Module 3: Control Structures in Python

Welcome to Module 3! In the previous module, you learned how to write basic Python code, handle inputs, and perform calculations. Now, we'll dive into **control structures**, which are fundamental building blocks that allow your programs to make decisions and repeat actions. This is where your code truly becomes "smart"!

Chapter 1: Conditional Statements (Decision Making)

Conditional statements allow your program to execute different blocks of code based on whether a certain condition is `True` or `False`. This is how your programs "make decisions."

1.1 Introduction to Control Flow

Normally, Python code executes sequentially, one line after another, from top to bottom. **Control flow** statements change this default order of execution. They allow your program to:

- **Make Decisions:** Execute specific code only if a condition is met (e.g., `if` statements).
- **Repeat Actions:** Execute a block of code multiple times (e.g., `for` and `while` loops).

1.2 Relational (Comparison) Operators

Before we make decisions, we need to be able to *compare* values. **Relational operators** (also known as comparison operators) are used to compare two values and evaluate to either `True` or `False`. These `True` or `False` values (known as **Booleans**) are the foundation of all decision-making in programming.

Operator	Meaning	Console Example	Result	Explanation
`==`	Equal to	`5 == 5`	`True`	Checks if two values are exactly the same.
		`'hi' == 'Hi'`	`False`	Case-sensitive for strings.
`!=`	Not equal to	`10 != 7`	`True`	Checks if two values are different.
		`5 != 5`	`False`	
`>`	Greater than	`10 > 5`	`True`	Checks if the left value is larger than the right.
`<`	Less than	`3 < 8`	`True`	Checks if the left value is smaller than the right.
`>=`	Greater than or equal to	`7 >= 7`	`True`	Checks if the left value is larger than or equal to the right.
		`7 >= 5`	`True`	
`<=`	Less than or equal to	`4 <= 4`	`True`	Checks if the left value is smaller than or equal to the right.
		`4 <= 6`	`True`	

Examples in the Console:

Python
```
>>> x = 10
>>> y = 7

>>> x == y
False
>>> x != y
True
>>> x > y
True
>>> x < y
False
>>> x >= 10
True
>>> y <= 7
True

>>> "apple" == "orange"
False
>>> "Python" != "python" # Case sensitivity matters!
True
>>>
```

These Boolean results (`True` or `False`) are what drive the `if` statements we're about to discuss!

1.3 The `if` Statement

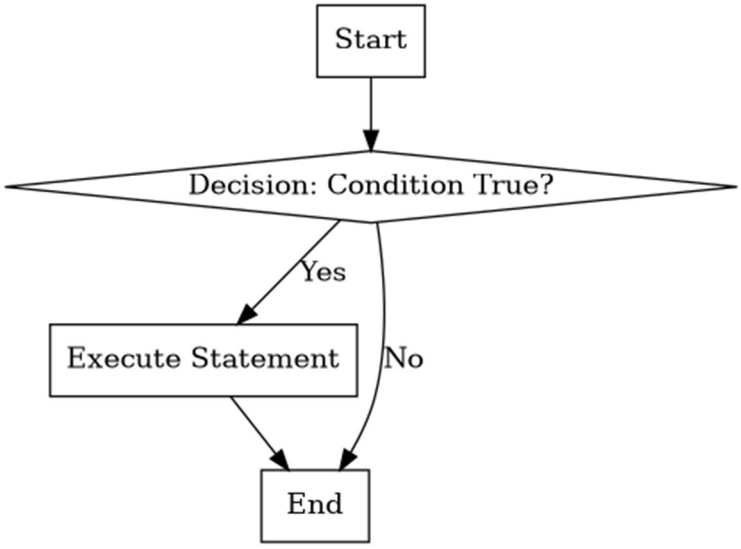
The `if` statement is the simplest decision-making statement. It executes a block of code only if a specified condition evaluates to `True`.

Syntax:

Python
```
if condition:
    # Code to execute if the condition is True
    # (This code block MUST be indented)
```

- `condition`: An expression (often using relational operators) that evaluates to either `True` or `False`.
- `: (colon)`: This is crucial! It marks the end of the `if` statement header and indicates that an indented code block follows.
- **Indentation:** The lines of code *inside* the `if` block must be indented (typically 4 spaces). Python uses indentation to define code blocks, unlike other languages that use curly braces `{}`.

