COMP 1023 Introduction to Python Programming
Looping Statements

Dr. Cecia Chan, Prof. SC Cheung, Dr. Alex Lam, Dr. Desmond Tsoi

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

Introduction

- Suppose that you need to display a string, e.g.,

  `COMP 1023 is the best COMP course!`

  a hundred times. It would be tedious to have to type the statement a hundred times:

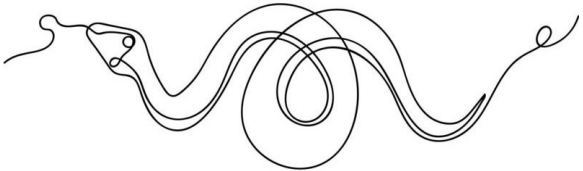
```
print("COMP 1023 is the best COMP course!") # 1st time
print("COMP 1023 is the best COMP course!") # 2nd time
# ...
print("COMP 1023 is the best COMP course!") # 100th time
```

- Instead, it would be handy to have a construct that controls how many times a sequence of operations is performed. For example:

```
count = 0
while count < 100:
    print("COMP 1023 is the best COMP course!")
    count += 1
```

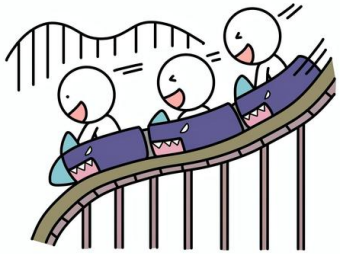
Why Looping (or Iteration)?

- Many tasks within a program are repetitive, and programming languages typically provide constructs that allow us to control how many times a statement or a block of statements should be executed.
- Those constructs are called loops.
- In Python, there are two types of loops:
  - while loop/iterative statement
  - for loop/iterative statement



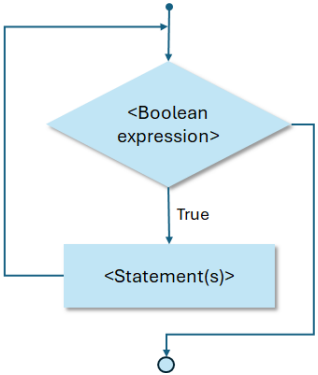
Looping (or Iteration)

- A loop has 4 components:
  1. Initialize
     Initialize the loop control variable.
  2. Test condition
     Evaluate the test condition (a Boolean expression).
  3. Loop body
     The statements in the loop are executed if the test condition evaluates to True.
  4. Update
     Typically, the loop control variable is updated/modified through the execution of the loop body. It can then go through the test condition again.



while Loop

- A while loop/statement allows you to execute a statement or a block of statements repeatedly as long as a condition (represented by the Boolean expression) is evaluated to True (i.e., satisfied).



Syntax

```
while <Boolean expression>:
    <statement 1>
    ...
    <statement N>
```

where `<Boolean expression>` is an expression that returns a Boolean result, and `<statement 1>`, ..., `<statement N>` denote the program statements that need to be executed when the `<Boolean expression>` is evaluated to True.

Types of Loops

There are two categories of loops:

1. Counter-controlled loops
   The loop body is repeated a specific number of times, and the number of repetitions is known before the loop starts execution.

2. Sentinel-controlled loops
   The number of repetitions is not known before the loop starts execution. Usually, a sentinel value (such as -1, which is different from regular data) is used to determine whether to execute the loop body.

while Loop (Counter-Controlled Loop)

```python
# Filename: counter-controlled_loop_mark.py

def main():
    total, mark = 0.0, 0.0
    counter = 0  # initialize

    while counter < 10:  # test condition
        # loop body
        mark = float(input("Enter mark: "))
        total += mark
        counter += 1  # update

    average = total / counter
    print("Average = ", average)

if __name__ == "__main__":
    main()
```

Output:

```
Enter mark: 52
Enter mark: 79
Enter mark: 67
Enter mark: 21
Enter mark: 56
Enter mark: 11
Enter mark: 99
Enter mark: 41
Enter mark: 69
Enter mark: 47
Average =  54.2
```

