the next iteration.

Question: What will the following code print?

```
for i in range(5):
    if i == 3:
        break
    print(i, end=' ')
else:
    print("Done", end=' ')
print("End")
```

Demo 1: For-Else with Break and Continue (Cont'd)

Answer

Output: `0 1 2 End`

- `else` clause only runs if loop completes normally (no `break`)
- Loop breaks when `i == 3`, so `else` is skipped
- "Done" is never printed
- "End" always prints (outside the for-else structure)

iPRS Question 2

Following the previous example, if we changed `break` to `continue`, what will the code print?

```
for i in range(5):
    if i == 3:
        continue
    print(i, end=' ')
else:
    print("Done", end=' ')
print("End")
```

A. 0 1 2 4 Done End

B. 0 1 2 4 End

C. 0 1 2 4 Done

D. 0 1 2 4

iPRS Question 2 - Answer

```
for i in range(5):
    if i == 3:
        continue
    print(i, end=' ')
else:
    print("Done", end=' ')
print("End")
```

A. 0 1 2 4 Done End

B. 0 1 2 4 End

C. 0 1 2 4 Done

D. 0 1 2 4

Answer: A

- `continue` skips only the current iteration (when `i == 3`), so 3 is not printed

Demo 2: Nested Loop with Continue and Break

Question: What will the following code print?

```
for i in range(4):
    for j in range(4):
        if i == j:
            continue
        elif i + j == 3:
            break
        print(f"({i},{j})", end=" ")
    print()
print(f"After all loops, i={i}, j={j}")
```

Demo 2: Nested Loop with Continue and Break (Cont'd)

Answer

Output:

```
(0,1) (0,2)
(1,0)
(2,0)

After all loops, i=3, j=0
```

Recap:

- `continue` and `break` affect the innermost loop only (i.e., the loop they are directly inside)
- Skips when `i == j`: (0,0), (1,1), (2,2); these pairs are not printed
- End inner loop when `i + j == 3`: (0,3), (1,2), (2,1), (3,0)
- Prints all remaining off-diagonal coordinates
- Outer loop continues normally after inner loop completes
- Variable `i` and `j` retain their last values after loops finish

Demo 3 (Try it Yourself): While Loop with Multiple Conditions

Question: What prints?

```
x, y = 1, 10
while x < 5 and y > 0:
    print(f"{x}:{y}", end=' ')
    x += 1
    y -= 2
    if x == 3:
        y = 0
print(f" Final: {x},{y}")
```

Demo 3: While Loop with Multiple Conditions (Cont'd)

```
x, y = 1, 10
while x < 5 and y > 0:
    print(f"{x}:{y}", end=' ')
    x += 1; y -= 2
    if x == 3: y = 0
print(f"Final:{x},{y}")
```

Answer

Output: `1:10 2:8 Final:3,0`

- Iteration 1: x=1, y=10, both conditions true, print "1:10"
- Update: x=2, y=8
- Iteration 2: x=2, y=8, both conditions true, print "2:8"
- Update: x=3, y=6, then y=0 (special condition)
- Next check: x=3, but y=0, loop ends

Using Loops to Solve the Problems

In the following slides, we will explore different methods to solve two problems using loops in Python:

- **Example 4:** Counting numbers divisible by 7 but not by 5 in a specified range.
- **Example 5:** Counting 3-digit narcissistic numbers (Armstrong numbers).

Key Concepts:

- Using `for` loops to iterate over a range of numbers.
- Applying conditional statements to filter numbers based on specific criteria.
- Using counters to track quantities instead of storing values.

Example 4: Number Counting with Conditions

Programming Problem: Write a program to count how many numbers are divisible by 7 but not multiples of 5, between 2000 and 3000 (both inclusive). The program should print the total count.

Requirements:

- Range: $2000 \leq \text{number} \leq 3000$
- Divisible by 7: `number % 7 == 0`
- Not divisible by 5: `number % 5 != 0`
- Output format: print the total count. E.g., "`Total count: X`"

iPRS Question 3

Problem: Count numbers divisible by 7 but not by 5, between 2000-3000 (inclusive). Consider the code snippet below. There is a bug in one line. Identify it.

```
1 count = 0
2 for num in range(2000, 3000):
3     if num % 7 == 0 and num % 5 != 0:
4         count += 1
5 print(f"Total count: {count}")
```

A. Line 2: `for num in range(2000, 3000):`

B. Line 3: `if num % 7 == 0 and num % 5 != 0:`

C. Line 4: `count += 1`

D. Line 5: `print(f"Total count:  {count}")`

iPRS Question 3 - Answer

Write a program to count how many numbers are divisible by 7 but not multiples of 5, between 2000 and 3000 (both inclusive).
Consider the code snippet below. There is a bug in one line. Identify it.

```
count = 0
for num in range(2000, 3000): # Bug here: should be range(2000, 3001)
    if num % 7 == 0 and num % 5 != 0:
        count += 1
print(f"Total count: {count}")
```

Answer: A

This line should be "`for num in range(2000, 3001):`" to include 3000 in the range.