Flowchart for `if` statement:



Example 1: Simple Age Check (Console)

Python
```
>>> age = 18
>>> if age >= 18:
...     print("You are an adult.")
...
You are an adult.
>>> age = 16
>>> if age >= 18:
...     print("You are an adult.")
... # Nothing is printed because the condition (16 >= 18) is False
>>>
```

Example 2: Check for a Positive Number (Script)

Create a file named `positive_checker.py`:

Python
```
# positive_checker.py
number = float(input("Enter a number: "))

if number > 0:
    print("The number is positive.")

print("Program finished.") # This line always runs, regardless of the if
condition
```

Interaction:

```
Enter a number: 10
The number is positive.
Program finished.

Enter a number: -5
Program finished.
```

Common `if` Statement Errors:

- **Missing Colon (`:`):** This is a very common `SyntaxError`.

  Python

  ```
  >>> x = 10
  >>> if x > 5
  SyntaxError: expected ':'
  ```

- **Incorrect Indentation (`IndentationError`):** Another frequent mistake.

  Python

  ```
  >>> x = 10
  >>> if x > 5:
  ... print("X is greater than 5") # Missing indentation here
  ...
  IndentationError: expected an indented block
  ```

  Remember: Python is very strict about indentation. All lines within the same code block must have the same level of indentation.

1.4 The `if-else` Statement

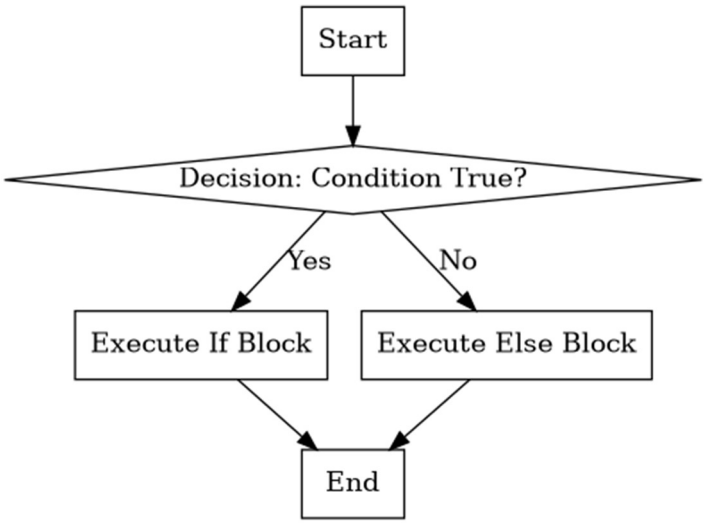
The `if-else` statement allows your program to execute one block of code if the condition is `True`, and a *different* block of code if the condition is `False`. It provides an alternative path.

Syntax:

Python
```
if condition:
    # Code to execute if the condition is True
else:
    # Code to execute if the condition is False
    # (This code block MUST also be indented)
```

- `else:` This keyword introduces the block of code that runs when the `if` condition is `False`. It also requires a colon.

Flowchart for `if-else` statement:



Example 1: Even or Odd Number (Console)

Python
```
>>> number = 7
>>> if number % 2 == 0:
...     print("The number is even.")
... else:
...     print("The number is odd.")
...
The number is odd.

>>> number = 10
>>> if number % 2 == 0:
...     print("The number is even.")
... else:
...     print("The number is odd.")
...
The number is even.
>>>
```

Example 2: Login Check (Script)

Create a file named `login_checker.py`:

Python
```
# login_checker.py
username = input("Enter username: ")
password = input("Enter password: ")

if username == "admin" and password == "secret123": # Using 'and' for
multiple conditions
    print("Login successful! Welcome, admin.")
else:
    print("Login failed. Invalid username or password.")
```

Interaction:

```
Enter username: admin
Enter password: secret123
Login successful! Welcome, admin.

Enter username: user
Enter password: wrongpass
Login failed. Invalid username or password.
```

Common `if-else` Statement Errors:

- **`else` without `if`:** An `else` statement must always follow an `if` or `elif` statement.

