Python while Loops

Introduction

A program often needs to repeat an action.

For example:

- Keep asking for a password until it is correct.
- Keep processing items while there are items left.
- Keep a game running while the player has lives remaining.
- Keep retrying an operation while it has not succeeded.
- Keep reading input while the user has not chosen to exit.

Python provides loops for these situations.

A **while loop** repeats a block of code **as long as a condition remains true**.

The core idea is simple:

> **While this condition is true, keep doing this work.**

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

But understanding while loops properly requires more than memorizing this syntax. You need to understand:

1. How the condition controls repetition
2. How Python executes the loop
3. How the loop eventually stops
4. How to update values inside the loop
5. How to avoid infinite loops
6. When `while` is better than `for`
7. How while loops appear in real applications

---

1. Why Do We Need while Loops?

Imagine you want to print numbers from 1 to 5.

Without a loop:

```
print(1)
print(2)
print(3)
```

```
print(4)
print(5)
```

This works, but it is repetitive.

A loop lets you express the actual rule:

> Start at 1 and keep printing while the number is 5 or less.

```
number = 1

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

Output:

```
1
2
3
4
5
```

The loop is useful because you describe **the condition for continuing**, rather than manually writing every repetition.

---

2. The Mental Model of a while Loop

Think of a while loop as a checkpoint.

Python repeatedly asks:

```
Is the condition True?
       |
       v
     YES
       |
       v
Run the loop body
       |
       v
Update something
       |
       v
Check condition again
       |
```

```
        +------> YES → repeat
        |
       NO
        |
        v
Exit loop
```

For example:

```
count = 1

while count <= 3:
    print(count)
    count += 1
```

Python thinks:

```
count = 1
1 <= 3 → True → run body

count = 2
2 <= 3 → True → run body

count = 3
3 <= 3 → True → run body

count = 4
4 <= 3 → False → stop
```

This execution model is extremely important.

---

3. Basic Syntax

The general syntax is:

```
while condition:
    statement
```

Example:

```
age = 18

while age <= 21:
    print(age)
    age += 1
```

There are three important parts:

1. `while`

Tells Python that a loop is beginning.

2. Condition

Determines whether the loop should continue.

```
age <= 21
```

3. Loop body

The indented code runs whenever the condition is `True`.

```
print(age)
age += 1
```

---

4. The Condition Must Produce True or False

A while loop depends on a Boolean condition.

For example:

```
count < 10
```

Python evaluates it as either:

```
True
```

or:

```
False
```

Example:

```
count = 5

while count < 10:
    print(count)
    count += 1
```

The first check is:

```
5 < 10
```

which is:

```
True
```

So the loop runs.

---

5. The Simplest while Loop

```
x = 1

while x <= 3:
    print("Hello")
    x += 1
```

Output:

```
Hello
Hello
Hello
```

The loop runs three times.

Why?

Because:

```
x = 1 → condition True
x = 2 → condition True
x = 3 → condition True
x = 4 → condition False
```

---

6. Updating the Loop Variable

One of the most important concepts in while loops is **progress**.

Consider:

```
number = 1

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

This line:

```
number += 1
```

changes the condition's input.

Without it:

```
number = 1

while number <= 5:
    print(number)
```

`number` remains `1`.

The condition remains:

```
1 <= 5
```

forever.

The loop never reaches a state where the condition becomes false.

This creates an **infinite loop**.

---

7. The Three Parts of a Safe while Loop

A well-designed while loop usually has three components:

```
Initialization
      ↓
Condition
      ↓
Update
```

Example:

```
count = 1

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

Initialization

```
count = 1
```

Defines the starting state.

Condition

```
count <= 5
```

Defines when the loop continues.

Update

```
count += 1
```

Moves the loop toward termination.

A useful mental checklist is:

> **Where does the loop start?**
>
> **Why does it continue?**
>
> **What makes it stop?**

---

8. Counting Up

A common use of while loops is counting upward.

```
number = 1

while number <= 10:
    print(number)
    number += 1
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

---

9. Counting Down

A while loop can also count backward.

```
number = 5

while number >= 1:
    print(number)
    number -= 1
```

Output:

```
5
4
3
2
1
```

Notice the update:

```
number -= 1
```

The value moves toward the condition becoming false.

