Fundamentos de Programação 2025-2026

Programming Fundamentals

Class #4 - Iteration

Overview

- Iteration
- Iteration in Python
  - The **`while`** statement
  - The **`for`** statement
  - **`Break`** and **`continue`** statements
  - Useful functions
    - **`range, enumerate`**
- Common uses and mistakes

PROBLEM

- Weather Station Logger
- Scenario:

  You're building a **basic weather station program**.

  Each day, a technician enters the temperature in Celsius.

  The system needs to convert it to Fahrenheit and display the result.

  This happens for 5 consecutive days.
- Task:
  - Write a program that:
    - Asks the user to input 5 temperatures in Celsius, one by one.
    - Converts each temperature to Fahrenheit.
    - Displays the converted Fahrenheit value for each input.

Using what we know …

```
BEGIN

   DISPLAY "Enter temperature in Celsius for Day 1:"
   READ celsius1
   SET fahrenheit1 TO (celsius1 * 9 / 5) + 32
   DISPLAY "Temperature in Fahrenheit for Day 1: " + fahrenheit1

   DISPLAY "Enter temperature in Celsius for Day 2:"
   READ celsius2
   SET fahrenheit2 TO (celsius2 * 9 / 5) + 32
   DISPLAY "Temperature in Fahrenheit for Day 2: " + fahrenheit2

   DISPLAY "Enter temperature in Celsius for Day 3:"
   READ celsius3
   SET fahrenheit3 TO (celsius3 * 9 / 5) + 32
   DISPLAY "Temperature in Fahrenheit for Day 3: " + fahrenheit3

   ...
```

Discussion of our solution

- What could be better?
- Problems ?

Much better solution

```
BEGIN
    SET number_of_days TO 5

    REPEAT FROM 1 TO number_of_days
      DISPLAY "Enter temperature in Celsius for Day " + day + ":"
      READ celsius_temperature
      SET fahrenheit_temperature TO (celsius_temperature * 9 / 5) + 32
      DISPLAY "Temp. Fahrenheit for Day " + day + ": " + fahrenheit_temperature
    END REPEAT
END
```

Needs

- How to repeat blocks of code?
- How to iterate from 1 to number_of_days ?

Iteration?

- Iteration means repeating a set of instructions until a condition is met or for a specific number of times.
- In programming, we use iteration to do something multiple times
  - like checking every item in a list, asking for input five times, or running a calculation again and again.

- Example: Imagine you want to ask five students their names.
  - Instead of writing five separate lines of code, you can use iteration to repeat the same instruction five times.

TYPICAL USES OF ITERATION

Fixed Repetition (Known Number of Times)

- Use case:
  - You know beforehand exactly how many times to repeat an action.

- Examples:
  - Printing 5 temperature conversions from Celsius to Fahrenheit.
  - Generating a multiplication table from 1 to 10.
  - Sending daily reminders for a week.

Conditional Repetition
(Unknown Number of Times)

- Use case:
  - **Repeat until a condition is met**.

- Examples:
  - Asking the user for a valid password until they enter it correctly.
  - Reading sensor data until a threshold is reached.
  - Processing items in a queue until it's empty.

Repetition Over Collections

- Use case:
  - Do something to every element in a list.

- Examples:
  - Converting a list of temperatures.
  - Summing up daily sales from a list.
  - Displaying names of registered participants.

Accumulation or Aggregation

- Use case:
  - Repeating to build up a result.

- Example:
  - Calculating the total cost of items in a cart.
  - Counting how many times a word appears in a text.
  - Averaging a series of grades.

Retry or Error Handling

- Use case:
  - **Repeat an operation if it fails**.
- Example:
  - Retrying a network request.
  - Asking for user input until it’s valid.
  - Reattempting a file read if the file is locked.

Simulation or Animation

- Use case:
  - Repeating steps to simulate time or movement.

- Example:
  - Simulating a soap dispenser dropping 10 drops.
  - Animating a bouncing ball.
  - Modeling population growth over time.

