Python Refresher for AI

Data Structures, Conditional Statements, and Loops

Quick Recap

- Programming involves writing instructions that computers execute.
- Programming languages include procedural, object-oriented, functional, and scripting types.
- Python provides a simple, readable syntax for efficient coding.
- Python supports data science, AI, and ML with extensive libraries.
- AI-powered tools like GitHub Copilot enhance coding but raise ethical concerns.

Engage and Think

As a junior software developer, you need to optimize a program that processes large amounts of data efficiently. Your goal is to store, retrieve, and manipulate data effectively while ensuring code readability and performance.

To achieve this, you must choose the right data structures for different scenarios and implement control flow and loops to automate operations. You also need to use functions to modularize your code and avoid redundancy.

How do you decide which data structure to use for a specific task?

Learning Objectives

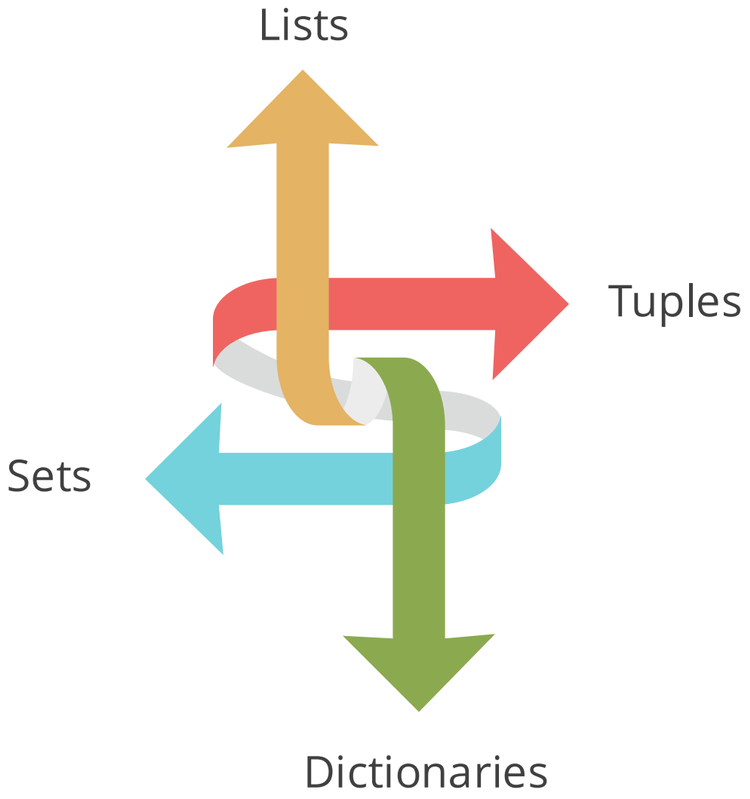
By the end of this lesson, you will be able to:

- Make use of lists, tuples, dictionaries, and sets to store, modify, and access data efficiently
- Apply control flow statements to execute decisions and manage program logic dynamically
- Utilize loops to automate repetitive tasks and iterate through data structures effectively
- Define and use functions to modularize code, improve reusability, and enhance readability
- Select the right data structure for different programming scenarios to optimize data handling