---

10. Skipping Values

You can change the variable by more than one.

```
number = 0

while number <= 10:
    print(number)
    number += 2
```

Output:

```
0
2
4
6
8
10
```

The rule is:

> Start at 0 and repeatedly add 2 while the number is at most 10.

---

11. Using Comparison Operators

while loops can use different comparison operators.

Less than

```
while number < 10:
```

Less than or equal

```
while number <= 10:
```

Greater than

```
while number > 0:
```

Greater than or equal

```
while number >= 0:
```

Not equal

```
while number != 5:
```

Understanding the condition is more important than memorizing the syntax.

---

12. Using Logical Operators

You can combine conditions.

`and`

Both conditions must be true.

```
age = 20
balance = 500

while age >= 18 and balance > 0:
    print("Transaction available")
    balance -= 100
```

The loop continues only while both conditions are true.

`or`

At least one condition must be true.

```
number = 1

while number < 5 or number == 10:
    print(number)
    number += 1
```

Be careful with complex conditions. Always verify what values can make the condition false.

`not`

You can reverse a Boolean condition.

```
is_logged_in = False

while not is_logged_in:
    print("Waiting for login...")
    is_logged_in = True
```

---

13. Boolean Variables in while Loops

A while loop can directly use a Boolean variable.

```
running = True

while running:
    print("Application is running")
    running = False
```

This pattern becomes very useful in applications.

For example:

```
game_running = True

while game_running:
    print("Game loop is running")
    game_running = False
```

The loop continues while the application is in a particular state.

---

14. User Input and while Loops

One of the most practical uses of while loops is repeatedly accepting user input.

```
password = ""

while password != "python123":
    password = input("Enter password: ")

print("Access granted")
```

The program keeps asking until the correct password is entered.

This is different from a fixed-count loop.

You don't know how many attempts the user will need.

They might enter:

```
hello
```

then:

```
python
```

then:

```
12345
```

and finally:

```
python123
```

The loop handles all of those possibilities.

---

15. Sentinel Values

A **sentinel value** is a special value that tells the program to stop.

Example:

```
command = ""

while command != "quit":
    command = input("Enter a command: ")

print("Program stopped.")
```

The word:

```
quit
```

acts as the sentinel.

This pattern appears frequently in command-line programs.

---

16. Building a Simple Menu

while loops are commonly used to create menus.

```
choice = ""

while choice != "4":
    print("\n1. Add")
    print("2. View")
    print("3. Delete")
    print("4. Exit")
```

```
    choice = input("Choose an option: ")

    if choice == "1":
        print("Adding...")
    elif choice == "2":
        print("Viewing...")
    elif choice == "3":
        print("Deleting...")
    elif choice == "4":
        print("Goodbye")
    else:
        print("Invalid choice")
```

The menu continues until the user chooses `4`.

This is a realistic use of while loops because the number of repetitions depends on user behavior.

---

17. while Loop With if

while and if often work together.

```
number = 1

while number <= 10:
    if number % 2 == 0:
        print(number, "is even")

    number += 1
```

The while loop controls repetition.

The if statement controls what happens during each repetition.

This combination is fundamental to programming.

---

18. Calculating a Sum

You can use a while loop to calculate totals.

```
number = 1
total = 0

while number <= 5:
    total += number
    number += 1
```

```
print(total)
```

Output:

```
15
```

The execution is:

```
total = 0

add 1 → total = 1
add 2 → total = 3
add 3 → total = 6
add 4 → total = 10
add 5 → total = 15
```

---

19. Calculating a Factorial

Factorial of 5:

```
5 × 4 × 3 × 2 × 1 = 120
```

Python:

```
number = 5
result = 1

while number > 1:
    result *= number
    number -= 1

print(result)
```

Output:

```
120
```

---

20. Searching With a while Loop

Suppose you want to find a particular value in a list.

```
numbers = [10, 20, 30, 40, 50]

index = 0
target = 40
```

```
while index < len(numbers):
    if numbers[index] == target:
        print("Found")
        break

    index += 1
```

Here:

```
index < len(numbers)
```

prevents the program from accessing an invalid index.

The loop moves through the list one position at a time.