Example 4: Advanced Solutions (FYI - Preview of Lists)

Note: The following solutions use lists, which you will learn soon.

Solution 2: Using List Comprehension (FYI)

```
# Method 2: List comprehension (more Pythonic)
numbers = [num for num in range(2000, 3001)
           if num % 7 == 0 and num % 5 != 0]
print(f"Total count: {len(numbers)}")
```

Solution 3: Using Filter

```
# Method 3: Using filter function
def is_valid(num):
    return num % 7 == 0 and num % 5 != 0
numbers = list(filter(is_valid, range(2000, 3001)))
print(f"Total count: {len(numbers)}")
```

Example 5: Narcissistic Numbers (Armstrong Numbers)

Programming Problem: Count how many narcissistic numbers exist in a given range. A narcissistic number is a number that is equal to the sum of its digits each raised to the power of the number of digits.

Examples:

- $153 = 1 ** 3 + 5 ** 3 + 3 ** 3 = 1 + 125 + 27 = 153$ (3-digit narcissistic number)
- $9474 = 9 ** 4 + 4 ** 4 + 7 ** 4 + 4 ** 4 = 6561 + 256 + 2401 + 256 = 9474$ (4-digit)
- $54748 = 5 ** 5 + 4 ** 5 + 7 ** 5 + 4 ** 5 + 8 ** 5 = 3125 + 1024 + 16807 + 1024 + 32768 = 54748$ (5-digit)

Task: Count how many 3-digit narcissistic numbers exist (range: 100-999).

Expected Output Format: "`Total 3-digit narcissistic numbers: 4`"

Example 5: Solution Method 1

Solution 1: Using String Conversion

```
def count_narcissistic_numbers(start, end):
    count = 0
    for num in range(start, end + 1):
        num_str = str(num)
        num_digits = len(num_str)
        # Calculate sum of digits raised to power of number of digits
        digit_sum = sum(int(digit) ** num_digits for digit in num_str)
        if digit_sum == num:
            count += 1
    return count
result = count_narcissistic_numbers(100, 999)
print(f"Total 3-digit narcissistic numbers: {result}")
```

Rationale:

- We convert the number to a string to easily access each digit.

Example 5: Solution Method 2

Solution 2: Using Mathematical Operations (Part 1)

```
def is_narcissistic(num):
    original = num
    # Count digits and extract them
    temp, digit_count = num, 0
    while temp > 0:
        digit_count += 1
        temp //= 10
    # Calculate sum of digits raised to power of number of digits
    temp, digit_sum = num, 0
    while temp > 0:
        digit = temp % 10
        digit_sum += digit ** digit_count
        temp //= 10
    return digit_sum == original
```

Example 5: Solution Method 2 (Cont'd)

Solution 2: Using Mathematical Operations (Part 2)

```
count = 0
for num in range(100, 1000):
    if is_narcissistic(num):
        count += 1
print(f"Total 3-digit narcissistic numbers: {count}")
```

Rationale:

- The is_narcissistic function counts digits and checks if the number is narcissistic without converting to string.
- Then we loop through the range 100-999 and use this function to count how many narcissistic numbers there are.

Example 5: Solution Method 3

Solution 3: Direct Calculation for 3-digit Numbers

```
# Method 3: Direct calculation for 3-digit numbers
def count_3digit_narcissistic():
    count = 0
    for i in range(1, 10):       # hundreds digit (1-9)
        for j in range(0, 10):  # tens digit (0-9)
            for k in range(0, 10):  # units digit (0-9)
                num = i * 100 + j * 10 + k
                if i**3 + j**3 + k**3 == num:
                    count += 1
    return count
result = count_3digit_narcissistic()
print(f"Total 3-digit narcissistic numbers: {result}")
```

Rationale:

- We directly iterate over all possible 3-digit combinations using nested loops.

Printing Triangle Patterns

Triangle patterns are common exercises to practice nested loops. We will explore several patterns and their implementations.

