PART 3 OF 8

Python Self-Learning Booklet 3: Conditional Logic, User Input, and Loops

In this session you will learn how to make your programs **smarter** – they'll be able to make decisions, respond to what the user types, and repeat actions automatically. These three ideas – conditionals, input, and loops – are the foundation of almost every real program you'll ever write.

Conditional Logic

Make decisions with `if`, `elif`, and `else`

User Input

Read what the user types with `input()`

While Loops

Repeat code using flags, `break`, and `continue`

LEARNING GOALS

What You'll Be Able to Do

By the end of this session, you should be comfortable with all five of these skills:

1

Write Conditional Tests

Expressions that evaluate to `True` or `False`

2

Control Program Flow

Use `if`, `if-else`, and `if-elif-else`

3

Combine Conditions

Use `and`, `or`, `in`, and `not in`

4

Handle User Input

Use `input()` and convert text to numbers with `int()`

5

Use While Loops

Repeat actions using flags, `break`, and `continue`

CONCEPT 1

Conditional Tests: True or False

A **conditional test** is an expression that evaluates to either `True` or `False`. Python uses these tests to decide whether to run certain code. Every comparison you write will produce one of these two outcomes.

Common Operators

- `==` – equal to
- `!=` – not equal to
- `<` – less than
- `>` – greater than
- `<=` – less than or equal
- `>=` – greater than or equal

Try It

```
print(1 == 2)   # False
print(2 == 2)   # True

car = 'bmw'
print(car == 'bmw')   # True
print(car == 'audi')  # False

age = 19
print(age < 21)   # True
print(age >= 21)  # False
```

CONCEPT 2

Assignment = vs. Equality ==

This is one of the most common beginner mistakes. These two operators look similar but do completely different things. Mixing them up causes errors that can be hard to spot.

Single = (Assignment)

Stores a value into a variable. This does **not** compare anything.

```
car = 'bmw'
# Puts 'bmw' into car
```

Double == (Equality Test)

Checks whether two values are equal. Returns `True` or `False`.

```
print(car == 'bmw')
# True
```

If you write `if car = 'bmw':` Python will throw a **SyntaxError**. Always use `==` inside conditions.

CONCEPT 3

Inequality: The != Operator

Use `!=` to test whether two values are **different**. It is the opposite of `==`. It's especially useful when you want to do something only when a value is *not* what you expect.

```
car = 'bmw'

print(car == 'bmw')   # True
print(car != 'bmw')   # False
print(car != 'audi')  # True
```

> ### Read it in plain English
>
> `car != 'audi'` means: "Is the value of `car` something other than `'audi'`?" Since it's `'bmw'`, the answer is `True`.

CONCEPT 4

Case Sensitivity and `.lower()`

Python treats uppercase and lowercase letters as completely different characters. `'Audi'` and `'audi'` are **not equal** to Python. The solution: convert both sides to lowercase before comparing.

The Problem

```
car = 'Audi'
print(car == 'audi')
# False — cases differ!
```

The Fix

```
car = 'Audi'
print(car.lower() == 'audi')
# True — now it matches!
```

`.lower()` does **not** change the original variable – it just returns a lowercase copy for the comparison. This is handy when processing user input, where you can't control how the user types.

CONCEPT 5

Logical Operators: `and` / `or`

Logical operators let you **combine** multiple tests into a single condition. Use them when a decision depends on more than one thing at once.

`and` — Both must be True

```
age_0 = 22
age_1 = 18

print(age_0 >= 21 and age_1 >= 21)
# False (age_1 is only 18)
```

`or` — At least one must be True

```
age_0 = 22
age_1 = 18

print(age_0 >= 21 or age_1 >= 21)
# True (age_0 qualifies)
```

Think of `and` as "strict" (all conditions must pass) and `or` as "flexible" (any condition can pass).

CONCEPT 6

Membership Tests: `in` and `not in`

Membership tests let you check whether a value **exists inside a list**. They make code much more readable than checking every item manually.