---

21. while Loops and Lists

You can process a list with a while loop.

```
names = ["Ali", "Sara", "Ahmed"]

index = 0

while index < len(names):
    print(names[index])
    index += 1
```

Output:

```
Ali
Sara
Ahmed
```

However, when you simply need to process every item, a `for` loop is often clearer:

```
for name in names:
    print(name)
```

The choice depends on the problem.

---

22. while vs for

Both are loops, but they communicate different intentions.

`for`

Usually means:

> Repeat this operation for each item or for a known sequence/range.

Example:

```
for number in range(1, 6):
    print(number)
```

`while`

Usually means:

> Keep doing this while this condition remains true.

Example:

```
number = 1

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

A useful rule:

Situation	**Better fit**
Loop through a list	`for`
Repeat a known number of times	`for`
Repeat until a condition changes	`while`
User controls	`while`

when to stop	
Retry until success	`while`
Menu until exit	`while`

This is not an absolute rule, but it is a useful starting point.

---

23. Infinite Loops

An infinite loop is a loop that never reaches a false condition.

Example:

```
number = 1

while number <= 5:
    print(number)
```

Why?

Because `number` never changes.

The program keeps evaluating:

```
1 <= 5
```

which is always true.

Another example:

```
while True:
    print("Running...")
```

This intentionally creates an infinite loop.

Infinite loops aren't always mistakes. Sometimes they are deliberately used in applications, servers, games, and event-driven programs.

But an intentional infinite loop must have a way to exit.

---

24. Stopping an Infinite Loop With break

Python provides `break` to immediately leave a loop.

```
while True:
    command = input("Enter command: ")

    if command == "quit":
        break

    print("Command:", command)
```

The loop technically has:

```
while True:
```

but `break` provides the exit condition.

This pattern is common when the stopping condition is easier to express inside the loop.

---

25. while True With a Menu

```
while True:
    print("1. Start")
    print("2. Settings")
    print("3. Exit")

    choice = input("Choose: ")

    if choice == "1":
        print("Starting...")
    elif choice == "2":
        print("Opening settings...")
    elif choice == "3":
        print("Exiting...")
        break
    else:
        print("Invalid choice")
```

This is often easier to read than putting a complicated condition in the `while` statement.

---

26. The `continue` Statement

`continue` skips the rest of the current iteration and starts the next iteration.

Example:

```
number = 0
```

```
while number < 5:
    number += 1

    if number == 3:
        continue

    print(number)
```

Output:

```
1
2
4
5
```

When `number` becomes 3, `continue` skips:

```
print(number)
```

and the next iteration begins.

---

27. Be Careful With continue

A common mistake is forgetting to update the loop variable before `continue`.

Bad:

```
number = 1

while number <= 5:
    if number == 3:
        continue

    print(number)
    number += 1
```

When `number` becomes 3:

```
continue
```

runs before:

```
number += 1
```

So `number` remains 3 forever.

This creates an infinite loop.

A safer approach:

```
number = 1

while number <= 5:
    if number == 3:
        number += 1
        continue

    print(number)
    number += 1
```

---

28. Nested while Loops

A while loop can contain another while loop.

Example:

```
row = 1

while row <= 3:
    column = 1

    while column <= 3:
        print(row, column)
        column += 1

    row += 1
```

Output:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

The inner loop completes its repetitions for every iteration of the outer loop.

---

29. Nested Loop Mental Model

Think of it like this:

```
Outer loop starts
    ↓
Inner loop runs completely
    ↓
Outer loop moves once
    ↓
Inner loop runs completely again
    ↓
Repeat
```

Nested loops are useful for:

- grids
- tables
- game boards
- matrix operations
- combinations
- pattern generation

But they can become expensive when the data size grows.

---

30. Input Validation With while

One of the most practical uses of while loops is validating user input.

```
age = int(input("Enter your age: "))

while age < 0:
    print("Age cannot be negative.")
    age = int(input("Enter your age again: "))

print("Valid age:", age)
```

The loop continues until the input satisfies the required rule.

---

31. Multiple Validation Rules

You can combine conditions.