Data Structures

Data Structures: Introduction

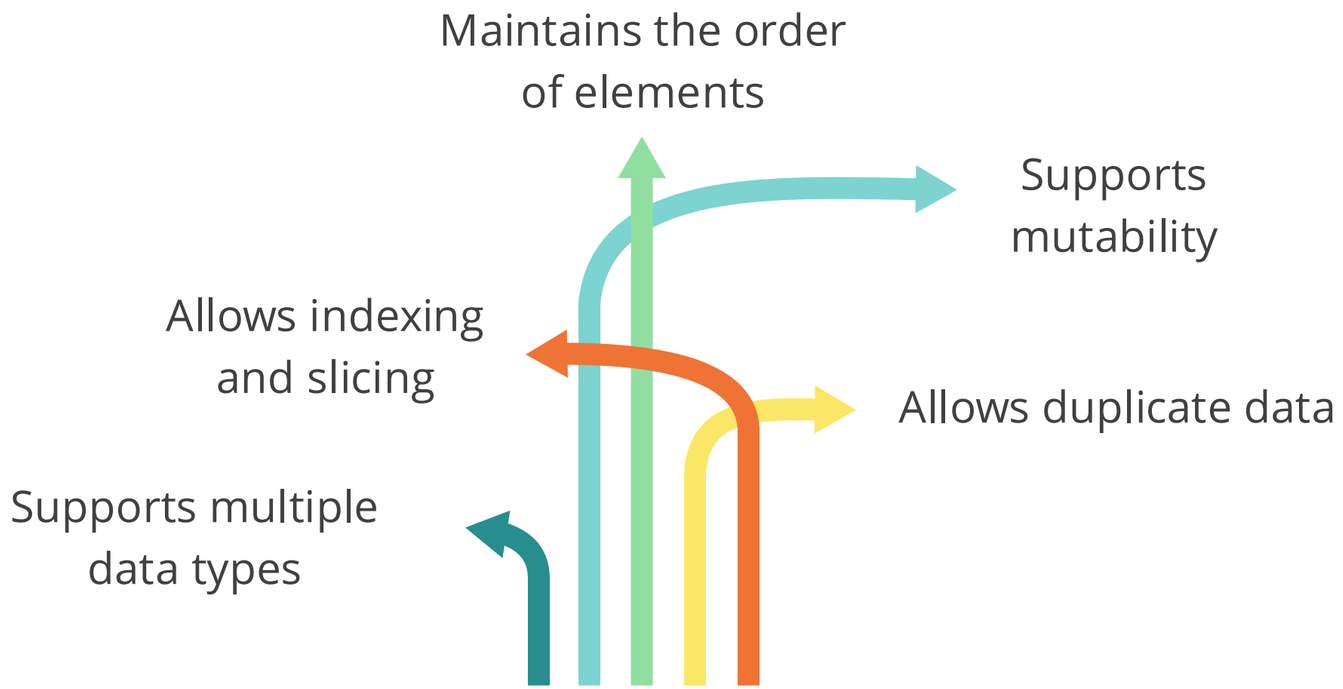
A data structure is a way of organizing, storing, and managing data in memory to enable efficient access and modifications. The types of data structures in Python are as:



Lists

Lists: Introduction

A list is a versatile, ordered, and mutable data structure that can hold elements of different data types. The following are its key characteristics:



Note

A list is defined using square brackets [ ], with elements separated by commas.

Creating Lists

The following is an example of how to create a list:

Syntax

```
my_list = [ value1,
value2, value3, .....]
```

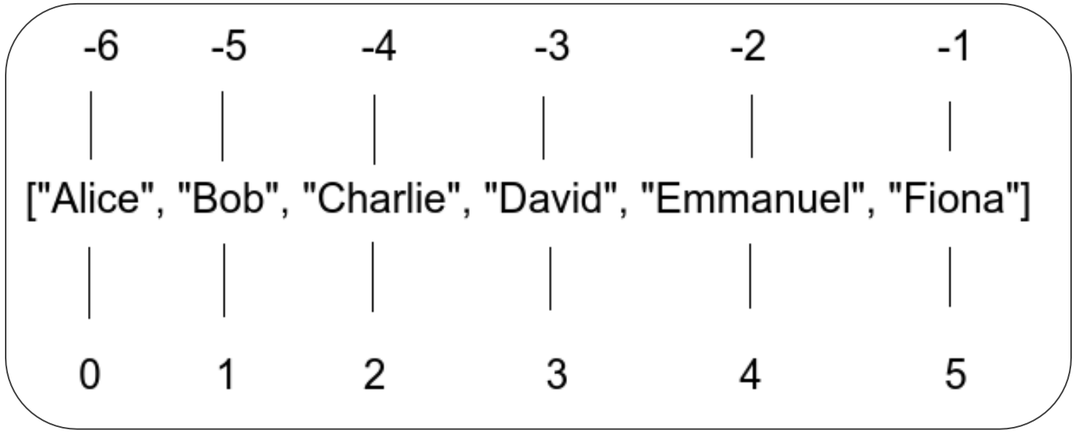
```
#Creating a list with multiple data types
mixed_list = [25, "hello", 3.14, True]
print(mixed_list)
```

Note

Use lists when you need an ordered collection of items that may change frequently.