  Python

  ```
  >>> x = 10
  >>> else:
  SyntaxError: invalid syntax
  ```

- **Indentation Issues:** Similar to `if` statements, consistent indentation is vital for both `if` and `else` blocks.

1.5 The `elif` Statement (Else If)

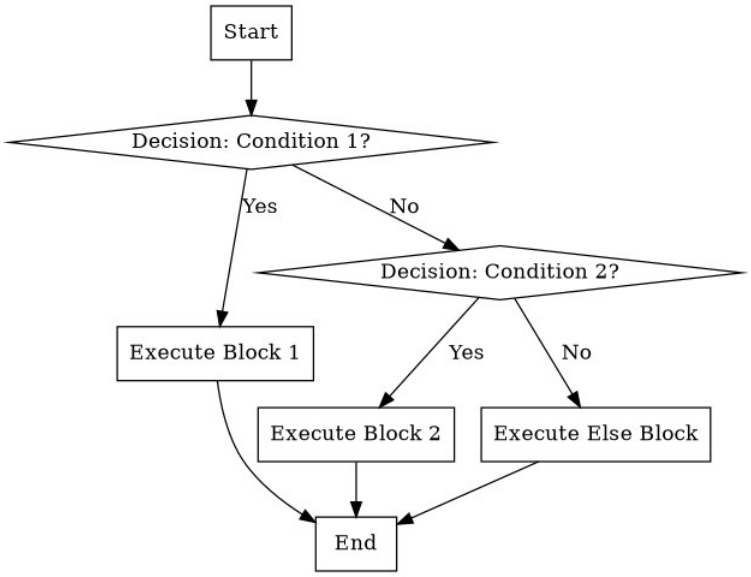
When you have more than two possible outcomes (more than just `True` or `False`), you use the `elif` (short for "else if") statement. This allows you to check multiple conditions sequentially.

Syntax:

Python
```
if condition1:
    # Code if condition1 is True
elif condition2:
    # Code if condition1 is False AND condition2 is True
elif condition3:
    # Code if condition1 is False AND condition2 is False AND condition3 is
True
else:
    # Code if all above conditions are False
```

- Python checks conditions from top to bottom. The first `if` or `elif` condition that evaluates to `True` will have its block executed, and then the rest of the `elif`/`else` chain is skipped.

Flowchart for `if-elif-else` statement:



Example: Grading System (Script)

Create a file named `grade_calculator.py`:

Python

```
# grade_calculator.py
score_str = input("Enter your test score (0-100): ")
score = int(score_str) # Convert score to an integer

if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
elif score >= 60:
    print("Grade: D")
else:
    print("Grade: F")

print("Grading complete.")
```

Interaction:

```
Enter your test score (0-100): 85
Grade: B
Grading complete.

Enter your test score (0-100): 95
Grade: A
Grading complete.

Enter your test score (0-100): 55
Grade: F
Grading complete.
```

Common `elif` Statement Errors:

- **Order of Conditions:** The order of `elif` statements matters! If you put a broader condition before a narrower one, the narrower one might never be reached.

  Python

  ```
  score = 75
  if score >= 60:
      print("Passed") # This will always execute if score is 60 or
  higher
  elif score >= 70:  # This condition will never be checked if score >=
  60 is True
      print("Good")
  ```

  Correction: Place more specific or stricter conditions first:

  Python

  ```
  score = 75
  if score >= 70:
      print("Good")
  elif score >= 60:
      print("Passed")
  ```

1.6 Logical Operators (`and`, `or`, `not`)

Logical operators combine multiple conditions to form more complex ones.

- `and`: `True` if **both** conditions are `True`.
- `or`: `True` if **at least one** condition is `True`.
- `not`: Reverses the Boolean value of a condition (`True` becomes `False`, `False` becomes `True`).