```
password = input("Enter a password: ")

while len(password) < 8:
```

```
    print("Password must contain at least 8 characters.")
    password = input("Enter again: ")

print("Password accepted.")
```

This pattern appears in:

- signup forms
- command-line tools
- configuration programs
- data entry systems
- validation workflows

---

32. Retry Logic

Suppose a program performs an operation that might fail.

A simple retry model:

```
attempts = 0
success = False

while attempts < 3 and not success:
    print("Trying...")

    success = True
    attempts += 1

print("Finished")
```

The loop has two stopping conditions:

```
attempts < 3
```

and:

```
not success
```

This is the foundation of retry logic used in many real systems.

In production applications, retry logic usually needs additional considerations such as delays, error types, maximum attempts, and logging.

---

33. A Real World Example: Login Attempts

```
correct_password = "python123"
attempts = 0
max_attempts = 3

while attempts < max_attempts:
    password = input("Enter password: ")

    if password == correct_password:
        print("Login successful")
        break

    attempts += 1
    print("Incorrect password")

if attempts == max_attempts:
    print("Account temporarily locked")
```

Important concepts here:

- condition
- counter
- user input
- `if`
- `break`
- maximum attempts

A real authentication system should never store passwords like this. This example is only for learning loop control.

---

34. A Real World Example: Shopping Cart Total

Imagine a simple shopping cart represented by a list.

```
prices = [100, 250, 50, 300]

index = 0
total = 0

while index < len(prices):
    total += prices[index]
    index += 1

print("Total:", total)
```

Output:

```
Total: 700
```

The loop processes one item at a time.

---

35. A Real World Example: Haas.dev Resource Processing

Imagine haas.dev has a list of resources that need to be processed.

```
resources = [
    "Python Variables",
    "Python Conditions",
    "Python Loops",
    "Python Functions"
]

index = 0

while index < len(resources):
    print("Processing:", resources[index])
    index += 1
```

Output:

```
Processing: Python Variables
Processing: Python Conditions
Processing: Python Loops
Processing: Python Functions
```

In a real application, the operation might involve:

- checking resource metadata
- generating a file
- validating a PDF
- updating a database
- creating an index
- processing API data

The loop provides the repeated execution mechanism.

---

36. A Real World Example: Game Loop Concept

Many games repeatedly perform work while the game is running.

Conceptually:

```
game_running = True

while game_running:
    print("Read player input")
    print("Update game state")
    print("Render game")

    # Some event eventually changes this
    game_running = False
```

A real game loop is much more complex, but the fundamental idea is similar:

> Keep processing while the application is running.

---

37. while Loops With Counters

A counter keeps track of how many iterations have occurred.

```
count = 0

while count < 5:
    print("Iteration:", count)
    count += 1
```

Counters are useful for:

- limiting attempts
- tracking retries
- counting processed items
- generating IDs
- monitoring progress

---

38. while Loops With Accumulators

An accumulator stores a running result.

```
total = 0
number = 1

while number <= 5:
    total += number
    number += 1
```

```
print(total)
```

Here:

```
total
```

is the accumulator.

This pattern is extremely common.

---

39. Counter vs Accumulator

These concepts are related but different.

Counter

Tracks occurrences.

```
count += 1
```

Accumulator

Builds a result.

```
total += price
```

Example:

```
count = 0
total = 0
number = 1

while number <= 5:
    count += 1
    total += number
    number += 1
```

At the end:

```
count = 5
total = 15
```

---

40. Off By One Errors

A common loop bug is processing one value too many or too few.

Example:

```
number = 1

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

Output:

```
1
2
3
4
```

It does not print 5 because:

```
5 < 5
```

is false.

If you want 5 included:

```
while number <= 5:
```

Understanding `<` versus `<=` is critical.

---

41. Forgetting Initialization

This is incorrect:

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

If `count` hasn't been defined earlier, Python raises:

```
NameError
```

Correct:

```
count = 0

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

---

42. Changing the Wrong Variable

Consider:

```
count = 1
number = 10

while count <= 5:
    print(number)
    number += 1
```

The condition depends on:

```
count
```

but the code updates:

```
number
```

`count` never changes.

Therefore the loop never terminates.

The update should affect the variable that controls the condition, directly or indirectly.

---

43. Changing a Variable Too Early

Consider:

```
number = 1

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

Output:

```
2
3
4
5
6
```

If the goal was to print:

```
1
2
3
```

```
4
5
```

then the update should happen after the print:

```
number = 1

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

The order of statements matters.

---

44. Debugging a while Loop

When a while loop behaves incorrectly, ask:

Question 1

What is the initial value?