List Indexing: Introduction

Indexing refers to accessing elements of a list using their position (index).



Note

Each item in a list is assigned an index number, which allows you to retrieve or modify elements efficiently.

List Indexing: Example

The following is an example of fetching an element from a list using positive indexing:

```
#Fetching list items from the fruits list using positive
index
fruits = ["apple", "banana", "cherry", "mango"]
print(fruits[0])  # Output: apple
print(fruits[2])  # Output: cherry
```

Note

Positive indexing happens from left to right, where the index starts at 0 for the first element.

List Indexing: Example

The following is an example of fetching an element from a list using negative indexing:

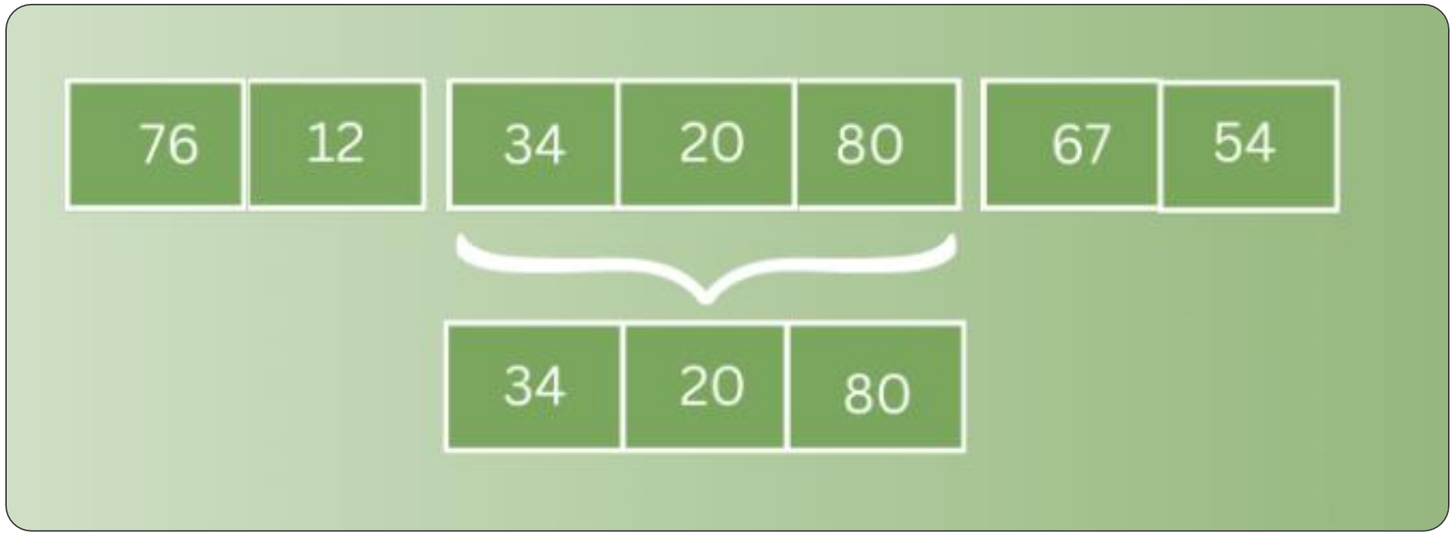
```
#Fetching list items from the fruits list using negative index
fruits = ["apple", "banana", "cherry", "mango"]
print(fruits[-1])  # Output: mango
print(fruits[-3])  # Output: banana
```

Note

Negative indexing happens from right to left, where the index starts at -1 for the last element.

List Slicing: Introduction

Slicing refers to extracting a subset of elements from a list using a specific range of indices.



Note

List slicing follows the syntax: **list[start_index:stop_index:step]**.

List Slicing: Example

The following is an example of extracting elements from indexes 1 to 3:

```
#Performing slicing on the list numbers
numbers = [10, 20, 30, 40, 50, 60]
print(numbers[1:4])  # Output: [20, 30, 40]
```

List Methods

The following is an example of how the **append()** method works:

```
fruits = ["apple", "banana"]
fruits.append("cherry")  # Adds 'cherry' at the end
print(fruits)  # Output: ['apple', 'banana', 'cherry']
```

Note

The append() method inserts a single element at the end of the list.