Using `in`

```
toppings = ['mushrooms', 'onions']
print('mushrooms' in toppings)
# True
```

Using `not in`

```
banned = ['andrew', 'carolina']
user = 'marie'
print(user not in banned)
# True
```

Tip: Membership tests also work on strings! `'py' in 'python'` returns `True`.

CONCEPT 7

Boolean Variables

A **Boolean** value is simply `True` or `False`. You can store these values in variables to track the **state** of your program – for example, whether a game is active or whether a user can edit a file.

```
game_active = True
can_edit = False

bigger = 1 > 0
print(bigger)  # True
```

Direct Assignment

Store `True` or `False` directly: `game_active = True`

From a Comparison

Store the result of a test: `bigger = 1 > 0` stores `True`

As a Loop Flag

Control loops: `while active:` keeps running as long as `active` is `True`

</> CONCEPT 8

The `if` Statement

The `if` statement runs a block of code **only when** the condition is `True`. The indented code beneath it is the "block" that belongs to the `if`.

```
age = 19

if age >= 18:
    print("You are old enough to vote!")
```

→ Python evaluates the condition

`age >= 18` → is 19 greater than or equal to 18? `True`

→ The indented block runs

Because the condition is `True`, the `print()` executes

→ If False, the block is skipped

Nothing is printed – Python moves on to the next line after the block

</> CONCEPT 8 CONT.

The if-elif-else Chain

Use if-elif-else when exactly **one path** should run. Python checks each condition from top to bottom and stops at the first match. Order matters – put the most specific conditions first.

Admission Pricing

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
else:
    price = 40

print(price)  # 25
```

You Can Skip else

If your elif branches cover every possible case, you don't need an else. This is fine when there's no "catch-all" default action needed.

```
elif age >= 65:
    price = 20

print(price)
```

Only the **first matching** condition runs. Once Python finds a match, it skips the rest of the chain.

CONCEPT 9

Independent `if` Statements

When you use separate `if` statements (instead of `elif`), **all** of them are checked independently. Multiple blocks can run if multiple conditions are true. Use this when checks are unrelated to each other.

if-elif (only one runs)

```
if 'mushrooms' in toppings:
    print("Adding mushrooms.")
elif 'extra cheese' in toppings:
    print("Adding cheese.")
```

Once the first match is found, the chain stops.

Separate ifs (both can run)

```
if 'mushrooms' in toppings:
    print("Adding mushrooms.")

if 'extra cheese' in toppings:
    print("Adding extra cheese.")
```

Both conditions are checked – both messages print.

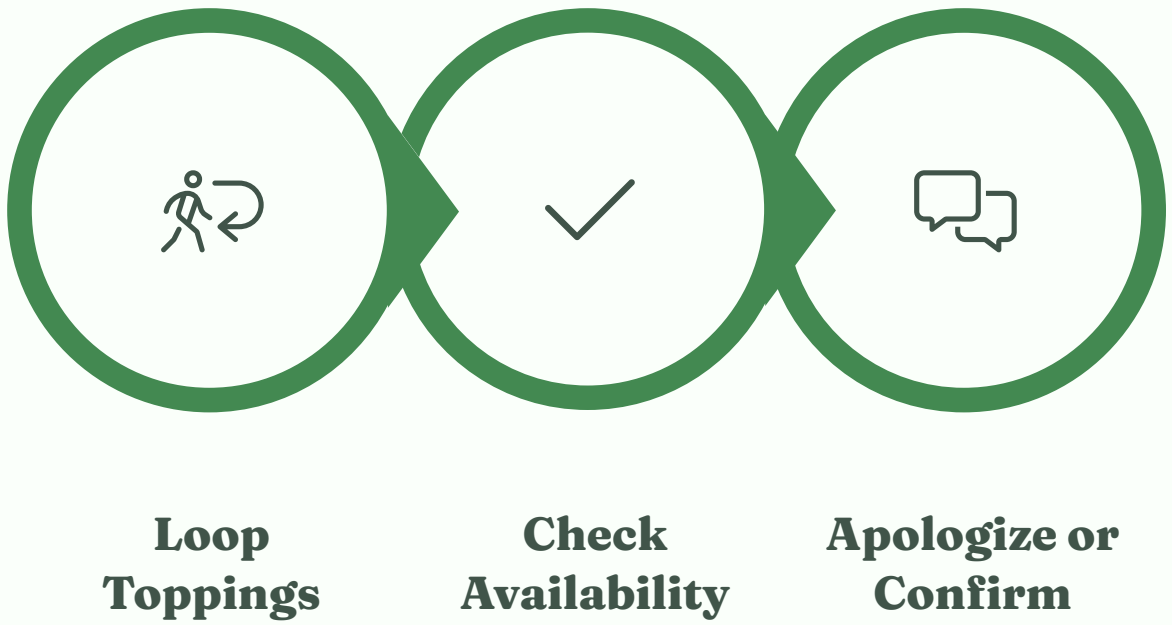
CONCEPT 10

`if` Statements with Lists

A powerful pattern: loop through a list and use `if` inside the loop to handle each item differently based on availability.