```
count = ?
```

Question 2

What is the condition?

```
while count < ?:
```

Question 3

What changes inside the loop?

```
count += ?
```

Question 4

Can the condition ever become false?

Question 5

Is the variable changing in the expected direction?

Question 6

Could `break` or `continue` be skipping important code?

This debugging checklist can solve many loop problems.

---

45. Trace the Loop Manually

When confused, create a small table.

For:

```
number = 1

while number <= 3:
    print(number)
    number += 1
```

Trace it:

Iteration	number	Condition	Action
1	1	True	Print 1
2	2	True	Print 2
3	3	True	Print 3
4	4	False	Stop

This is one of the best ways to understand loops.

---

46. Common Mistakes

Mistake 1: Forgetting the update

```
count = 1

while count <= 5:
    print(count)
```

Possible result: infinite loop.

Mistake 2: Updating the wrong variable

```
count = 1
number = 0

while count <= 5:
    number += 1
```

`count` never changes.

Mistake 3: Wrong comparison

```
while number < 5:
```

instead of:

```
while number <= 5:
```

This can cause missing values.

Mistake 4: Incorrect indentation

Incorrect:

```
while count < 5:
print(count)
```

Correct:

```
while count < 5:
    print(count)
```

Mistake 5: Accidentally creating an infinite `while True`

```
while True:
    print("Running")
```

If intentional, provide a clear exit mechanism.

Mistake 6: Misusing `continue`

Make sure the next iteration can actually make progress.

---

47. Best Practices

1. Make the stopping condition obvious

Prefer:

```
while attempts < 3:
```

over unnecessarily complicated conditions.

2. Keep loop bodies focused

Avoid putting huge amounts of unrelated logic inside one loop.

3. Update the control state clearly

For example:

```
count += 1
```

4. Use meaningful names

Prefer:

```
attempts
```

over:

```
x
```

when the purpose matters.

5. Use `for` when iterating over a known collection

Don't use while loops just because you can.

6. Test boundary values

Check:

- first iteration
- last expected iteration
- condition becoming false
- empty input
- invalid input

---

48. while Loop and Program State

A deeper way to understand while loops is through **state**.

Suppose:

```
balance = 500

while balance > 0:
    balance -= 100
```

The program's state changes:

```
500
400
300
200
100
0
```

At:

```
0
```

the condition:

```
balance > 0
```

becomes false.

The loop stops.

So a while loop can be viewed as:

> Repeatedly transform the program's state until a condition is no longer true.

This mental model becomes very useful when you move toward algorithms and larger applications.

---

49. Termination

A good while loop should have a clear **termination argument**.

Ask:

> What eventually makes the condition false?

Example:

```
count = 0

while count < 10:
    count += 1
```

Termination is easy to explain:

- `count` starts at 0.
- Each iteration increases it by 1.
- Eventually it reaches 10.
- `count < 10` becomes false.

This is a fundamental programming skill.

---

50. while Loops and Algorithms

Many algorithms can be expressed using while loops.

For example, repeatedly dividing a number:

```
number = 100

while number > 1:
    number //= 2
    print(number)
```

Output:

```
50
25
12
6
3
1
```

The loop continues until the problem reaches a smaller state.

This idea appears in:

- searching
- numerical algorithms
- simulations
- parsing
- retry mechanisms
- state machines
- data processing

---

51. A Practical Example: Number Guessing Game

```
secret = 7
guess = None

while guess != secret:
    guess = int(input("Guess the number: "))

    if guess < secret:
        print("Too low")
    elif guess > secret:
        print("Too high")
    else:
        print("Correct!")
```

The number of iterations is unknown.

The loop ends when:

```
guess == secret
```

This is exactly the kind of problem where `while` makes sense.

---

52. A Better Version With Attempt Tracking

```
secret = 7
attempts = 0
max_attempts = 5

while attempts < max_attempts:
    guess = int(input("Guess the number: "))
    attempts += 1

    if guess == secret:
        print("Correct!")
        break
    elif guess < secret:
        print("Too low")
    else:
        print("Too high")

if attempts == max_attempts and guess != secret:
    print("Game over")
```

Now the program has:

- a condition
- a counter
- user input
- branching
- `break`
- a maximum number of attempts

This combines several fundamental Python concepts.

---

53. A Practical Example: Processing Until Empty

Suppose you repeatedly remove items from a list.