Examples in the Console:

Python

```
>>> age = 25
>>> has_license = True

>>> if age >= 18 and has_license:
...     print("Eligible to drive.")
...
Eligible to drive.

>>> is_raining = False
>>> is_cold = True

>>> if is_raining or is_cold:
...     print("Bring a jacket.")
...
Bring a jacket.

>>> if not is_raining:
...     print("It's not raining.")
```

```
...
It's not raining.
>>>
```

Chapter 2: Repetitive Statements (Loops)

Loops are powerful tools that allow your program to perform the same actions multiple times without writing the same code repeatedly. They are fundamental for automating tasks and processing collections of data efficiently.

2.1 Why Loops? The Power of Repetition

Imagine you need to:

- Print "Happy Birthday!" for each of 50 friends.
- Calculate the average score of 100 students.
- Process every line in a very long text file.
- Keep asking a user for input until they enter a specific word.

Without loops, you'd be writing the same lines of code over and over, which is tedious, error-prone, and makes your code very long and hard to manage. Loops solve this by telling Python: "Execute this block of code *this many times*," or "Execute this block of code *until this condition is no longer true*."

Python offers two main types of loops:

- **`for` loop:** Used for **definite iteration**. This means you know beforehand how many times you want to loop, or you want to process each item in a collection (like a list or string). It "iterates over" a sequence of items.
- **`while` loop:** Used for **indefinite iteration**. This means the loop continues as long as a certain condition remains `True`. You don't necessarily know the number of repetitions when the loop starts; it keeps going until the condition becomes `False`.

Let's explore each in detail!

2.2 The `range()` Function

The `range()` function is your best friend when you need to perform an action a specific number of times with a `for` loop, especially when dealing with numerical sequences. It generates a sequence of numbers, but it does so "on-the-fly" without creating a big list of numbers in memory, making it very efficient.

Syntax Variations:

1. **`range(stop)`**:
    - Generates numbers starting from `0` (default) up to, but **not including**, the `stop` value.
    - Example: `range(5)` will produce `0, 1, 2, 3, 4`.

2. **`range(start, stop)`**:
    - Generates numbers starting from `start` up to, but **not including**, the `stop` value.
    - Example: `range(2, 8)` will produce `2, 3, 4, 5, 6, 7`.
3. **`range(start, stop, step)`**:
    - Generates numbers starting from `start`, going up to (but **not including**) `stop`, incrementing by `step` each time.
    - `step` can be positive (for increasing sequences) or negative (for decreasing sequences).
    - Example: `range(1, 10, 2)` will produce `1, 3, 5, 7, 9`.
    - Example: `range(10, 0, -1)` will produce `10, 9, 8, 7, 6, 5, 4, 3, 2, 1`.

Examples in the Console:

To actually *see* the numbers `range()` generates, we can convert it to a `list` (though in `for` loops, you don't usually do this).

Python
```
>>> # Example 1: range(stop)
>>> print(list(range(5)))
[0, 1, 2, 3, 4] # 5 numbers, starting from 0, ending before 5

>>> # Example 2: range(start, stop)
>>> print(list(range(2, 8)))
[2, 3, 4, 5, 6, 7] # 6 numbers, starting from 2, ending before 8

>>> # Example 3: range(start, stop, step) - Positive step
>>> print(list(range(1, 10, 2)))
[1, 3, 5, 7, 9] # Numbers starting from 1, adding 2 each time, until 10 is
reached/exceeded

>>> # Example 4: range(start, stop, step) - Negative step (countdown!)
>>> print(list(range(5, 0, -1)))
[5, 4, 3, 2, 1] # Numbers starting from 5, subtracting 1, until 0 is
reached/exceeded

>>> print(list(range(0, 10, 3))) # Every third number
[0, 3, 6, 9]
>>>
```

Key Takeaway: `range()` is like a recipe for a sequence of numbers. A `for` loop is what *follows* that recipe, taking each number one by one.

2.3 The `for` Loop: Iterating a Fixed Number of Times (with `range()`)

This is the most common pattern for using a `for` loop when you want to repeat an action a specific number of times.

