1. Conditional statements: (If, if...else, if...elif...else)

**Conditional statements.**

As you start this lesson, make sure you have a good understanding of boolean variables and expressions since they are building blocks of conditional statements.

Conditional statements help in the decision-making process, for example,

* “Are you hungry? If so, then eat” is a conditional statement

Phrased as: - ***If you’re hungry, then eat***

Let's break this down to pick a boolean expression in our conditional statement.

* “Are you hungry” is the boolean expression here, so the conditional statement is checking if it’s True.
* If “are you hungry” == True, the conditional statement will be executed and you will eat.
* If [you are hungry], then [eat food]



In computer programming, we use the if statement to run a block code only when a certain condition is met.

In Python, there are three forms of the if...else statement.

1. if statement
2. if...else statement
3. if...elif...else statement

**1. Python if Statement**

The basic syntax for the if statement:

# Example of a basic if statement

temperature = 30

if temperature > 25:

print("It's a hot day!")

The if statement evaluates the condition.

1. If condition is evaluated to True, the code inside the body of if is executed.
2. If condition is evaluated to False, the code inside the body of if is skipped.

Practice more on using if conditional statements with:

- Relational operators: (>, >=, <, <=)

- Boolean operators: (and, or, not)

**2. Python if...else Statement**

Syntax

# Example of an if-else statement

temperature = 20

if temperature > 25:

print("It's a hot day!")

else:

print("It's a cool day!")

If the condition evaluates to True,

* the code inside if is executed
* the code inside else is skipped

If the condition evaluates to False,

* the code inside else is executed
* the code inside if is skipped

**3. Python if...elif...else Statement**

The if...else statement is used to execute a block of code among two alternatives.

However, if we need to make a choice between more than two alternatives, then we use the if...elif...else statement.

The syntax of the if...elif...else statement is:

# Example of an if...elif...else statement

temperature = 15

if temperature > 25:

print("It's a hot day!")

elif temperature > 15:

print("It's a warm day!")

else:

print("It's a cold day!")

Here,

1. If the condition evaluates to true, code block 1 is executed.
2. If condition1 evaluates to false, then condition2 is evaluated.
3. If condition 2 is true, code block 2 is executed.
4. If condition 2 is false, code block 3 is executed.

*Test YourSelf with the Question Below :*

Write an if/elif/else statement for a college with a grading system as shown below:

* If grade is 90 or higher, print "A"
* Else if grade is 80 or higher, print "B"
* Else if grade is 70 or higher, print "C"
* Else if grade is 60 or higher, print "D"
* Else, print "F"

**More Resources for if statements**

[**https://www.programiz.com/python-programming/if-elif-else**](https://www.programiz.com/python-programming/if-elif-else)

[**https://www.codecademy.com/resources/docs/python/conditionals**](https://www.codecademy.com/resources/docs/python/conditionals)

2. Loops ( for loops & while loop)

**Python Loops**

In computer programming, loops are used to repeat a block of code. We perform a process of *iteration* (repeating tasks).

There are 2 types of loops in Python:

* [for loop](https://www.programiz.com/python-programming/for-loop)
* [while loop](https://www.programiz.com/python-programming/while-loop)

Programming languages like Python implement two types of iteration:

1. *Indefinite iteration*, where the number of times the loop is executed depends on how many times a condition is met.
2. *Definite iteration*, where the number of times the loop will be executed is defined in advance (usually based on the collection size).

In a for loop, we will know in advance how many times the loop will need to iterate because we will be working on a collection with a predefined length.

With for loops, on each iteration, we will be able to perform an action on each element of the collection.

**For loop syntax:**

# Example of a for loop

fruits = ["apple", "banana", "cherry"]

for fruit in fruits:

print(f"I love {fruit}!")

**Example Explanation:**

1. **List**: fruits is a list containing "apple", "banana", and "cherry".
2. **Loop**: The for loop iterates through each item in the list.
3. **Action**: During each iteration, fruit represents the current item, and print outputs a message for each fruit.

Let’s break down each of these components:

1. A for keyword indicates the start of a for loop.
2. A <temporary variable> is used to represent the value of the element in the collection the loop is currently on.
3. An in keyword separates the temporary variable from the collection used for iteration.
4. A <collection> to loop over. In our examples, we will be using a list.
5. An <action> to do anything on each iteration of the loop.

**For Loops: Using Range**

A [range](https://www.programiz.com/python-programming/methods/built-in/range) is a series of values between two numeric intervals.

We use Python's built-in function range() to define a range of values.

For example,

five\_steps = range(5)

# five\_steps is now a collection with 5 elements:

# 0, 1, 2, 3, 4

# Example of a for loop with range

for number in range(1, 6): # Loops from 1 to 5

print(f"Number: {number}")

**Example Explanation:**

1. **range(1, 6)**: Generates numbers starting from 1 up to (but not including) 6.
2. **Loop**: The for loop iterates over each number in this range.
3. **Action**: During each iteration, number represent the current value, and print outputs it.