```
tasks = ["Study", "Code", "Practice"]

while tasks:
    task = tasks.pop(0)
    print("Completed:", task)
```

Python treats a non-empty list as truthy.

When the list becomes empty:

```
while tasks:
```

becomes false.

This is an important Python feature:

```
while tasks:
```

is effectively asking:

> While there are still tasks remaining...

---

54. Truthy and Falsy Values

Python allows many values to be used directly as conditions.

For example:

```
items = [1, 2, 3]

while items:
    print(items.pop())
```

The list is truthy while it contains elements.

An empty list is falsy.

Similarly:

```
name = "Hafsa"

while name:
    print(name)
    name = ""
```

An empty string is falsy.

Common falsy values include:

```
False
None
0
""
[]
{}
()
```

Understanding truthy and falsy values makes Python conditions much more expressive.

---

55. while With Functions

A while loop can call a function repeatedly.

```
def show_message():
    print("Hello from Python")

count = 0

while count < 3:
    show_message()
    count += 1
```

Output:

```
Hello from Python
Hello from Python
Hello from Python
```

As your programs grow, moving repeated logic into functions makes loops easier to understand.

---

56. while Loops and Error Handling

Later, while loops can work with `try` and `except` to repeatedly request valid input.

Example:

```
while True:
    try:
        age = int(input("Enter your age: "))
        break
    except ValueError:
        print("Please enter a valid number.")
```

The loop continues when invalid input causes an error.

Once valid input is received:

```
break
```

ends the loop.

This pattern is common in robust command-line programs.

---

57. Performance Considerations

A while loop itself is not automatically fast or slow.

Performance depends on what happens inside it and how many times it executes.

For example:

```
number = 0

while number < 1_000_000:
    number += 1
```

performs a very large number of iterations.

When working with large datasets, consider:

- whether the loop is necessary
- whether a built-in Python operation can do the work
- whether the algorithm can be improved
- whether unnecessary work is being repeated

The goal is not to avoid loops. The goal is to use the right algorithm.

---

58. while Loop vs Recursion

Some problems can be solved using either loops or recursion.

For example, repeatedly reducing a number could be written iteratively:

```
number = 5

while number > 0:
    print(number)
    number -= 1
```

Or recursively:

```
def countdown(number):
    if number == 0:
        return
```

```
    print(number)
    countdown(number - 1)
```

For simple repetition, a loop is often easier to understand and avoids creating additional function-call stack frames.

You will study recursion separately.

---

59. Engineering Thinking: Don't Just Ask "How Many Times?"

Beginners often think:

> A loop means repeat something 10 times.

That's only one use case.

A better question is:

> **What condition determines whether the work should continue?**

If you know the number of repetitions:

```
for number in range(10):
    ...
```

If you don't know how many repetitions will be required:

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

For example:

```
Keep asking until valid input
Keep retrying until successful
Keep running until the user exits
Keep processing until the queue is empty
Keep reading until the end condition appears
```

These are natural while-loop problems.

---

60. A Complete Example: Simple ATM Menu

```
balance = 1000

while True:
```

```
    print("\nATM")
    print("1. Check balance")
    print("2. Withdraw")
    print("3. Exit")

    choice = input("Choose an option: ")

    if choice == "1":
        print("Balance:", balance)

    elif choice == "2":
        amount = int(input("Enter amount: "))

        if amount <= 0:
            print("Invalid amount")
        elif amount > balance:
            print("Insufficient balance")
        else:
            balance -= amount
            print("Withdrawal successful")

    elif choice == "3":
        print("Goodbye")
        break

    else:
        print("Invalid option")
```

This example demonstrates how while loops become part of a complete program.

The loop manages the application state:

```
Show menu
   ↓
Read input
   ↓
Perform action
   ↓
Return to menu
   ↓
Repeat
   ↓
Exit when requested
```

---

61. Five Questions to Ask Before Writing a while Loop

Before coding, answer these:

1. What is the starting state?

Example:

```
attempts = 0
```

2. What condition means "keep going"?

Example:

```
attempts < 3
```

3. What changes after each iteration?

Example:

```
attempts += 1
```

4. What state makes the loop stop?

Example:

```
attempts == 3
```

5. Can the loop accidentally run forever?

If yes, redesign it before running the program.

This simple planning process prevents many bugs.

---

62. Mini Quiz

Question 1

What will this print?