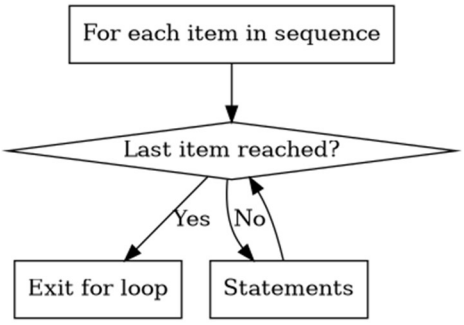
Syntax:

Python
```
for iteration_variable in range(start, stop, step):
    # Code block to be executed repeatedly
```

```
    # (MUST be indented)
```

- `iteration_variable`: This is a temporary variable that Python creates for the loop. In each repetition (or **iteration**) of the loop, this variable will automatically take on the *next value* generated by `range()`. You can name it anything you like (e.g., `i`, `num`, `count`), but `i` or `j` are common for simple numerical loops.

Flow chart:



How it Works (Step-by-Step Trace):

Let's trace `for i in range(3): print(i)`

1. **Initialization:** The `for` loop begins. `range(3)` is ready to produce `0, 1, 2`.
2. **1st Iteration:**
   - The `iteration_variable` `i` is assigned the *first* value from `range(3)`, which is `0`.
   - The code inside the loop (`print(i)`) is executed. Output: `0`
3. **2nd Iteration:**
   - `i` is assigned the *next* value from `range(3)`, which is `1`.
   - The code inside the loop (`print(i)`) is executed. Output: `1`
4. **3rd Iteration:**
   - `i` is assigned the *next* value from `range(3)`, which is `2`.
   - The code inside the loop (`print(i)`) is executed. Output: `2`
5. **End of Loop:** `range(3)` has no more numbers to provide. The loop terminates. Execution continues with any code *after* the loop (not indented).

Example 1: Counting to 5 (Console)

Python

```
>>> for count in range(1, 6): # Loop from 1 up to (but not including) 6
...     print("Counting:", count)
...
Counting: 1
Counting: 2
Counting: 3
Counting: 4
```

```
Counting: 5
>>> print("Loop finished.")
Loop finished.
>>>
```

Example 2: Sum of First N Numbers (Script)

Let's calculate the sum of numbers from 1 up to a number provided by the user. Create a file named `sum_numbers.py`:

Python
```
# sum_numbers.py
num_str = input("Enter a positive integer: ")
n = int(num_str)

total_sum = 0 # Initialize a variable to store the sum

print(f"Calculating sum from 1 to {n}...")
for current_num in range(1, n + 1): # Loop from 1 up to and including 'n'
    total_sum += current_num       # Add the current number to total_sum
(shorthand for total_sum = total_sum + current_num)
    print(f"Adding {current_num}. Current sum: {total_sum}") # See the sum
grow

print(f"\nThe final sum of numbers from 1 to {n} is: {total_sum}")
```

Interaction:

```
Enter a positive integer: 4
Calculating sum from 1 to 4...
Adding 1. Current sum: 1
Adding 2. Current sum: 3
Adding 3. Current sum: 6
Adding 4. Current sum: 10

The final sum of numbers from 1 to 4 is: 10
```

Common `for` Loop Errors:

- **Forgetting `range()` or an Iterable:** The `for` loop needs something to iterate over.

  Python
  ```
  >>> for i in 5:
  ...     print(i)
  ...
  TypeError: 'int' object is not iterable
  ```

  Correction: `for i in range(5):`

- **Incorrect Indentation:** This remains the most common `IndentationError`.

  Python
  ```
  for num in range(3):
  ```

```
    print(num) # This line is not indented!
```

- **Off-by-One Errors with `range()`:** A frequent mistake is miscalculating the `stop` value. Remember `range()` goes *up to but not including* the `stop` value. If you want to include `N`, your `stop` should be `N+1`. If you want `0` to `N-1`, then `N` is your `stop`.

2.4 The `for` Loop: Iterating Over Collections (Strings & Lists)

Beyond `range()`, the `for` loop is incredibly powerful for directly accessing each element within a collection (like a string or a list). You don't need `range()` if you simply want to process *each item*.

How it Works:

When you iterate over a string, the `iteration_variable` takes on each character of the string, one by one. When you iterate over a list, the `iteration_variable` takes on each item of the list, one by one.