**While Loops**

A while loop performs a set of instructions as long as a given condition is true.

While loop structure

# Example of a while loop

count = 1

while count <= 5:

print(f"Count: {count}")

count += 1 # Increment the counter

**Explanation:**

1. **Initialization**: count = 1 sets the starting value.
2. **Condition**: while count <= 5 keeps the loop running as long as count is less than or equal to 5.
3. **Action**: Inside the loop:
   * It prints the current value of count.
   * The count the variable is incremented by 1 (count += 1) to eventually meet the exit condition.

**Loop controls: Break and continue**

The **break** statement is used to terminate the loop immediately when it is encountered.

# Example of loop controls: break and continue

for number in range(1, 10): # Loop through numbers 1 to 9

if number == 5:

print("Breaking the loop at 5")

break # Exit the loop when number is 5

elif number % 2 == 0:

print(f"Skipping {number} because it's even")

continue # Skip the rest of the loop body for even numbers

print(f"Processing number: {number}")

**Explanation:**

1. **break**: Stops the loop entirely when number == 5.
2. **continue**: Skips the current iteration when number is even and moves to the next loop iteration.

**Python continue Statement**

The continue statement is used to skip the current iteration of the loop and the control flow of the program goes to the next iteration.

While the break control statement will come in handy, there are other situations where we don’t want to end the loop entirely. What if we only want to skip the current iteration of the loop?

**Nested Loops**

In Python, loops can be nested inside other loops. Nested loops can be used to access items of lists which are inside other lists. The item selected from the outer loop can be used as the list for the inner loop to iterate over.

**Example code**

# Example of a nested loop

for i in range(1, 4): # Outer loop

for j in range(1, 4): # Inner loop

print(f"Outer loop: {i}, Inner loop: {j}")

**More Resources for loops**

[**https://www.codecademy.com/resources/docs/python/loops**](https://www.codecademy.com/resources/docs/python/loops)

[**https://www.programiz.com/python-programming/if-elif-else**](https://www.programiz.com/python-programming/if-elif-else)

[**https://www.programiz.com/python-programming/for-loop**](https://www.programiz.com/python-programming/for-loop)

[**https://www.programiz.com/python-programming/while-loop**](https://www.programiz.com/python-programming/while-loop)

[**https://www.programiz.com/python-programming/break-continue**](https://www.programiz.com/python-programming/break-continue)

1. List Comprehensions

List comprehensions in Python provide a concise and elegant way to create and manipulate lists. They offer a more readable and expressive alternative to traditional loops for generating lists.

**What is a List Comprehension?**

A list comprehension is a compact syntax for creating a list by evaluating an expression for each element in an iterable (like a list or range), optionally filtering elements based on a condition.

**Syntax:**

[expression for item in iterable if condition]

* **expression**: The value or transformation applied to each element.
* **item**: A variable that represents each element in the iterable.
* **iterable**: A sequence (like a list, tuple, or range) to iterate over.
* **condition**: (Optional) A filter that determines whether to include the element.

**Basic Example**

# Traditional loop

squares = []

for x in range(5):

squares.append(x\*\*2)

# List comprehension

squares = [x\*\*2 for x in range(5)]

print(squares) # Output: [0, 1, 4, 9, 16]

Here, is the list of comprehension [x\*\*2 for x in range(5)] creates a list of squares of numbers from 0 to 4.

**Using Conditions in List Comprehensions**

You can include a condition to filter items.

# List comprehension with a condition

even\_numbers = [x for x in range(10) if x % 2 == 0]

print(even\_numbers) # Output: [0, 2, 4, 6, 8]

**Nested List Comprehensions**

List comprehensions can be nested to handle complex operations, such as creating a matrix.

# Create a 3x3 matrix using nested list comprehensions

matrix = [[i \* j for j in range(1, 4)] for i in range(1, 4)]

print(matrix) # Output: [[1, 2, 3], [2, 4, 6], [3, 6, 9]]

**Examples of List Comprehensions**

**1. Transforming Data:**

names = ["Alice", "Bob", "Charlie"]

uppercased\_names = [name.upper() for name in names]

print(uppercased\_names) # Output: ['ALICE', 'BOB', 'CHARLIE']

**2. Filtering Data:**

numbers = [10, 15, 20, 25, 30]

divisible\_by\_5 = [num for num in numbers if num % 5 == 0]

print(divisible\_by\_5) # Output: [10, 15, 20, 25, 30]

**3. Flattening a List:**

nested\_list = [[1, 2], [3, 4], [5, 6]]

flat\_list = [item for sublist in nested\_list for item in sublist]

print(flat\_list) # Output: [1, 2, 3, 4, 5, 6]

**Advantages of List Comprehensions**

1. **Conciseness**: More compact than traditional loops.
2. **Readability**: Easier to understand when used appropriately.
3. **Performance**: Faster than loops in many cases due to optimization.