List Methods

The following is an example of how the **pop()** method works:

```
numbers = [10, 20, 30, 40]
removed_element = numbers.pop(2)  # Removes element at index 2
print(numbers)  # Output: [10, 20, 40]
print(removed_element)  # Output: 30
```

Note

The pop() method removes and returns an element from a specified index in a list. If no index is provided, the pop() method removes and returns the last element by default.

List Methods

The following is an example of how the **extend()** method works:

```
colors = ["red", "blue"]
colors.extend(["green", "yellow"])  # Adds multiple elements
print(colors)  # Output: ['red', 'blue', 'green', 'yellow']
```

Note

The extend() method merges another iterable (list, tuple, or set) into the list.

Quick Check

Which of the following methods adds a single element at the end of the list?

A. append()

B. extend()

C. pop()

D. None of the above

Demo: Working with Lists in Python

Duration: 10 minutes

DEMONSTRATION

Overview:

This demo covers creating, modifying, and accessing Python lists using indexing, slicing, and built-in methods. You will learn how to add, remove, and update elements to manipulate lists effectively.

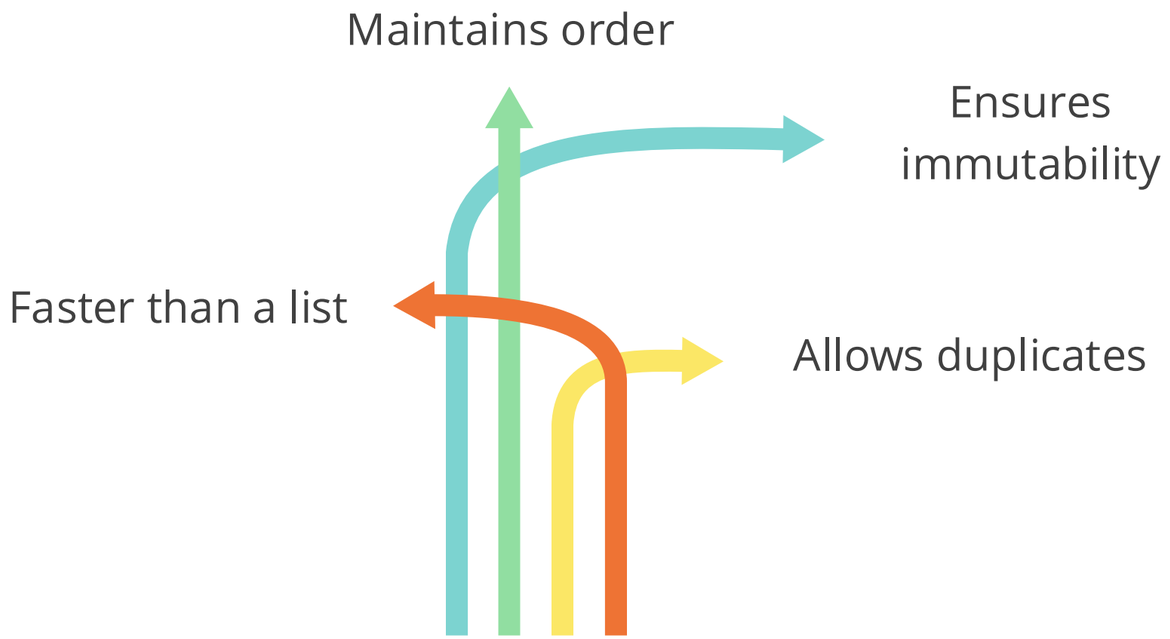
Note:

Please download the demo document from the reference material section for step-by-step guidance.

Tuples

Tuples: Introduction

A tuple is an ordered, immutable collection of elements.
The following are its key characteristics:



Note

Tuples are defined using parentheses () and can store multiple data types.

Creating Tuples

The following is an example of how to create tuples:

Syntax

my_tuple = (value1, value2, value3, ...)

```
# Creating a tuple with different data types
fruits = ("apple", "banana", "cherry")
numbers = (1, 2, 3, 4)
mixed = (10, "hello", True, 3.14)

# Creating a tuple using the tuple() constructor
tuple_from_list = tuple([5, 6, 7])
print(fruits)  # Output: ('apple', 'banana',
'cherry')
print(tuple_from_list)  # Output: (5, 6, 7)
```

Note

Use tuples when you need a read-only collection that should not change.