In every loop, there must be a point in the loop body to make the test condition become false; otherwise, it will result in an infinite loop.

while Loop (Sentinel-Controlled Loop)

```python
# Filename: sentinel_controlled_loop_mark.py
def main():
    total, mark = 0.0, 0.0  # initialize
    counter = 0
    mark = float(input("Enter mark: "))

    while mark != -1:  # test condition
        # loop body
        total += mark
        counter += 1
        # update
        mark = float(input("Enter mark: "))

    if counter != 0:
        average = total / counter
        print("Average =", average)

if __name__ == "__main__":
    main()
```

Output:

```
Enter mark: 52
Enter mark: 79
Enter mark: 67
Enter mark: 21
Enter mark: 56
Enter mark: 11
Enter mark: 99
Enter mark: 41
Enter mark: 69
Enter mark: 47
Enter mark: -1
Average = 54.2
```

The number of marks to be entered depends on when the sentinel value -1 is entered.

Numeric Errors

- Let's try to sum a series that starts with 0.01 and ends with 1.0. The numbers in the series will increment by 0.01, as follows: 0.01 + 0.02 + 0.03 and so on.

```
# Filename: numeric_errors.py

def main():
    total_sum = 0
    # Add 0.01, 0.02, ..., 0.99, 1 to total_sum
    i = 0.01              # Initialize
    while i <= 1.0:       # Test condition
        total_sum += i    # Loop body
        i = i + 0.01      # Update
    # Display result
    print("The total sum is", total_sum)

if __name__ == "__main__":
    main()
```

Floating-point numbers are represented by approximation.

Output:

```
The sum is 49.50000000000003
```

- 0.01 + 0.02 + 0.03 + ... + 0.99 + 1 should be 50.5. How come the result displayed is 49.5?
- Since when the loop ends, the value of i is slightly larger than 1 (not exactly 1), which causes the last i value not to be added into the total_sum.

How to fix the problem?

for Loop

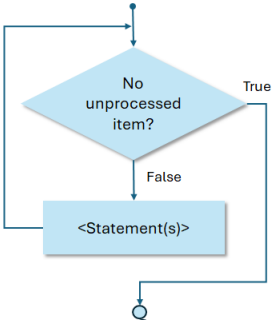
- The for statement is another way of writing repetition logic. It aims to handle counter-controlled loops.

Syntax

```
for <variable>
    in range(<initial value>,
             <end value>,
             <step value>):
    <statement 1>
    ...
    <statement N>

for <variable> in <sequence>:
    <statement 1>
    ...
    <statement N>
```

where `<variable>` is the control variable, `<initial value>` is its starting value (optional, default = 0), `<end value>` is the stopping value (exclusive), and `<step value>` is the increment (optional, default = 1). The statements `<statement 1>`, ..., `<statement N>` are executed for each value. `<sequence>` is a collection of items stored in order, such as strings, lists, and tuples.

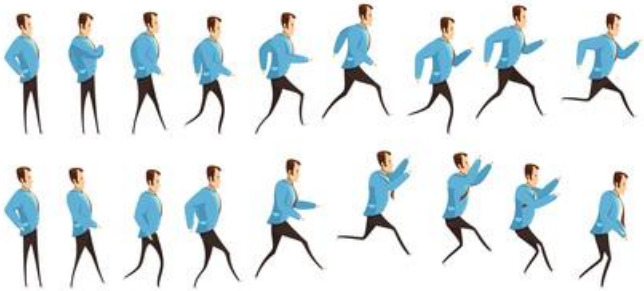


range() Function

```
for i in range(5):
    print(i, end=" ")  # Print 0 1 2 3 4
for i in range(4, 8):
    print(i, end=" ")  # Print 4 5 6 7
for i in range(3, 9, 2):
    print(i, end=" ")  # Print 3 5 7
for i in range(5, 1, -1):
    print(i, end=" ")  # Print 5 4 3 2
```