- Right Triangle (stars)
- Number Triangle
- Centered Triangle
- Inverted Triangle
- Diamond
- Pascal's Triangle

Example 6: Printing Triangle Patterns

Programming Problem: Write programs to print different triangle patterns using nested loops.

Pattern 1 - Right Triangle (stars):

```
*
**
***
****
*****
```

Pattern 2 - Number Triangle:

```
1
12
123
1234
12345
```

Example 6: Solution - Pattern 1 & 2

Pattern 1: Right Triangle (Stars)

```
def print_star_triangle(n):
    for i in range(1, n + 1):
        for j in range(i):
            print('*', end='')
        print()  # New line after each row
print_star_triangle(5)
```

Pattern 2: Number Triangle

```
def print_number_triangle(n):
    for i in range(1, n + 1):
        for j in range(1, i + 1):
            print(j, end='')
        print()  # New line after each row
print_number_triangle(5)
```

Example 6: Pattern 3

Programming Problem: Write a program to print a centered triangle pattern.
Pattern 3 - Centered Triangle:

```
    *
   ***
  *****
 *******
*********
```

Example 6: Solution - Pattern 3

Pattern 3: Centered Triangle

```
# Pattern 3: Centered triangle
def print_centered_triangle(n):
    for i in range(1, n + 1):
        # Print leading spaces
        for j in range(n - i):
            print(' ', end='')

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

        print()  # New line after each row

print_centered_triangle(5)
```

iPRS Question 4 - Alternative Solution for Pattern 3

Will the following code produce the same output as the previous solution for Pattern 3 (Centered Triangle)?

```
# More concise version using string operations
def print_centered_triangle_v2(n):
    for i in range(1, n + 1):
        spaces = ' ' * (n - i)
        stars = '*' * (2 * i - 1)
        print(spaces + stars)
print_centered_triangle_v2(5)
```

```
    *
   ***
  *****
 *******
*********
```

Will it produce the same shape as shown on the right?

A. Yes, it produces the same shape.

B. No, it produces a different shape.

iPRS Questions 4 - Answer

Alternative: Using String Multiplication

```python
# More concise version using string operations
def print_centered_triangle_v2(n):
    for i in range(1, n + 1):
        spaces = ' ' * (n - i)
        stars = '*' * (2 * i - 1)
        print(spaces + stars)
print_centered_triangle_v2(5)
```

```
    *
   ***
  *****
 *******
*********
```

A. Yes, it produces the same shape.

B. No, it produces a different shape.

Answer: A

Both implementations produce the same centered triangle pattern. The second version uses string multiplication for conciseness.

Extended Reading: More Triangle Patterns

Programming Problem: Write programs to print these more complex patterns.

Pattern 4 - Inverted Triangle:

```
*****
****
***
**
*
```

Pattern 5 - Diamond:

```
    *
   ***
  *****
*******
  *****
   ***
    *
```

Extended Reading: Solution - Pattern 4

Pattern 4: Inverted Triangle

```
# Pattern 4: Inverted triangle
def print_pattern4(n):
    for i in range(n, 0, -1):
        for j in range(i):
            print('*', end='')
        print()

print_pattern4(5)
```

Extended Reading: Solution - Pattern 5

Pattern 5: Diamond

```python
# Pattern 5: Diamond (combination of upward and downward triangles)
def print_pattern5(n):
    # Upper half (including middle)
    for i in range(1, n + 1):
        spaces = ' ' * (n - i)
        stars = '*' * (2 * i - 1)
        print(spaces + stars)
    # Lower half
    for i in range(n - 1, 0, -1):
        spaces = ' ' * (n - i)
        stars = '*' * (2 * i - 1)
        print(spaces + stars)
print_pattern5(4)  # Creates a 7-row diamond
```

Extended Reading: Pattern 6

Programming Problem: Write a program to print Pascal's Triangle.
Pattern 6 - Pascal's Triangle:

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

Extended Reading: Solution - Pattern 6

Pattern 6: Pascal's Triangle

```python
# Pattern 6: Pascal's triangle using combinatorial formula
def print_pattern6(n):
    for i in range(n):
        # Print leading spaces
        print(' ' * (n - i - 1), end='')

        # Calculate and print Pascal's triangle numbers
        num = 1
        for j in range(i + 1):
            print(f"{num}", end=' ')
            num = num * (i - j) // (j + 1)  # Calculate next number
        print()  # New line
print_pattern6(5)
```

That's all!

Any questions?