Accessing Tuple Items: Indexing

Since tuples are ordered, elements can be accessed using indexing and slicing, just like lists. The following is an example of accessing tuple items using their indexes:

```
animals = ("cat", "dog", "elephant", "lion")

#Accessing elements using positive indexing
print(animals[0])  # Output: cat
print(animals[2])  # Output: elephant

#Accessing elements using negative indexing
print(animals[-1])  # Output: lion (last element)
print(animals[-3])  # Output: dog
```

Accessing Tuple Items: Slicing

The following is an example of accessing tuple items using slicing:

```
animals = ("cat", "dog", "elephant", "lion")

#Accessing elements using slicing
print(animals[1:3])  # Output: ('dog', 'elephant')
print(animals[:2])  # Output: ('cat', 'dog’)

#Selects every second element
print(animals[::2])  # Output: ('cat', 'elephant’)
```

Tuple Packing

It means assigning multiple values to a single tuple.
The following is an example of tuple packing:

```
#Tuple packing
person = ("John", 25, "Developer")
print(person)  # Output: ('John', 25, 'Developer')
```

Tuple Unpacking

It allows values to be extracted in separate variables.
The following is an example of tuple unpacking:

```
#Tuple unpacking
person = ("John", 25, "Developer")
name, age, job = person
print(name)  # Output: John
print(age)   # Output: 25
print(job)   # Output: Developer
```

Quick Check

Which of the following statements about tuples is true?

A. Tuples are mutable.

B. Tuples support item reassignment.

C. Tuples are faster than lists for iteration.

D. Tuples can contain only integers.

Demo: Working with Tuples in Python

Duration: 10 minutes

Overview:

This demo covers creating, accessing, and working with tuples in Python. You will learn how to retrieve elements, explore their immutability, and understand their performance benefits.

Note:

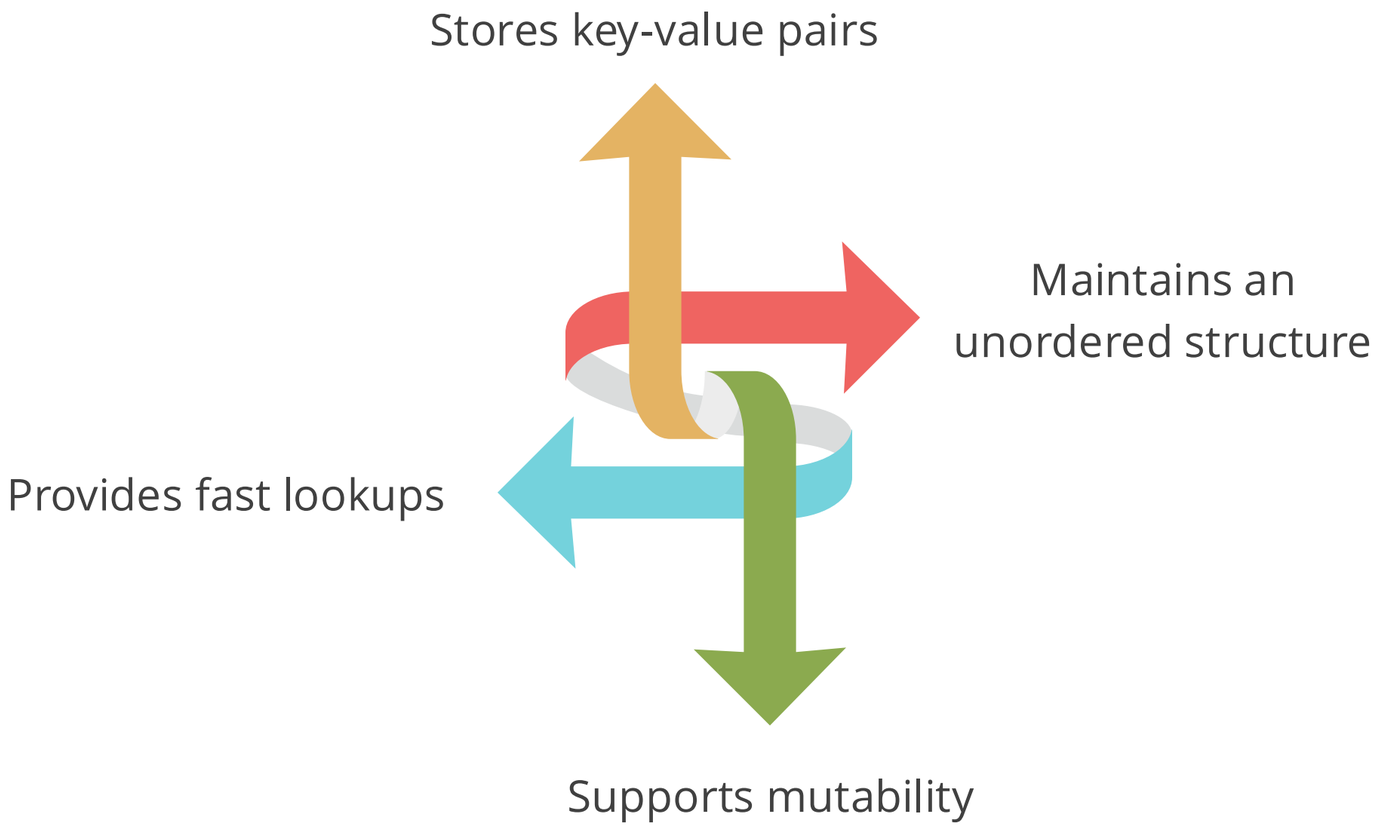
Please download the demo document from the reference material section for step-by-step guidance.

