PYTHON LESSON

Infinite Loops

Your calculator program's own debugging session is a near-perfect case study in infinite loops — you hit the two most common mistakes in a single sitting: a loop that should repeat but doesn't (a missing `break`), and a loop that repeats forever when it shouldn't (a condition that never changes). This lesson walks through both, plus a related validation gap still sitting in the code.

⚡ Coming from Java / JavaScript

`while True:` is exactly `while (true) {}` — same idea, no braces, indentation defines the block. `break` and `continue` are the identical keywords doing the identical job in all three languages, so that part transfers directly.

The real gap: **Python has no `do-while` loop.** Java/JS let you write `do { ... } while (condition);` to guarantee the body runs at least once before checking. Python doesn't have that construct at all — the idiomatic replacement is exactly the pattern your own script ended up using: `while True:` with an `if ...: break` at the point where the real condition would go.

🚀 FastAPI relevance

The `while True:` + `break` pattern is exactly how retry loops are written against flaky external services — call an API, and if it fails, wait and try again, breaking out only on success or after too many attempts. You'll see this shape constantly once you're writing backend code that talks to other services.

The Building Blocks

while True: — Deliberate Infinite Loop

Runs forever on purpose. Something inside *must* break out, or it never ends.

```
while True:
    print("Runs forever...")
    # ...until something breaks out
```

break — Escape Immediately

Exits the nearest enclosing loop the instant it runs — nothing else in the loop body executes.

```
while True:
    ans = input("quit? ")
    if ans == "quit":
        break
```

continue — Skip to Next Iteration

Skips whatever's left in the current pass and jumps straight back to the loop's top.

```
if choice not in "123456":
    print("Invalid")
    continue
```

if vs. while — Once vs. Every Time

An `if` checks a condition exactly once. It never loops back to check again, no matter what.

```
op1 = input("Enter: ")
if not op1.isnumeric():
    print("Error!")
# ...code continues here regardless
```

The Stale-Condition Trap

A `while` re-checks its condition every pass — but only if something inside actually changes it.

```
op1 = input("Enter: ")
while not op1.isnumeric():
    print("Error!")
    # op1 never changes here!
```

No do-while — Python's Idiom

This is the standard replacement: loop forever, check the real condition, break when satisfied.

```
while True:
    op1 = input("Enter: ")
    if op1.isnumeric():
        break
```

In Context: The Corrected Validation Loop

Re-Prompting Inside the Loop — The Missing Piece

Your first fix (switching `if` to `while`) was necessary but not sufficient — it stopped the crash, but created the "bonus" infinite loop, because `op1` was never reassigned inside the loop body. The condition `not op1.isnumeric()` was checking the exact same value, forever. The real fix needs *both* pieces: a `while` to keep checking, and a fresh `input()` call inside the loop to actually give the condition a chance to change.

```
op1 = input("Please enter the first operand: ")
while not op1.isnumeric():
    print("Error :  It must be a  number!")
    op1 = input("Please enter the first operand: ")  # <- re-asks, updates op1

op2 = input("Please enter the second operand: ")
while not op2.isnumeric():
```

```
    print("Error :  It must be a  number!")
    op2 = input("Please enter the second operand: ")
```

In Context: The Nested Loop and the Missing break

Your calculator has two loops: an outer `while True:` showing the menu, and an inner `while True:` collecting the two operands and performing the calculation. The inner loop's whole job is to run *once* per calculation, then hand control back to the outer menu loop — which only happens if it actually `break`s.

```
while True:  # outer: the menu
    operation = input("Enter 1-6: ")
    # ...

    while True:  # inner: collect operands, calculate
        op1 = input("Please enter the first operand: ")
        # ...validation loops here...
        # ...perform the calculation, print the result...
        break  # <- without this, the inner loop just asks for op1 and op2 again
```

Without that `break`, the inner loop never returns control to the outer one — the program just keeps re-asking for operands forever, even though there's nothing actually wrong with the input this time. It's a genuine infinite loop, just a quieter one than the stale-condition version, since it looks like the program is working right up until you realize the menu never comes back.