```
available_toppings = ['mushrooms', 'extra cheese']
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for topping in requested_toppings:
    if topping not in available_toppings:
        print("Sorry,", topping, "is not available today.")
    else:
        print("Adding", topping)
```



This pattern – loop, then check, then act – is one you'll use constantly in real programs.

>_ CONCEPT 11

Getting User Input with `input()`

`input()` pauses the program, displays a prompt to the user, waits for them to type something, then returns what they typed as a **string**.

Basic Usage

```
message = input("Tell me something: ")
print(message)
```

With an F-String

```
name = input("Please enter your name: ")
print(f"Hello, {name}!")
```

> **Write clear, descriptive prompts**
>
> Always tell the user exactly what to type. A good prompt reads like a complete instruction: `"Please enter your age: "` rather than just `"Age: "`. Notice the space before the closing quote – it keeps the cursor away from the prompt text.

CONCEPT 12

Input is Always a String — Convert with `int()`

Even when the user types `19`, Python receives `'19'` – a string, not a number. You **must convert** it before using it in numeric comparisons.

The Problem

```
age = input("How old are you? ")
print(type(age))
# <class 'str'>

# This would fail or give wrong result:
# if age >= 18:
```

The Fix — Use `int()`

```
age = input("How old are you? ")
age = int(age)

if age >= 18:
    print("You can vote!")
else:
    print("Too young to vote.")
```

Remember: `int()` will crash if the user types something that isn't a number (like `'hello'`). You'll learn how to handle that safely in a later session.

% CONCEPT 13

The Modulo Operator %

The % operator gives you the **remainder** after division. Its most common use is checking whether a number is even or odd – or more generally, detecting multiples.

How It Works

```
print(4 % 3)  # 1
print(6 % 3)  # 0
```

If the remainder is 0, the number divides evenly.

Even / Odd Checker

```
number = int(input("Enter a number: "))