DEMONSTRATION

Dictionaries

Dictionaries: Introduction

A dictionary is a mutable, unordered data structure that stores key-value pairs, allowing efficient data retrieval and modification. The following are its key characteristics:



Creating Dictionaries

The following is an example of how to create dictionaries:

Syntax

```
my_dict = {key1:value1,
key2:value2,
key3:value3, .....}
```

```
# Creating a dictionary with key-value pairs
student = {"name": "Alice", "age": 21, "course": "Computer
Science"}

# Using the dict() constructor
employee = dict(id=101, name="John", department="HR")

print(student)
# Output: {'name': 'Alice', 'age': 21, 'course': 'Computer
Science'}

print(employee)
# Output: {'id': 101, 'name': 'John', 'department': 'HR'}
```

Note

Use dictionaries when you need to store data in key-value pairs.

Accessing Values

The following is an example of accessing values using keys:

```
# Creating a dictionary with key-value pairs
student = {"name": "Alice", "age": 21, "course": "Computer Science"}

#accessing element using key
print(student["name"])  # Output: Alice

#accessing element using get method
print(student.get("age"))  # Output: 21

print(student.get("grade", "Not Found"))  # Output: Not Found
```

Modifying Values

The following is an example of modifying values using keys:

```
# Creating a dictionary with key-value pairs
student = {"name": "Alice", "age": 21, "course": "Computer Science"}

# Modify and add elements
student["age"] = 22  # Updating a value
student["city"] = "New York"  # Adding a new key-value pair
print(student)
```

Dictionary Methods

The following is an example of the different methods in a dictionary:

```
# Creating a dictionary with key-value pairs
student = {"name": "Alice", "age": 21, "course": "Computer Science","city":"Newyork"}

#Returns a view object containing all keys in the dictionary
print(student.keys())
#Returns a view object containing all values in the dictionary
print(student.values())
#Returns a view object containing all key-value pairs as tuples
print(student.items())


student = {"name": "Alice", "age": 22, "course": "Computer Science"}
# Removes and returns the value for the given key
removed_value = student.pop("age")
print(student)

student = {"name": "Alice", "age": 22, "course": "Computer Science"}
#Removes all key-value pairs, making the dictionary empty
student.clear()
print(student)
```

Nested Dictionaries

Dictionaries can contain other dictionaries, allowing hierarchical data structures. The following is an example of creating a nested dictionary:

```
# Nested dictionary
company = {
    "CEO": {"name": "Alice", "age": 45},
    "Manager": {"name": "Bob", "age": 38},
    "Intern": {"name": "Charlie", "age": 22}
}

# Accessing nested dictionary values
print(company["CEO"]["name"])  # Output: Alice
print(company["Manager"]["age"])  # Output: 38
```

Quick Check

Which method is used to get the value of a key in a dictionary safely without raising an error?