Syntax:

Python
```
for item_variable in collection:
    # Code to be executed for each item in the collection
    # 'item_variable' will take on the value of each element in turn
```

Example 1: Iterating through a String (Console)

Python
```
>>> message = "Hello"
>>> for char in message: # 'char' will be 'H', then 'e', then 'l', then
'l', then 'o'
...     print("Current character:", char)
...
Current character: H
Current character: e
Current character: l
Current character: l
Current character: o
>>>
```

Example 2: Processing Items in a List (Script)

Let's find the largest number in a list of grades. Create a file named `find_largest.py`:

Python
```
# find_largest.py
grades = [85, 92, 78, 95, 88, 70] # This is a Python list!

largest_grade = 0 # Initialize with a value smaller than any possible grade

print("Checking grades...")
for grade in grades: # 'grade' will take on each value in the 'grades' list
    print(f"Currently checking: {grade}")
```

```
    if grade > largest_grade: # Compare current grade with the largest
found so far
        largest_grade = grade # If current grade is larger, update
largest_grade

print(f"\nThe highest grade is: {largest_grade}")
```

Interaction:

```
Checking grades...
Currently checking: 85
Currently checking: 92
Currently checking: 78
Currently checking: 95
Currently checking: 88
Currently checking: 70

The highest grade is: 95
```

Example 3: Counting Vowels in a Word

Python

```
# count_vowels.py
word = input("Enter a word: ").lower() # Get word and convert to lowercase
immediately
vowel_count = 0
vowels = "aeiou" # Define the vowels we are looking for

for char in word: # Loop through each character of the word
    if char in vowels: # Check if the current character is one of the
vowels
        vowel_count += 1

print(f"The word '{word}' has {vowel_count} vowels.")
```

Interaction:

```
Enter a word: Programming
The word 'programming' has 3 vowels.

Enter a word: rhythm
The word 'rhythm' has 0 vowels.
```

Important Note: Immutability vs. Mutability

- **Strings are Immutable:** You cannot change individual characters of a string *in place* using a `for` loop. If you want a modified string, you must build a *new* string.
- **Lists are Mutable:** You *can* change elements within a list while iterating (though it often requires careful handling, especially if adding/removing elements from the list you're iterating over). We'll cover lists in more detail in a future module.

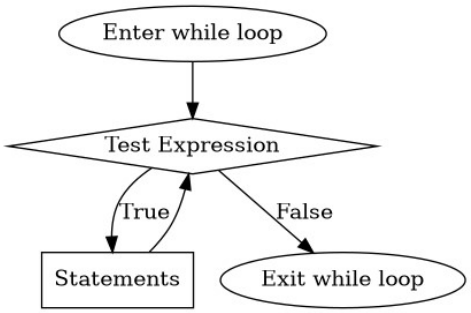
2.5 The `while` Loop: Repeating Until a Condition is Met

The `while` loop is used for **indefinite iteration**. It keeps executing a block of code repeatedly *as long as* a specified condition remains `True`. You typically use a `while` loop when you don't know the exact number of times you need to loop beforehand.

Syntax:

Python
```
while condition:
    # Code block to execute as long as the condition is True
    # (MUST be indented)
    # IMPORTANT: There MUST be code inside the loop that changes
    # variables involved in the 'condition' so it eventually becomes False,
    # Otherwise, you will create an INFINITE LOOP!
```

Flow Chart:



Key Elements of a `while` Loop (The "ICU" principle):

For a `while` loop to work correctly and avoid infinite loops, remember the "ICU" principle:

1. **I - Initialization:** A variable involved in the condition *must* be initialized *before* the loop starts.
2. **C - Condition:** The `while` statement checks this condition *before* each iteration.
3. **U - Update:** Something *inside* the loop's body must update the variable(s) involved in the condition, moving it closer to the state where the condition becomes `False`.