Quick Reference

CONSTRUCT	WHAT IT DOES	EXAMPLE
`while True:`	Loops forever until something inside breaks out	`while True: ...`
`break`	Exits the nearest enclosing loop immediately	`if done: break`

CONSTRUCT	WHAT IT DOES	EXAMPLE
`continue`	Skips the rest of this pass, jumps to the next	`if bad: continue`
`while condition:`	Re-checks the condition every pass — loops while True	`while not valid: ...`
`if condition:`	Checks once — never loops, no matter what	`if not valid: print(...)`
`try / except ValueError`	Robust numeric validation — handles negatives and decimals, unlike `isnumeric()`	`try: float(x)` `except ValueError: ...`

🧩 Practice Problems

Problem 1 — The Nested Loop That Won't Let Go

The calculator correctly performs the operation and prints a result — but the menu never reappears. The program just keeps asking for a first operand, over and over, forever.

```
while True:
    op1 = input("Please enter the first operand: ")
    # ...(validation loops omitted for brevity)...
    op2 = input("Please enter the second operand: ")
    # ...(validation loops omitted for brevity)...

    op1 = float(op1)
    op2 = float(op2)

    if operation == '1':
        print("The results is ", op1 + op2)
    # ...other elif branches...
```

SOLUTION

The inner loop is missing its exit. Add a `break` as the last line of the inner loop's body, once the calculation and print are done — that's what actually hands control back to the outer menu loop.

```
    if operation == '1':
        print("The results is ", op1 + op2)
    # ...other elif branches...
    break  # <- exits the inner loop, returns to the menu
```

Problem 2 — The Bonus Infinite Loop

You've switched from `if` to `while` to force re-validation. It compiles and runs — but typing an invalid value now freezes the program in an endless stream of error messages, with no way to correct it.

```
op1 = input("Please enter the first operand: ")
while not op1.isnumeric():
    print("Error :  It must be a  number!")
```

The condition `not op1.isnumeric()` is checked again on every pass — but `op1` itself is never touched inside the loop body, so it can never stop being invalid.

SOLUTION

Re-ask for the input *inside* the loop body, reassigning `op1` each time, so the condition actually has a chance to change.

```
op1 = input("Please enter the first operand: ")
while not op1.isnumeric():
    print("Error :  It must be a  number!")
    op1 = input("Please enter the first operand: ")  # <- the fix
```

Problem 3 — Negative Numbers Rejected

The validation loop now correctly re-prompts on genuinely invalid input like `"abc"` — but it also refuses `"-5"`, even though that's a perfectly valid number for this calculator to work with. A user who only wants to subtract using a negative operand is stuck in what functions as an infinite loop from their side: no input they can honestly give will ever satisfy the check.

```
op1 = input("Please enter the first operand: ")
while not op1.isnumeric():  # "-5".isnumeric() is False — the minus sign isn't num
    print("Error :  It must be a  number!")
    op1 = input("Please enter the first operand: ")
```

`str.isnumeric()` only recognizes digit characters — a leading `-` (or a decimal point) makes it return `False`, even though `float()` would happily convert that same string.

SOLUTION

Replace the `isnumeric()` check with an attempt to actually convert the value, catching the failure if it isn't a real number. This is the more robust, idiomatic way to validate numeric input in Python — it also naturally accepts decimals like `"3.5"`, another case `isnumeric()` would have rejected.

```
def is_valid_number(text):
    try:
        float(text)
        return True
    except ValueError:
        return False

op1 = input("Please enter the first operand: ")
while not is_valid_number(op1):
    print("Error :  It must be a  number!")
    op1 = input("Please enter the first operand: ")

op1 = float(op1)  # now safe — "-5" converts to -5.0 correctly
```

⚠ Gotchas for Java / JavaScript Programmers

No do-while: reach for `while True:` + `if ...: break` instead of hunting for a construct that doesn't exist.

if is not a loop: if you want "keep asking until it's valid," it has to be `while` — an `if` only ever checks once, then execution moves on regardless.

A while loop is only as good as what changes inside it: if the condition depends on a variable, something inside the loop body must actually update that variable, or the

loop runs forever — a trap in any language, but easy to hit the first time you translate "keep checking" logic into Python.

Validation methods can be stricter than they look: `isnumeric()` silently rejects negative numbers and decimals. And a related edge case worth remembering separately: dividing by zero in Python raises `ZeroDivisionError` rather than returning `Infinity` the way JavaScript does — worth a guard clause before a division operation with user-supplied input.