```
x = 1

while x <= 3:
    print(x)
    x += 1
```

A. `1 2`

B. `1 2 3`

C. `0 1 2 3`

D. Infinite loop

Question 2

What is wrong with this code?

```
count = 1

while count <= 5:
    print(count)
```

A. The syntax is invalid.

B. The condition is invalid.

C. `count` never changes, so the loop can run forever.

D. `print()` cannot be used inside a loop.

Question 3

Which loop is more natural when you need to keep asking a user for input until they enter `"quit"`?

A. `if`

B. `for`

C. `while`

D. `match`

---

63. Mini Quiz Answers

Question 1

B. `1 2 3`

The loop executes while `x <= 3`.

Question 2

C. `count` never changes

The condition remains true forever.

Question 3

C. `while`

The number of repetitions depends on when the user enters `"quit"`.

---

64. 5 Minute Challenge

Write a program that prints numbers from 10 down to 1.

Expected output:

```
10
9
8
7
6
5
4
3
2
1
```

Requirements:

- Use a `while` loop.
- Start with a variable containing `10`.
- Decrease it by `1` each iteration.
- Do not manually write ten `print()` statements.

Then modify it so that after `1`, it prints:

```
Blast off!
```

---

65. Practice Exercises

Exercise 1: Even Numbers

Print all even numbers from 2 to 20 using a while loop.

Exercise 2: Countdown

Create a countdown from 10 to 1.

Exercise 3: Sum

Calculate the sum of numbers from 1 to 100.

Exercise 4: Password

Keep asking for a password until the user enters the correct password.

Exercise 5: Maximum Attempts

Allow a user only three password attempts.

Exercise 6: Menu

Create a menu:

```
1. Start
2. Help
3. Exit
```

Keep showing it until the user selects `3`.

Exercise 7: Number Guessing

Choose a secret number and repeatedly ask the user to guess it until they get it right.

Exercise 8: List Processing

Given:

```
numbers = [10, 20, 30, 40, 50]
```

Use a while loop to calculate the total.

---

66. Mini Project: Console Task Manager

Build a small task manager using a while loop.

The program should repeatedly display:

```
1. Add task
2. View tasks
3. Remove task
4. Exit
```

Requirements:

Add task

Ask the user for a task and add it to a list.

View tasks

Display all current tasks.

Remove task

Ask which task should be removed.

Exit

Stop the program.

A simplified starting structure:

```
tasks = []

while True:
    print("\n1. Add task")
    print("2. View tasks")
    print("3. Remove task")
    print("4. Exit")

    choice = input("Choose: ")

    if choice == "1":
        task = input("Enter task: ")
        tasks.append(task)

    elif choice == "2":
        for task in tasks:
            print(task)

    elif choice == "3":
        task = input("Enter task to remove: ")

        if task in tasks:
            tasks.remove(task)

    elif choice == "4":
        break

    else:
        print("Invalid choice")
```

This project combines:

- variables
- lists
- input
- conditions
- while loops
- for loops
- `break`
- list methods

It is a good example of how one programming concept becomes part of a larger system.

---

67. Interview Questions

Beginner

1. What is a while loop in Python?

A while loop repeatedly executes a block of code while its condition is true.

2. When should you use a while loop?

When repetition depends on a condition and the number of iterations may not be known beforehand.

3. What happens if the condition is initially false?

The loop body does not execute even once.

4. What causes an infinite loop?

A condition that never becomes false, often because the loop's control state is never updated correctly.

5. How do you exit a loop immediately?

Use:

```
break
```

Intermediate

6. What is the difference between `break` and `continue`?

`break` exits the entire loop. `continue` skips the current iteration and starts the next one.

7. Can a while loop contain another while loop?

Yes. This is called a nested loop.

8. Can a while loop have an `else` block?

Yes.