How it Works (Step-by-Step Trace):

Let's trace

```
count = 0; while count < 3: print(count); count += 1
```

1. **Initialization:** `count` is set to `0`.
2. **Check Condition (1st time):** Is `count < 3`? (`0 < 3`) -> `True`.
3. **1st Iteration:**
    - Code inside loop executes: `print(count)` (Output: `0`)
    - Code inside loop executes: `count += 1` (now `count` is `1`)
4. **Check Condition (2nd time):** Is `count < 3`? (`1 < 3`) -> `True`.
5. **2nd Iteration:**
    - Code inside loop executes: `print(count)` (Output: `1`)
    - Code inside loop executes: `count += 1` (now `count` is `2`)

6. **Check Condition (3rd time):** Is `count < 3`? (`2 < 3`) -> `True`.
7. **3rd Iteration:**
    - Code inside loop executes: `print(count)` (Output: `2`)
    - Code inside loop executes: `count += 1` (now `count` is `3`)
8. **Check Condition (4th time):** Is `count < 3`? (`3 < 3`) -> `False`.
9. **End of Loop:** The condition is `False`, so the loop terminates. Execution continues with code after the loop.

Example 1: Simple Countdown (Console)

Python
```
>>> num = 5
>>> while num > 0: # Loop as long as num is positive
...     print(num)
...     num -= 1 # Decrement num; this is the 'update' that makes the loop
eventually stop
...
5
4
3
2
1
>>> print("Blast off!")
Blast off!
>>>
```

Example 2: User Input Validation (Script)

A common use case for `while` loops is to repeatedly ask the user for input until they provide valid data. Create a file named `valid_age.py`:

Python
```
# valid_age.py
age = -1 # Initialize age to an invalid value to ensure the loop runs at
least once

while age < 0 or age > 120: # Loop as long as age is outside the reasonable
range
    age_str = input("Please enter your age (0-120): ")
    try: # Try to convert to integer (more advanced topic: error handling!)
        age = int(age_str)
        if age < 0 or age > 120:
            print("Age must be between 0 and 120. Please try again.")
    except ValueError: # If conversion fails (e.g., user types "abc")
        print("Invalid input. Please enter a number.")

print(f"Thank you! Your age is: {age}")
```

Interaction:

```
Please enter your age (0-120): twenty
Invalid input. Please enter a number.
Please enter your age (0-120): -5
Age must be between 0 and 120. Please try again.
Please enter your age (0-120): 150
Age must be between 0 and 120. Please try again.
```

```
Please enter your age (0-120): 30
Thank you! Your age is: 30
```

Common `while` Loop Errors:

- **Infinite Loop:** This is the most critical error with `while` loops. It happens when the condition *never* becomes `False`, causing the loop to run endlessly. Your program will appear to freeze, or it might continuously print output.

  Python

  ```
  >>> counter = 0
  >>> while counter < 5:
  ...     print("Stuck!") # This will print "Stuck!" forever
  ...     # Missing: counter += 1
  ```

  How to stop an infinite loop: In a terminal, press `Ctrl + C`. In an IDE like PyCharm, look for a "Stop" button (often a red square). **Prevention:** Always double-check your "Update" step.

- **Off-by-One Errors:** Ensuring the loop runs the exact desired number of times. Be precise with your relational operators (<, <=, >).

  Python

  ```
  # Goal: Print numbers 1 to 3
  count = 1
  while count < 3: # This will print 1, 2 (count becomes 3, condition
  (3 < 3) is False)
      print(count)
      count += 1
  # Corrected:
  count = 1
  while count <= 3: # This will print 1, 2, 3 (count becomes 4,
  condition (4 <= 3) is False)
      print(count)
      count += 1
  ```

Chapter 3: Coding Challenges for Control Structures

Time to put your decision-making and looping skills to the test!

Challenge 1: Number Classifier

Goal: Write a program that takes a single number from the user and classifies it as positive, negative, or zero. Also, check if it's even or odd if it's not zero.

Concepts Covered: `if-elif-else`, `input()`, type conversion (`int()` or `float()`), modulus operator (`%`), nested `if-else`.