if number % 2 == 0:
    print("Even number")
else:
    print("Odd number")
```

0

Remainder = Even

Any number divisible by 2 with no remainder is even

1

Remainder = Odd

Any number that leaves remainder 1 when divided by 2 is odd

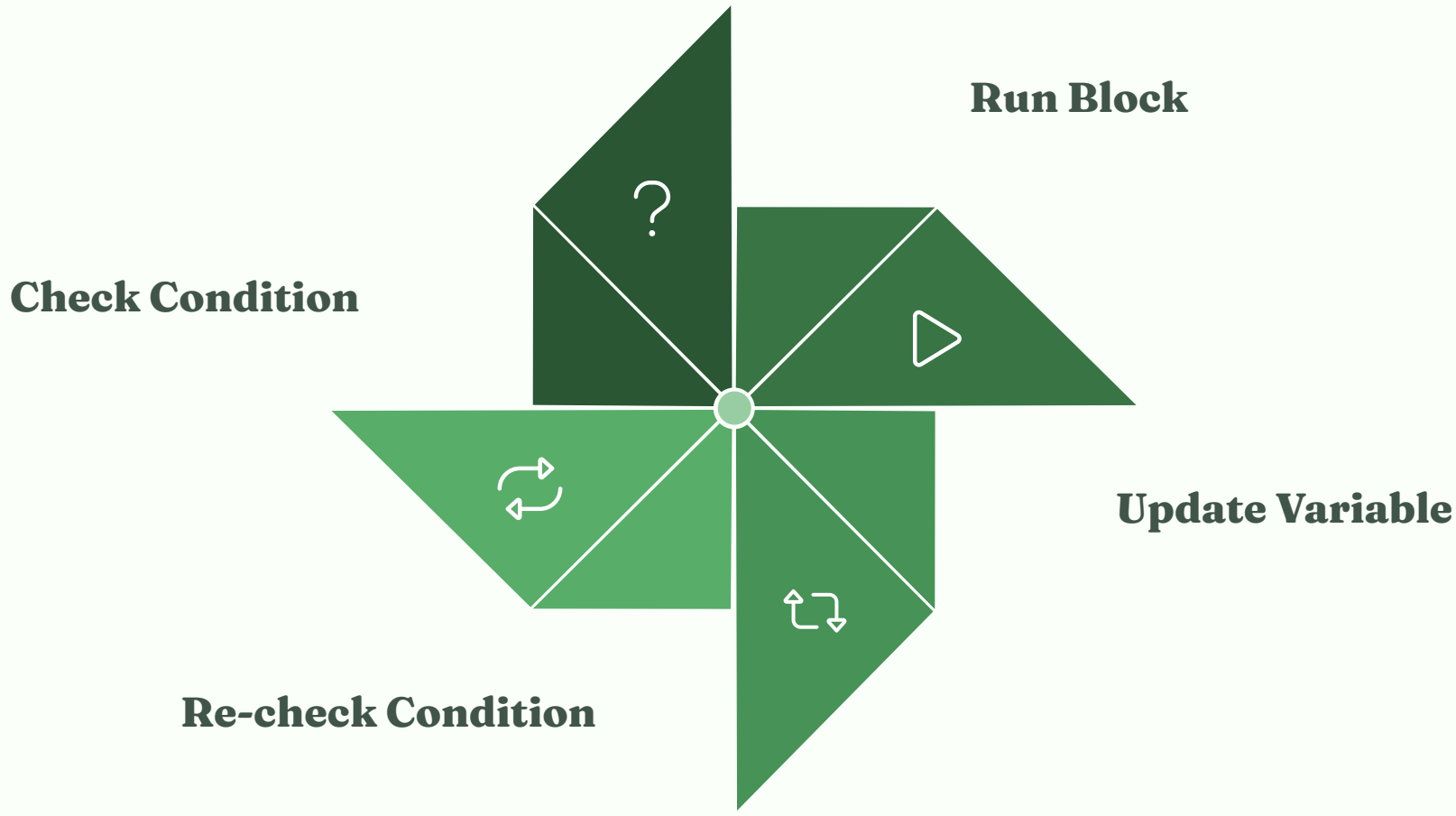
CONCEPT 14

The `while` Loop

A `while` loop keeps running its block of code **as long as the condition stays True**. Each pass through the loop is called an **iteration**.

```
current_number = 1

while current_number <= 5:
    print(current_number)
    current_number += 1
```



Critical: Always update your loop variable inside the loop (e.g., `current_number += 1`). If the condition never becomes `False`, the loop runs forever – an **infinite loop**!

CONCEPT 15

Ending a while Loop: Flags, break, continue

There are three clean ways to control when a loop stops or skips. Each has its best use case.

Flag Variable

Best when you have **multiple exit conditions**. A boolean variable controls the loop.

```
active = True
while active:
    msg = input("Enter text or 'quit': ")
    if msg == 'quit':
        active = False
    else:
        print(msg)
```

break

Exits the loop **immediately**, right where it's called.

```
while True:
    city = input("Enter city or 'quit': ")
    if city == 'quit':
        break
    print(f"I'd love to go to {city}!")
```

continue

Skips the rest of the current iteration and **jumps back** to the condition check.

