The while Loop, break & continue

AiCE Software Engineering Foundations — Python Programming

The for loop knows how many times it will run before it starts. This lesson introduces the while loop, for situations where you don't know that number in advance, along with break and continue — the two statements that let you control a loop's flow mid-execution.

1. The while Loop

A while loop repeats a block of code as long as a condition remains True. Unlike the for loop, which knows in advance how many times to run, the while loop keeps going until its condition becomes False — you use it when you don't know ahead of time how many iterations you'll need.

```
while condition:
    # code to repeat while condition is True
```

```
# Basic while loop
count = 1

while count <= 5:
    print(f"Iteration {count}")
    count += 1

print("While loop finished.")
```

The loop starts with count = 1. It checks: is count <= 5? Yes — so it runs the body. After printing, count += 1 increases count to 2. It checks again: is 2 <= 5? Yes — runs again. This continues until count becomes 6, at which point 6 <= 5 is False, and the loop ends.

> **The Infinite Loop**
> *The count += 1 line is critical. Without it, count would never change, and the condition would always be True — an infinite loop that runs forever until forcibly stopped with Ctrl + C. Every while loop must have a mechanism, usually a counter*

> *increment or variable update, that eventually makes its condition False. Always double-check your while loops before running them.*

2. while Loops With User Input

The while loop is the natural choice when a program should keep asking for input until the user provides something valid, or chooses to exit:

```
# Keep asking until the user types "quit"
print("AiCE Q&A System")
print("Type a question and press Enter. Type 'quit' to exit.")
print("-" * 45)

user_input = ""

while user_input.lower() != "quit":
    user_input = input("Your question: ")
    if user_input.lower() != "quit":
        print(f"You asked: '{user_input}'")
        print("Our team will respond within 24 hours.")
        print("-" * 45)

print("Thank you for using AiCE Q&A. Goodbye!")
```

The loop keeps accepting questions until the user types 'quit'. This pattern — loop until the user chooses to exit — is used in virtually every menu-driven application, chatbot, and command-line tool ever built.

Input Validation with while

Another powerful use of while is validating user input — keep asking until the user provides an acceptable answer:

```
# Validate that the score is between 0 and 100
score = -1

while score < 0 or score > 100:
    score = float(input("Enter exam score (0 to 100): "))
```

```
    if score < 0 or score > 100:
        print("Invalid score. Please enter a value between 0
and 100.")

print(f"Valid score recorded: {score}")
```

Entering -5, then 150, then 78 shows the loop rejecting every invalid value and only accepting a score genuinely within range.

3. for vs. while — Choosing the Right Loop

Situation	Use	Reason
You know exactly how many times to repeat	for	range() gives precise control
You are iterating over a list or string	for	for naturally handles sequences
You don't know how many iterations you need	while	Repeats until a condition changes
Waiting for valid user input	while	Condition depends on what the user enters
Running a menu until the user exits	while	Exit condition is user-controlled
Counting or accumulating over a fixed range	for	Cleaner and safer than while for counting

4. The break Statement — Exiting a Loop Early

break immediately terminates a loop, regardless of whether its condition is still True or whether more items remain to iterate. Execution jumps to the first line after the loop:

```
# Searching for a student in a list
students = ["Amara", "Kofi", "Ngozi", "Fatima", "Kwame",
"Aisha"]
target = input("Enter student name to search:
").strip().title()
```

```
found = False

for student in students:
    if student == target:
        print(f"Found: {student} is in the AiCE cohort.")
        found = True
        break    # No need to keep searching

if not found:
    print(f"{target} is not in the current cohort.")
```

The loop stops as soon as it finds the target — there's no reason to keep checking the remaining names once a match is confirmed.

break is also common in an intentional infinite loop, exited only when a specific condition is met — the backbone of menu-driven programs:

```
print("=== AiCE MENU ===")

while True:
    print("1. View Profile")
    print("2. View Courses")
    print("3. Exit")
    choice = input("Choose an option: ")

    if choice == "1":
        print("Loading your profile...")
    elif choice == "2":
        print("Loading courses...")
    elif choice == "3":
        print("Goodbye!")
        break
    else:
        print("Invalid choice. Try again.")
```

while True: with break

while True: creates a loop with no natural end — it would run forever without a break inside it. Combined with break, this is an extremely common and intentional pattern for menu-driven programs, where the loop should keep running until the user explicitly chooses to exit.

5. The continue Statement — Skipping an Iteration

continue skips the rest of the current iteration and jumps immediately to the next one. The loop does not exit — it simply skips whatever comes after continue for that particular pass:

```
# Skip students who have not paid fees
students = [
    {"name": "Amara", "paid": True},
    {"name": "Kofi",   "paid": False},
    {"name": "Ngozi", "paid": True},
    {"name": "Fatima", "paid": False},
    {"name": "Kwame", "paid": True},
]

print("Sending course materials to paid students:")

for student in students:
    if not student["paid"]:
        print(f"Skipping {student['name']} — fee not paid.")
        continue
    print(f"Sending to {student['name']} — materials
dispatched.")
```

When continue is hit for Kofi, Python skips the rest of that iteration — the 'materials dispatched' line — and moves straight to Ngozi. The loop continues; it does not exit.

```
# Print only odd numbers from 1 to 10
for num in range(1, 11):
    if num % 2 == 0:
        continue
```

```
    print(num)
```

Whenever num is even, continue skips the print entirely and moves to the next number — only odd numbers ever reach the print statement.

Key Takeaways

while repeats for as long as its condition holds True, and must always contain a mechanism that eventually makes it False — otherwise you get an infinite loop. break exits a loop immediately and completely; continue skips only the current iteration and keeps the loop running. while True: combined with break is the standard pattern behind virtually every menu-driven program. The next lesson covers nesting loops inside one another, and combining loops with the conditional logic from an earlier lesson to build genuinely intelligent, data-processing programs.