```
number = 1

while number <= 3:
    print(number)
    number += 1
else:
    print("Loop finished")
```

The `else` block runs when the loop finishes normally rather than being terminated by `break`.

9. Can a while loop use multiple conditions?

Yes.

```
while attempts < 3 and not success:
    ...
```

10. What is a sentinel value?

A special value that signals the program to stop looping.

Example:

```
while command != "quit":
    ...
```

---

68. while...else

Python has a useful feature that beginners often overlook.

```
number = 1

while number <= 3:
    print(number)
    number += 1
else:
    print("Finished")
```

Output:

```
1
2
3
Finished
```

But if the loop exits using `break`:

```
number = 1

while number <= 3:
    if number == 2:
        break

    print(number)
    number += 1
else:
    print("Finished")
```

The `else` block does not run.

This can be useful for search operations where you want to distinguish between:

- loop completed normally
- loop stopped because something was found

---

69. Search Example With while...else

```
numbers = [10, 20, 30, 40]

index = 0
target = 50

while index < len(numbers):
    if numbers[index] == target:
        print("Found")
        break

    index += 1
else:
    print("Not found")
```

Output:

```
Not found
```

The `else` executes because the loop ended normally without `break`.

---

70. Important Distinction: Condition vs Event

A while loop can be controlled by either a changing condition or an event.

Condition-based:

```
count = 0

while count < 5:
    count += 1
```

Event-based:

```
while True:
    command = input("Command: ")

    if command == "quit":
        break
```

In the second example, the program doesn't know when it will stop until an external event occurs.

This is common in interactive software.

---

71. The Core Pattern to Remember

Most while loops can be understood using this structure:

```
state = initial_value

while condition(state):
    perform_work(state)
    update_state(state)
```

For example:

```
attempts = 0

while attempts < 3:
    print("Attempt:", attempts + 1)
    attempts += 1
```

This pattern is more valuable than memorizing individual examples.

---

72. Cheat Sheet

Basic while

```
while condition:
    # code
```

Counter

```
count = 0

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

Countdown

```
count = 5

while count > 0:
    print(count)
    count -= 1
```

User controlled loop

```
while True:
    command = input("> ")

    if command == "quit":
        break
```

Skip iteration

```
while condition:
    if something:
        continue
```

Use carefully so the loop still makes progress.

Multiple conditions

```
while attempts < 3 and not success:
    ...
```

List processing

```
while items:
    item = items.pop()
```

Nested while

```
while outer_condition:
    while inner_condition:
        ...
```

while...else

```
while condition:
    ...
else:
    ...
```

The `else` runs when the loop finishes normally.

---

73. Key Takeaways

1. A `while` loop repeats code while a condition is true.

2. The condition is checked before every iteration.
3. If the condition is initially false, the loop runs zero times.
4. A loop needs a path toward termination.
5. Updating the loop's state is often what makes termination possible.
6. Forgetting the update can create an infinite loop.
7. `break` exits the loop immediately.
8. `continue` skips the current iteration.
9. `while True` is useful for event-driven or user-controlled loops when paired with a clear exit condition.
10. `while` is especially useful when you don't know the number of repetitions in advance.
11. `for` is often clearer for iterating through known sequences or ranges.
12. Counters track iterations; accumulators build results.
13. Nested while loops can solve grid and repeated-subproblem problems but may increase work significantly.
14. `while...else` can distinguish normal completion from `break`.
15. The most important question is not "How many times should this run?" but:

> **"What condition determines whether this should continue?"**

Final Practice Task

Build a **Python Study Session Tracker**.

The program should:

1. Start with `minutes = 0`.
2. Ask the user whether they studied for another 10 minutes.
3. Keep repeating while the user chooses `"yes"`.
4. Add 10 minutes after every successful response.
5. Stop when the user chooses `"no"`.
6. Display the total study time.

Example:

```
Did you study another 10 minutes? yes
Did you study another 10 minutes? yes
Did you study another 10 minutes? yes
Did you study another 10 minutes? no

Total study time: 30 minutes
```

Then improve it by:

- rejecting invalid answers
- showing the current total after every session
- adding a daily goal
- stopping automatically when the goal is reached

This exercise forces you to think about **state, conditions, updates, input validation, and termination**—the core skills behind while loops.

Visit haas.dev for more resources and guides.

haas.dev

https://dev-roast-app.vercel.app