ITERATION IN PYTHON

The `while` statement

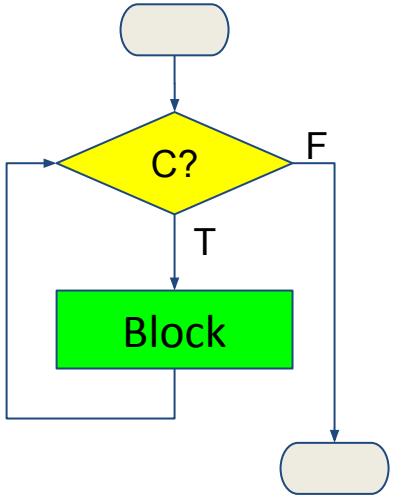
- The **`while`** statement tells Python to **repeatedly execute** some target statements for as long as a given condition is true.

Syntax

```
...
while condition:
    statements
...
```

Example

```
n = 3
while n > 0:
    print(n)
    n = n-1
print("Go!")
```

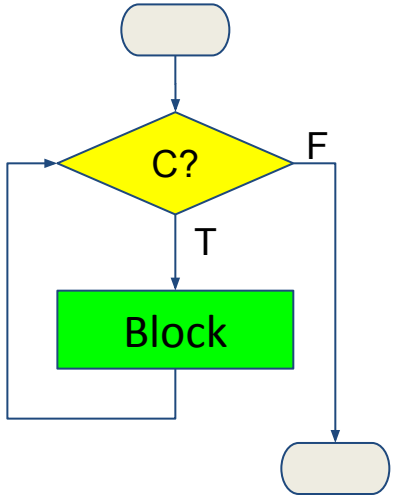


Play ▶️

The `while` statement (cont.)

Example

```
n = 3
while n > 0:
    print(n)
    n = n-1
print("Go!")
```



- If the `condition` is true, the `statements` are executed.
- The condition is re-evaluated, and if still true, the statements are repeated.
- When the condition becomes false, execution skips to the line immediately following the block of indented statements.

- The `condition` should be a Boolean expression.
  - Other types of expressions are implicitly converted to `bool`.
  - Any null or empty value means false.

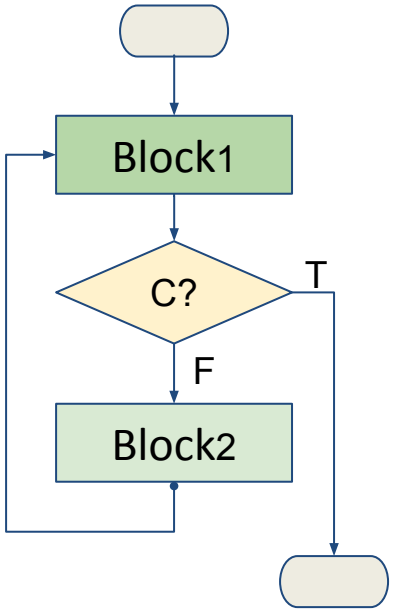
Loop control (1) - the **`break`** statement

- The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates.
  - Otherwise, the loop will repeat forever, which is called an infinite loop.

- Quite often the best place to decide if the loop should stop is halfway through the body.
  - In that case you can use the **`break`** statement to jump out of the loop.

```
while True:
    line = input('Enter text? ')
    if line == 'done':
        break
    print(line)
print('The end')
```

Play ▶️

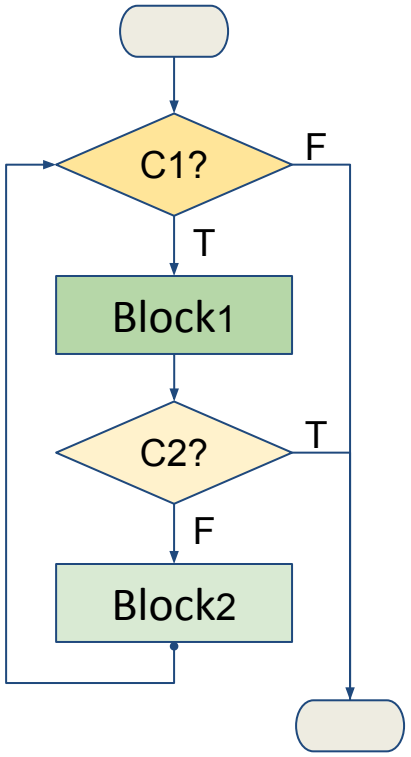


- A loop with this pattern is called a *loop-and-a-half*.