Requirements:

1. Ask the user to enter an integer.
2. Use `if-elif-else` to check:
    - If the number is `> 0`, print "The number is positive."
    - If the number is `< 0`, print "The number is negative."
    - If the number is `== 0`, print "The number is zero."
3. **Inside** the positive and negative branches (i.e., if the number is not zero), add another `if-else` statement to check if the number is even or odd.
    - Hint: A number is even if `number % 2 == 0`.

Example Interaction:

```
Enter an integer: 7
The number is positive.
The number is odd.

Enter an integer: -4
The number is negative.
The number is even.

Enter an integer: 0
The number is zero.
```

Challenge 2: Countdown Timer

Goal: Create a simple countdown from a user-specified number to 1, then print "Blast off!". Implement this using *both* a `for` loop and a `while` loop for practice.

Concepts Covered: `for` loop, `while` loop, `range()`, `input()`, type conversion (`int()`).

Requirements:

Part A: Using a `for` loop

1. Ask the user for a starting number for the countdown.
2. Use a `for` loop with `range()` to count down from that number to 1. (Hint: think about the `step` argument for `range()`).
3. Print each number in the countdown.
4. After the loop, print "Blast off!".

Part B: Using a `while` loop

1. Repeat steps 1-4 from Part A, but this time use a `while` loop. Ensure you properly initialize a counter variable and decrement it within the loop.

Example Interaction (for both parts):

```
--- Countdown (for loop) ---
Enter a starting number: 5
5
4
3
2
1
Blast off!

--- Countdown (while loop) ---
Enter a starting number: 3
3
2
1
Blast off!
```

Challenge 3: Vowel Counter

Goal: Write a program that counts the number of vowels (a, e, i, o, u) in a word provided by the user.

Concepts Covered: `for` loop (iterating over strings), `if` statement, string methods (`.lower()`), counters (`+=`).

Requirements:

1. Ask the user to enter a word.
2. Initialize a variable `vowel_count` to `0`.
3. Use a `for` loop to iterate through each character in the input word.
4. Inside the loop, use an `if` statement to check if the current character is a vowel. (Hint: convert the character to lowercase first to handle both 'A' and 'a'. You might use `if char in 'aeiou':`).
5. If it's a vowel, increment `vowel_count`.
6. After the loop, print the total number of vowels found.

Example Interaction:

```
Enter a word: Programming
Number of vowels: 3

Enter a word: Hello World
Number of vowels: 3

Enter a word: rhythm
Number of vowels: 0
```

Challenge 4: Simple Calculator Menu

Goal: Create a basic calculator that allows the user to choose an operation (add, subtract, multiply, divide) and perform it, continuing until they choose to exit.

Concepts Covered: `while` loop (for menu), `if-elif-else` (for operation choice), `input()`, type conversion (`float()`), arithmetic operators, error handling for division by zero.

Requirements:

1. Use a `while` loop to repeatedly display a menu of options:
    - 1. Add
    - 2. Subtract
    - 3. Multiply
    - 4. Divide
    - 5. Exit
2. Prompt the user to enter their choice (1-5).
3. If the choice is between 1 and 4:
    - Ask the user for two numbers.
    - Perform the chosen operation.
    - Print the result.
4. If the choice is `5`, print a goodbye message and use a mechanism (like setting the loop condition to `False`) to exit the `while` loop.
5. If the choice is invalid (not 1-5), print an error message.
6. Handle potential `ZeroDivisionError` if the user attempts to divide by zero. You can use an `if` statement to check `if num2 == 0:` before performing division.

Example Interaction:

```
--- Simple Calculator ---
1. Add
2. Subtract
3. Multiply
4. Divide
5. Exit
Enter your choice (1-5): 1
Enter first number: 10
Enter second number: 5
Result: 15.0

1. Add
2. Subtract
3. Multiply
4. Divide
5. Exit
Enter your choice (1-5): 4
Enter first number: 20
Enter second number: 0
Error: Cannot divide by zero!

1. Add
2. Subtract
3. Multiply
4. Divide
5. Exit
Enter your choice (1-5): 6
Invalid choice. Please enter a number between 1 and 5.

1. Add
2. Subtract
3. Multiply
4. Divide
5. Exit
Enter your choice (1-5): 5
Thank you for using the calculator! Goodbye!
```