**When Not to Use List Comprehensions**

While list comprehensions are powerful, they can become hard to read if they are too complex. In such cases, traditional loops or generator functions may be more appropriate.

# Complex list comprehension (less readable)

result = [x \* y for x in range(10) for y in range(5) if x + y > 5]

# Better as a loop (more readable)

result = []

for x in range(10):

for y in range(5):

if x + y > 5:

result.append(x \* y)

**Conclusion**

List comprehensions are a versatile and efficient tool in Python, enabling you to write cleaner and faster code. They are ideal for transforming, filtering, or generating lists in a Pythonic way. Start practicing them to enhance your coding skills!

1. Python Functions

Functions are a fundamental part of Python programming. They allow you to encapsulate reusable blocks of code to make programs modular, easier to understand, and maintain.

**What is a Function?**

A function is a block of code designed to perform a specific task. You define a function once and can use it multiple times throughout your code.

**Syntax:**

def function\_name(parameters):

"""Optional docstring explaining the function."""

# Code block

return value # Optional return statement

**Types of Functions**

1. **Built-in Functions**: Predefined functions in Python (e.g., print(), len(), type()).
2. **User-defined Functions**: Functions created by the user.

**Defining and Calling a Function**

Here’s how you define and call a function in Python:

# Function definition

def greet(name):

"""Greet a person by their name."""

return f"Hello, {name}!"

# Function call

print(greet("Alice")) # Output: Hello, Alice!

**Key Components of a Function**

1. **Function Name**: Should be descriptive and follow naming conventions.
2. **Parameters**: Variables passed into the function.
3. **Docstring**: An optional description of what the function does.
4. **Return Statement**: Outputs a value from the function (optional).

**Parameters and Arguments**

Functions can accept zero or more arguments.

**1. Positional Arguments**:

def add(a, b):

return a + b

print(add(3, 5)) # Output: 8

**2. Default Arguments**:

def greet(name="Guest"):

return f"Hello, {name}!"

print(greet()) # Output: Hello, Guest!

print(greet("Alice")) # Output: Hello, Alice!

**3. Keyword Arguments**:

def introduce(name, age):

return f"My name is {name}, and I'm {age} years old."

print(introduce(age=25, name="Bob")) # Output: My name is Bob, and I'm 25 years old.

**Returning Values**

A function can return values using the return statement.

def square(number):

return number \*\* 2

result = square(4)

print(result) # Output: 16

**Anonymous Functions: Lambda**

Python supports anonymous functions using the lambda keyword. They are useful for short, simple functions.

# Lambda function for adding two numbers

add = lambda x, y: x + y

print(add(3, 5)) # Output: 8

# Using lambda with map()

numbers = [1, 2, 3, 4]

squares = list(map(lambda x: x\*\*2, numbers))

print(squares) # Output: [1, 4, 9, 16]

**Recursive Functions**

A function can call itself, enabling recursive problem-solving.

def factorial(n):

if n == 1:

return 1

return n \* factorial(n - 1)

print(factorial(5)) # Output: 120

**Benefits of Functions**

1. **Reusability**: Write once, and use multiple times.
2. **Modularity**: Break programs into smaller, manageable parts.
3. **Improved Readability**: Descriptive names make code easier to understand.

**Conclusion**

Functions are a powerful tool that makes your code cleaner, more efficient, and easier to maintain. Mastering their use is essential for any Python developer. Start experimenting with different types of functions to understand their versatility!

1. Weekly Code Challenge

**Coding Challenges for basic control flows and functions**

**1. Large Power**

Create a method that tests whether the result of taking the power of one number to another number provides an answer which is greater than 5000. We will use a conditional statement to return **True** if the result is greater than 5000 or return **False** if it is not. In order to accomplish this, we will need the following steps:

1. Define the function to accept two input parameters called **base** and **exponent**
2. Calculate the result of **base** to the power of **exponent**
3. Use an **if** statement to test if the result is greater than 5000. If it is then return **True**. Otherwise, return **False**

**Coding Question**

Create a function named **large\_power()** that takes two parameters named **base** and **exponent**.

If **base** raised to the **exponent** is greater than **5000**, return **True**, otherwise return **False**

**2.Divisible By Ten**

Create a function that determines whether or not a number is divisible by ten. A number is divisible by ten if the remainder of the number divided by 10 is 0. Using this, we can complete this function in a few steps:

1. Define the function header to accept one input **num**
2. Calculate the remainder of the input divided by 10 (use modulus)
3. Use an **if** statement to check if the remainder was 0. If the remainder was 0, return **True**, otherwise, return **False**

**Coding question**

Create a function called **divisible\_by\_ten()** that has one parameter named **num**.

The function should return **True** if **num** is divisible by **10**, and **False** otherwise. Consider using modulo operator **%** to check for divisibility.