Multi-exit loops

- Sometimes there are **several conditions to terminate the loop** and multiple places to test them along the body of the loop.
- Use multiple **`if-break`** statements to achieve that.

```
while C1:
      Block1
      if C2: break
      Block2
      if C3: break
      Block3
...
```



The `for` statement

- Another loop mechanism is the **`for`** statement.
- It repeats statements **once for each item** in a collection of items,
  - such as a list, a string or a range

Syntax	Example

`...`
`for var in collection:`
`    statements`
`...`
	`print("Start")`
	`for n in [3, 1, 9]:`
	`    print(n)`
	`print("End")`

Play ▶️

The `for` statement

Syntax	Example

`...`
`for var in collection:`
`    statements`
`...`
	`print("Start")`
	`for n in [3, 1, 9]:`
	`    print(n)`
	`print("End")`

Play ▶️

- The collection is an expression, and it is evaluated first.
- Then, the first item in the collection is assigned to the iterating variable var,
  - and the statements block is executed once.
- Next, the next item is assigned to var, the statements are executed again,
  - and so on, until the end of the collection.

The `range()` function

- This built-in function returns an object that generates a sequence of integers in arithmetic progression.

```
list(range(4))  →  [0, 1, 2, 3]
```

- It is often used in **`for`** loops.

```
for n in range(0, 4):
    print(n)
```

- It may be called with 1, 2 or 3 arguments, as follows:
  - `range(stop)`
  - `range(start, stop)`
  - `range(start, stop, step)`
- All arguments must be integers (positive or negative).
- Generates integers up/down to, but not including, stop.

Loop control (2) - `continue` statement

- The **`continue`** statement **skips to the next iteration** of the enclosing loop body
  - without executing the remaining statements in the current iteration.
- As **`break`**, it can be used inside a loop body to change the normal flow of execution.
- Example:

```
for number in range(1, 10):
    if number % 2 == 0:
        continue  # Skip even numbers
    print(f"{number} is odd")
```

Play ▶️

The `else` clause

- The iteration statements may have an optional **`else`** clause.

```
count = 0
#count = 1        # uncomment to cause break
while count < 5:
    print(count, "is less than 5")
    if count==3: break
    count += 2
else:
    print(count, "is not less than 5")
print("END")
```

Play ▶️

- Statements in the **`else`** clause are executed only when the loop terminates without executing a **`break`**.

- WARNING: This feature is unusual, confusing, and seldom used. You may want to avoid it.
  - We describe it here for the sake of completeness.

Useful function - `enumerate`

- This built-in function generates **pairs of index and item** from an iterable.

  `list(enumerate(['a','b','c'])) → [(0,'a'), (1,'b'), (2,'c')]`

- It is often used in **for** loops to track both position and value.
- It may be called with 1 or 2 arguments, as follows

```
enumerate(iterable)
enumerate(iterable, start)
```

  - The first argument must be an iterable (e.g., list, string).
  - The optional start argument sets the initial index (default is 0).

- Example:

```
colors = ['red', 'green', 'blue']
for index, color in enumerate(colors):
    print(f"{index}: {color}")
```

0: red
1: green
2: blue

`for` or `while`?

- Use **`while`** when the loop depends on a condition that may change unpredictably.
- Ideal for loops where you don’t know in advance how many times you'll iterate.
- Common in interactive programs, waiting for user input, or searching until a condition is met.

```
# Example: Keep asking until the user types 'exit'

user_input = ""
while user_input ≠ "exit":
    user_input = input("Type 'exit' to quit: ")
```

`for` or `while`?