Note

The numbers in the range function must be integers. For example, `range(1.5, 8.5)`, or `range(1.5, 8.5, 1)` would be incorrect.



while Loop vs for Loop

```python
# Filename: while_loop.py
def main():
    total = 0.0
    counter = 0
    while counter < 10:
        mark = float(input("Enter mark: "))
        total += mark
        counter += 1
    average = total / counter
    print("Average =", average)

if __name__ == "__main__":
    main()
```

```python
# Filename: for_loop.py
def main():
    total = 0.0
    counter = 0
    for i in range(10):
        mark = float(input("Enter mark: "))
        total += mark
        counter += 1
    average = total / counter
    print("Average =", average)

if __name__ == "__main__":
    main()
```

for Loop

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$$

```python
# Filename: factorial.py

def main():
    n = int(input("Enter value of n: "))
    result = n
    for i in range(n-1, 0, -1):
        result *= i
    print("The factorial of", n, "is", result)

if __name__ == "__main__":
    main()
```

Output:

```
Enter value of n: 5
The factorial of 5 is 120
```

Output:

```
Enter value of n: 8
The factorial of 8 is 40320
```

Output:

```
Enter value of n: 12
The factorial of 12 is 479001600
```

Which Loop Do You Use?

- while loop
  Most appropriate for sentinel-controlled loops.
- for loop
  Most appropriate for a fixed number of repetitions (i.e., counter-controlled loops).

break Statement

- The break statement alters the flow inside loops, i.e., `while` and `for` loops.
- Execution of `break` causes immediate termination of the innermost enclosing loop.

```
def main():  # Filename: break_rect.py
    area = 100.0
    while area > 50.0:
        length = float(input("Enter length of rect: "))
        if length < 5.0:
            break
        width = float(input("Enter width of rect: "))
        if width < 5.0:
            break
        area = length * width
        print("The area =", area)

if __name__ == "__main__":
    main()
```

Output:

```
Enter length of rect: 10
Enter width of rect: 20
The area = 200.0
Enter length of rect: 10
Enter width of rect: 3
```

The while loop terminates if the input value of width or length is less than 5.

Execute the Loop At Least Once

- In some programming languages, a "do-while" loop ensures that the code within the loop executes at least once before checking the condition.
- It is useful to repeatedly prompt a user for input until they provide a valid response.
- In Python, there is no "do-while" loop, but it can be emulated using a while loop with a True condition and a strategically placed break statement.

```python
# Filename: valid_input.py
def main():
    while True:
        user_input = input("Enter a positive number: ")
        if user_input.isdigit() and int(user_input) > 0:
            break
        print("Invalid input. Please try again.")
    print("Valid input received.")

if __name__ == "__main__":
    main()
```

Output:

```
Enter a positive number: -5
Invalid input. Please try again.
Enter a positive number: -1023
Invalid input. Please try again.
Enter a positive number: a
Invalid input. Please try again.
Enter a positive number: 100
Valid input received.
```

continue Statement

- The `continue` statement can be used in loops, such as `while` and `for` loops.
- When a `continue` statement is encountered, control jumps to the beginning of the nearest enclosing loop for the next iteration.
- All subsequent statements after the `continue` statement are not executed for that particular iteration.

continue Statement

```python
# Filename: skip_negative_num.py

def main():
    total = 0
    print("Enter 8 numbers")
    for i in range(8):
        data = int(input("Enter number {}: ".format(i + 1)))
        if data < 0:
            continue
        total += data
    print("The sum =", total)

if __name__ == "__main__":
    main()
```