```
number = 0
while number < 6:
    number += 1
    if number == 3:
        continue
    print(number)  # prints 1,2,4,5,6
```

CONCEPT 16

Processing Lists with `while`

`while` loops are great for modifying lists as you go – something you can't safely do with a `for` loop. Two key patterns:

Move Items Between Lists

```
unconfirmed = ['alice', 'brian', 'candace']
confirmed = []

while unconfirmed:
    user = unconfirmed.pop()
    print(f"Verifying {user}")
    confirmed.append(user)
```

Runs until `unconfirmed` is empty (an empty list is `False`).

Remove All Occurrences

```
pets = ['dog', 'cat', 'dog', 'goldfish', 'cat']

while 'cat' in pets:
    pets.remove('cat')
    print(pets)
```

`list.remove()` only removes **one** occurrence at a time – the loop handles the rest.

CHAPTER BREAK

Putting It All Together

You now have all the building blocks. The following example combines **user input**, **conditionals**, **while loops**, **break**, and **list membership** into a single runnable program – a simple drink ordering helper.

What it does

Keeps asking the user to pick a drink from an available list

When to stop

Exits cleanly when the user types `quit`

At the end

Prints the complete order, or a message if nothing was added

STEP-BY-STEP EXAMPLE

The Full Code

```
available_items = ['water', 'tea', 'coffee', 'juice']
order = []

while True:
    item = input("Choose a drink (water/tea/coffee/juice) or type 'quit': ")

    if item == 'quit':
        break

    item = item.lower()

    if item not in available_items:
        print(f"Sorry, {item} is not available.")
    else:
        order.append(item)
        print(f"Added {item}.")

print("\nYour order:")
if order:
    for item in order:
        print("-", item)
else:
    print("No items were added.")
```

STEP-BY-STEP EXAMPLE

How the Code Works — Step by Step

01

Set up data

Create `available_items` list and an empty `order` list

02

Start infinite loop

`while True:` runs forever until explicitly stopped

03

Ask for input

Use `input()` to prompt the user for a drink choice

04

Check for quit

If user typed `'quit'`, call `break` to exit immediately

05

Normalize case

Apply `.lower()` so `'Tea'` and `'tea'` both work

06

Check membership & update order

If item is available, append it; otherwise print an apology

07

Print final order

Use `if order:` to handle the empty-order edge case

COMMON MISTAKES

Watch Out for These Pitfalls

These are the most frequent errors beginners make in this session. Reading them now will help you spot them faster when debugging your own code.

❌ Using `=` instead of `==`

`if x = 5:` is a **SyntaxError**. Python can't assign inside a condition. Always use `==` to compare.

❌ Forgetting `input()` returns a string

Comparing a string like `'19'` to a number like `18` gives unexpected results. Always convert with `int()` first.

❌ Infinite loops

Forgetting to update the loop variable (e.g., `current_number += 1`) means the condition never becomes `False` and the loop runs forever.

❌ Wrong indentation

In Python, indentation *is* the structure. Code indented under an `if` or `while` belongs to that block. Misaligned code produces bugs or errors.

❌ Case-sensitivity in comparisons

Forgetting that `'BMW'` ≠ `'bmw'` leads to silent failures. Use `.lower()` when comparing user-supplied text.

PRACTICE TASKS

Practice Task 1: Score Checker

Apply your knowledge of `if-else` to a simple pass/fail grading system.

Your Goal

- Create a variable `score`
- If `score >= 60`, print `"Pass"`
- Otherwise print `"Fail"`
- Test with at least two different scores

Starter Structure

```
score = ???