- Use **`for`** when you're iterating over a known sequence or range.
- Best for fixed-length iterations or when looping through lists, strings, dictionaries, files, etc.
- Cleaner and more readable when the number of iterations is predictable.

```
# Print "Hello!" five times

for i in range(5):
    print("Hello!")
```

`for` or `while`?

Situation	Use `for`	Use `while`
Iterating over a list or string	✅	
Looping a fixed number of times	✅	
Waiting for a condition to change		✅
User-driven or unpredictable input		✅
Searching until a match is found		✅
Monitoring a system or sensor		✅

TYPICAL USAGES IN PYTHON

Fixed Repetition

- Problem: Print 5 temperature conversions from Celsius to Fahrenheit.
- You know the number of repetitions on advance.

- Possible solution:

```
for celsius in range(0, 50, 10):
    fahrenheit = (celsius * 9/5) + 32
    print(f'{celsius}°C = {fahrenheit:.1f}°F')
```

- Output:

0°C = 32.0°F
10°C = 50.0°F
20°C = 68.0°F
30°C = 86.0°F
40°C = 104.0°F

Conditional Repetition

- Problem: Keep asking for a password until the correct one is entered.
- You don’t know how many tries it will take.
- Possible solution:

```
password = ''
while password != 'secret123':
    password = input('Enter password: ')
print('Access granted!')
# Keeps asking until user types 'secret123'.
```

- Ideal for validation, retries or user interaction.

Accumulation / Aggregation

- Problem: Sum 5 daily sales amounts.
- You need to build up a results (a total).
- Possible solution:

```
total = 0
for i in range(5):
    sale = float(input(f'Enter number {i+1}: '))
    total += sale
print('Total sum =', total)

# Adds all numbers entered by the user.
```

- Useful for totals, averages, or statistics.

Validation Loop

- Problem: Ask the user for an integer between 1 and 10 until valid input is given.

- Possible solution:

```
value = 0
while value < 1 or value > 10:
    value = int(input('Enter a number (1–10): '))
print(f'Valid input: {value}')
# Repeat until user enters a number within range.
```

- Essential for robust, user-friendly programs.

Validation Loop (2)

- Problem: Ask for a number until the user enters a valid float.

- Possible solution:

```
while True:
    try:
        value = float(input("Enter a float: "))
        break
    except:
        print("Invalid input. Please try again.")
print(f"You entered: {value}")
```

- We will study the try … except statement later.

SOME COMMON MISTAKES BEGINNERS MAKE WITH LOOPS IN PYTHON

And How to Identify & Fix Them

Infinite Loops

- Mistake: Forgetting to update loop variable in while loops.

```
i = 0
while i < 5:
    print(i)
    # forgot i += 1 → infinite loop
```

- Symptom: Program never stops
- Fix: Ensure loop variable changes inside the loop.

```
i = 0
while i < 5:
    print(i)
    i += 1
```

Off-by-One Errors

- Mistake: Misunderstanding range()
  - end is excluded

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

- Symptom: Loop runs too few or too many times.
- Fix: Adjust range boundaries correctly

```
for i in range(1, 6):  # gives 1,2,3,4,5
    print(i)
```

While True Without Break

- Mistake: Using while True with no exit condition.

```
while True:
    print("hello")  # no break → infinite loop
```

- Symptom: Infinite loop.
- Fix: Add a clear break condition.

```
while True:
    response = input("Enter q to quit: ")
    if response == "q":
        break
```

Indentation Errors

- Mistake: Misaligned indentation in loop body.

```
for i in range(3):
print(i)    # IndentationError
```

- Symptom: IndentationError or unexpected logic.

- Fix: Use consistent 4 spaces (or tab) per indentation level.

Confusing break and continue

- Mistake: Using continue when break is needed

```
for i in range(5):
    if i == 3:
        continue   # skips 3, but loop continues
    print(i)
```

- Symptom: Loop skips elements but doesn't stop.
- Fix: Use break to exit loop completely, continue to skip iteration.

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

Exercises

- Do these codecheck exercises.