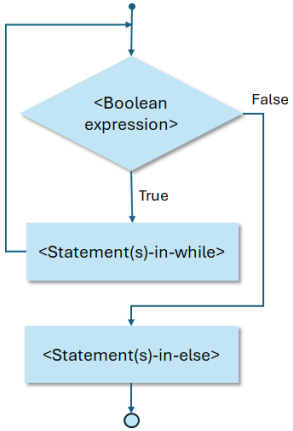
Output:

```
Enter 8 numbers
Enter number 1: 1
Enter number 2: 2
Enter number 3: 3
Enter number 4: 4
Enter number 5: -5
Enter number 6: 6
Enter number 7: -7
Enter number 8: 8
The sum = 24
```

If the input data is negative, control is immediately passed to the next iteration of the loop, and the subsequent statement `total += data` is not executed.

while-else Loop

- A while-else loop functions like a regular while loop but includes an additional `else` block that executes when the while condition becomes false, provided the loop ends normally without hitting a `break` statement.



Syntax

```
while <Boolean expression>:
    <statement A1>
    ...
    <statement AN>
else:
    <statement B1>
    ...
    <statement BN>
```

where `<Boolean expression>` is an expression that returns a Boolean result. `<statement A1>`, ..., `<statement AN>` are executed when the `<Boolean expression>` is True, and `<statement B1>`, ..., `<statement BN>` are executed when it is False.

Example

```python
def main():    # Filename: valid_input_max_attempts.py
    max_attempts = 3
    attempts = 0

    while attempts < max_attempts:
        user_input = input("Enter a valid number: ")

        if user_input.isdigit():
            print(f"Valid input: {user_input}")
            break
        else:
            print("Invalid input.")
            attempts += 1
    else:
        print("Exceeded maximum attempts", max_attempts,
              "\nExiting program.")

if __name__ == "__main__":
    main()
```

Output:

```
Enter a valid number: abc
Invalid input.
Enter a valid number: def
Invalid input.
Enter a valid number: ghi
Invalid input.
Exceeded maximum attempts 3
Exiting program.
```

Output:

```
Enter a valid number: abc
Invalid input.
Enter a valid number: 123
Valid input: 123
```

for-else Loop

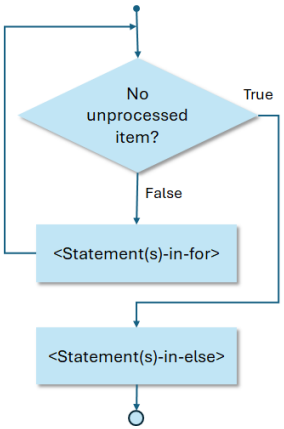
- A for-else loop functions like a regular for loop but includes an additional `else` block that executes when the for loop completes its iterations, provided the loop ends normally without hitting a `break` statement.

Syntax

```
for <variable> in range(<initial value>,
                        <end value>,
                        <step value>):
    <statement A1>
    ...
else:
    <statement B1>
    ...

for <variable> in <sequence>:
    <statement A1>
    ...
else:
    <statement B1>
    ...
```

where `<variable>` is the control variable, `<initial value>` is its starting value (optional, default = 0), `<end value>` is the stopping value (exclusive), `<step value>` is how much the variable increases (optional, default = 1). The statements `<statement A1>`, ... are executed for each value. The statements `<statement B1>`, ... run after the for loop finishes. `<sequence>` is a collection of items stored in order, like a string of characters, or lists and tuples.



Example

```python
# Filename: check_prime.py

def main():
    number = int(input("Enter a number: "))

    for i in range(2, number):
        if number % i == 0:
            print(number, "is not a prime number.")
            break
    else:
        print(number, "is a prime number.")

if __name__ == "__main__":
    main()
```

Output:

```
Enter a number: 13
13 is a prime number.
```

Output:

```
Enter a number: 10
10 is not a prime number.
```

Nested Loops

- A loop may appear inside another loop; this is called a nested loop.
- We can define as many levels of loops as the hardware allows.
- Different types of loops can also be nested.
- For example, printing a multiplication table:

	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

Nested Loops

```python
# Filename: multiplication_table.py

def main():
    for i in range(1, 10):
        for j in range(1, 10):
            # rjust returns a 3-character long,
            # right-justified version of the
            # string representation of i*j
            print(str(i * j).rjust(3), end=" ")
        print()

if __name__ == "__main__":
    main()
```