if score >= 60:
    print("???")
else:
    print("???")
```

Try `score = 75` and `score = 45`. Do you get the right output each time?

PRACTICE TASKS

Practice Task 2: Stage of Life

Combine `input()`, `int()`, and an `if-elif-else` chain to categorize the user's age.

Age Range	Category	Condition to Write
Under 2	Baby	`if age < 2:`
2 - 4	Toddler	`elif age < 5:`
4 - 13	Kid	`elif age < 13:`
13 - 20	Teenager	`elif age < 20:`
20 - 65	Adult	`elif age < 65:`
65 and over	Elder	`else:`

Remember to convert the input: `age = int(input("Your age: "))`

PRACTICE TASKS

Practice Tasks 3, 4 & 5

Task 3: Username Check

- Create `current_users = ['Alex', 'Mina', 'admin']`
- Create a list of `new_users` with some overlapping names in different cases
- For each new user, check if their name (lowercased) already exists
- **Hint:** use `.lower()` on both sides of the comparison

Task 4: Multiple of 10

- Ask the user for a number with `input()`
- Convert it with `int()`
- Use `%` to check if it divides evenly by `10`
- Print `"Multiple of 10"` or `"Not a multiple of 10"`

Task 5: Toppings Loop

- Use `while True:` to keep asking for a topping
- If the user types `'quit'`, use `break`
- Otherwise print a confirmation and add it to a list
- After the loop, print the full list of toppings ordered

EXPLORATION QUESTIONS

Think Deeper: Exploration Questions

These questions don't have a single "correct" answer – they're designed to deepen your understanding by encouraging you to **experiment and observe**.

1 What happens without `.lower()`?

In the step-by-step example, remove the line `item = item.lower()`. Then try entering `Tea` or `COFFEE`. What does the program say? Why? How does this demonstrate Python's case sensitivity?

2 Flag vs. `break` — when to use which?

Write a simple loop that exits when the user types `'quit'` – once using a flag variable, and once using `break`. When would a flag be better? When is `break` cleaner?

3 Why not use `for` to remove items?

Try replacing `while 'cat' in pets:` with a `for` loop that calls `pets.remove('cat')`. Run it and observe. What unexpected behavior do you notice? Why does modifying a list while iterating over it cause problems?

SESSION SUMMARY

Key Concepts at a Glance

Conditional Tests

Expressions using `==`, `!=`, `<`, `>=` etc. evaluate to `True` or `False`. Use `.lower()` for case-insensitive comparisons.

If / Elif / Else

Use `if-elif-else` chains for mutually exclusive paths. Use separate `if` statements when multiple conditions should be checked independently.

Logical & Membership

`and` / `or` combine multiple tests. `in` / `not in` check list membership cleanly and readably.

User Input

`input()` always returns a string. Use `int()` to convert to a number before doing math or numeric comparisons.

While Loops

Repeat code while a condition is `True`. Use flags for multiple exit rules, `break` to exit immediately, `continue` to skip an iteration.

QUICK REFERENCE

Operators & Syntax Cheat Sheet

Operator / Syntax	What It Does	Example
`==`	Tests equality	`car == 'bmw'` → `True`
`!=`	Tests inequality	`car != 'audi'` → `True`
`and` / `or`	Combines conditions	`a > 0 and b > 0`
`in` / `not in`	List membership	`'cat' in pets`
`input()`	Reads user text (returns string)	`name = input("Name: ")`
`int()`	Converts string to integer	`age = int(age)`
`%`	Remainder after division	`6 % 3` → `0`
`while condition:`	Repeat while True	`while active:`
`break`	Exit loop immediately	Inside `if city == 'quit':`
`continue`	Skip rest of iteration	Jump back to `while` check

UP NEXT

Well Done — You Completed Session 3!

You've learned the core tools for writing programs that **make decisions**, **interact with users**, and **repeat actions** automatically. These skills are used in virtually every Python program you'll ever write.

✅ What You Mastered

- Conditional tests with comparison operators
- if, elif, else chains
- and, or, in, not in
- User input with input() and int()
- while loops with flags, break, continue

➡️ Coming in Session 4

- Functions – writing reusable blocks of code
- Parameters and return values
- Organizing larger programs
- Scope and variable visibility

Before moving on, make sure you've completed all five practice tasks and tested your code with multiple different inputs. If anything feels unclear, re-read the relevant concept and try the examples in your Python environment.