Output:

```
  1   2   3   4   5   6   7   8   9
  2   4   6   8  10  12  14  16  18
  3   6   9  12  15  18  21  24  27
  4   8  12  16  20  24  28  32  36
  5  10  15  20  25  30  35  40  45
  6  12  18  24  30  36  42  48  54
  7  14  21  28  35  42  49  56  63
  8  16  24  32  40  48  56  64  72
  9  18  27  36  45  54  63  72  81
```

Nested Loops

```python
def main():  # Filename: pattern.py
    # Prompt the user for the height of the pattern
    height = int(input("Height of the pattern: "))

    # Outer loop for each line of the pattern
    for lines in range(1, height + 1):
        # Inner loop to print spaces before the stars
        for a in range(1, height - lines + 1):
            print(" ", end="")

        # Inner loop to print stars for the current line
        for b in range(1, 2 * lines):
            print("*", end="")

        # Move to the next line after printing stars
        print()

if __name__ == "__main__":
    main()
```

Output:

```
Height of the pattern: 10
          *
         ***
        *****
       *******
      *********
     ***********
    *************
   ***************
  *****************
*******************
```

Variable Reuse in Nested Loops

- If you use the same variable name for the loop variable in both an outer and an inner nested loop, the inner loop's assignment to that variable will overwrite the value from the outer loop.

```
for i in range(2):
    print("Outer loop: i =", i)
    for i in "OK": # A string is a sequence
        print("Inner loop: i =", i)
    # i will hold the last value from the inner loop
    print("Outer loop after inner: i =", i)
```

Output:

```
Outer loop: i = 0
Inner loop: i = O
Inner loop: i = K
Outer loop after inner: i = K
Outer loop: i = 1
Inner loop: i = O
Inner loop: i = K
Outer loop after inner: i = K
```

Python allows the same variable name to be used in nested scopes without creating a new variable. The inner loop's variable affects the outer variable.

Note: C++ maintains the scope of the outer variable, and the inner variable is treated as a new variable that exists only within the inner loop.

Variable Persistence After Loop

- The loop variable retains its last assigned value even after the loop has finished executing.

```
for i in range(3):
    print(i)
print(i)  # i will be 2
```

In Python, the loop variable is accessible after the loop ends and retains its value. Since variables defined within loops or branching statements are not limited to that block, they are accessible within the scope of the function or module in which they are defined.

In C++, the loop variable is scoped to the loop itself and cannot be accessed afterward.

Key Terms

- break statement
- continue statement
- counter-controlled loop
- for loop
- infinite loop
- initialize statement
- iteration statements
- loop body
- looping statements
- nested loops
- sentinel-controlled loop
- sentinel value
- test condition statement
- update statement
- while loop

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

1. There are two types of repetition statements: the _____ loop and the _____ loop in Python.
2. The part of the loop that contains the statements to be repeated is called the _____.
3. An _____ is a loop statement that executes infinitely.
4. The while loop checks the test conditions first. If the condition is _____, the loop body is executed; otherwise, the loop _____.
5. A _____ is a special value that signifies the end of the input.

Answer: 1. while; for, 2. loop body, 3. infinite loop, 4. True; terminates, 5. sentinel value.

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

6. The _____ loop is a counter-controlled loop and is used to execute a loop body a predictable number of times.
7. Two keywords, _____ and _____, can be used to change the default behaviors of a loop.
8. The _____ keyword immediately ends the innermost loop that contains itself/it belongs to.
9. The _____ keyword ends only the current iteration of the innermost loop that contains itself/it belongs to.
10. The _____ retains its last assigned value even after the loop has finished executing.

Answer: 6. for, 7. break; continue, 8. break, 9. continue, 10. loop variable

Further Reading

- Read Sections 5.1 - 5.13 of the textbook "Introduction to Python Programming and Data Structures".

That's all!

Any questions?