A. get()

B. items()

C. keys()

D. values()

Demo: Working with Dictionaries in Python

Duration: 10 minutes

Overview:

This demo covers creating, updating, and retrieving data using Python dictionaries. You will learn how to store key-value pairs, modify entries, and efficiently access data.

Note:

Please download the demo document from the reference material section for step-by-step guidance.

DEMONSTRATION

Sets

Sets: Introduction

A set is a collection of unique elements. The following are key characteristics of a set:

Remains unordered

Ensures the uniqueness of elements

Provides fast lookups

Supports mutability

Creating Sets

The following is an example of how to create sets:

Syntax

my_set = {value1, value2, value3, ...}

```
# Creating a set with unique elements
fruits = {"apple", "banana", "cherry", "apple"}
print(fruits)
# Output: {'apple', 'banana', 'cherry'}

# Creating an empty set
empty_set = set()  # {} creates an empty dictionary, so
use set()
print(type(empty_set))  # Output: <class 'set'>
```

Note

Use sets when you need a collection of unique items.

Set Operations

The following is an example of the **union()** method in sets:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

union_set = set1.union(set2)
print(union_set)
# Output: {1, 2, 3, 4, 5}
```

Note

union() combines all unique elements from two or more sets.

Set Operations

The following is an example of the **intersection()** method in sets:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

common_set = set1.intersection(set2)
print(common_set)
# Output: {3}
```

Note

intersection() retrieves only the common elements between sets.

Set Operations

The following is an example of the **difference()** method in sets:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

diff_set = set1.difference(set2)
print(diff_set)
# Output: {1, 2}
```

Note

difference() returns elements that exist in one set but not in another.

Quick Check

Which of the following statements correctly finds the difference between two sets in Python?

A. set1.difference(set2)

B. set1 & set2

C. set1 | set2

D. set1.intersection(set2)

Demo: Working with Sets in Python

Duration: 10 minutes

Overview:

This demo covers working with sets in Python, including adding, removing, and performing mathematical set operations like union, intersection, and difference. You will learn how to manipulate sets efficiently.

Note:

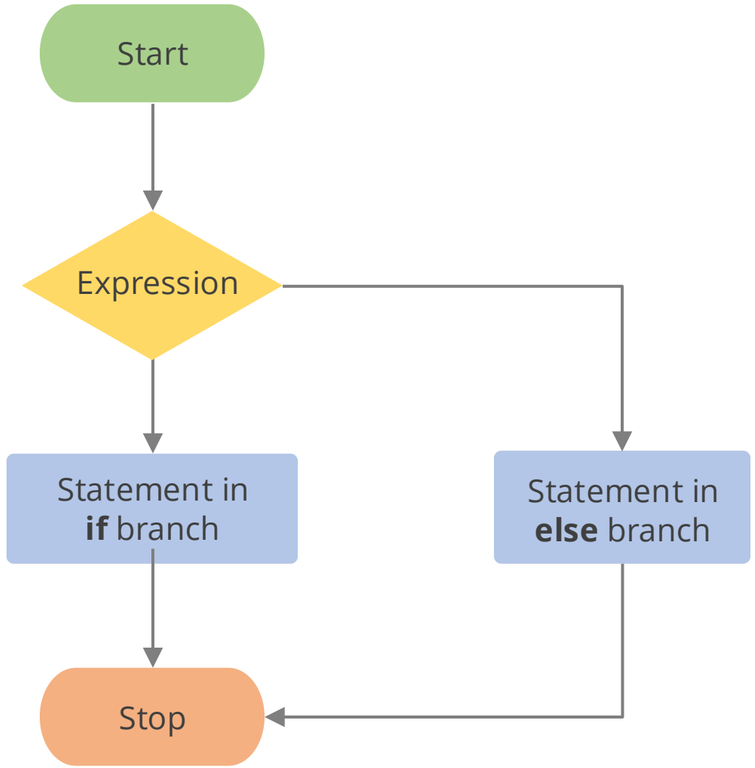
Please download the demo document from the reference material section for step-by-step guidance.

DEMONSTRATION

Conditional Statements

Conditional Statements: Introduction

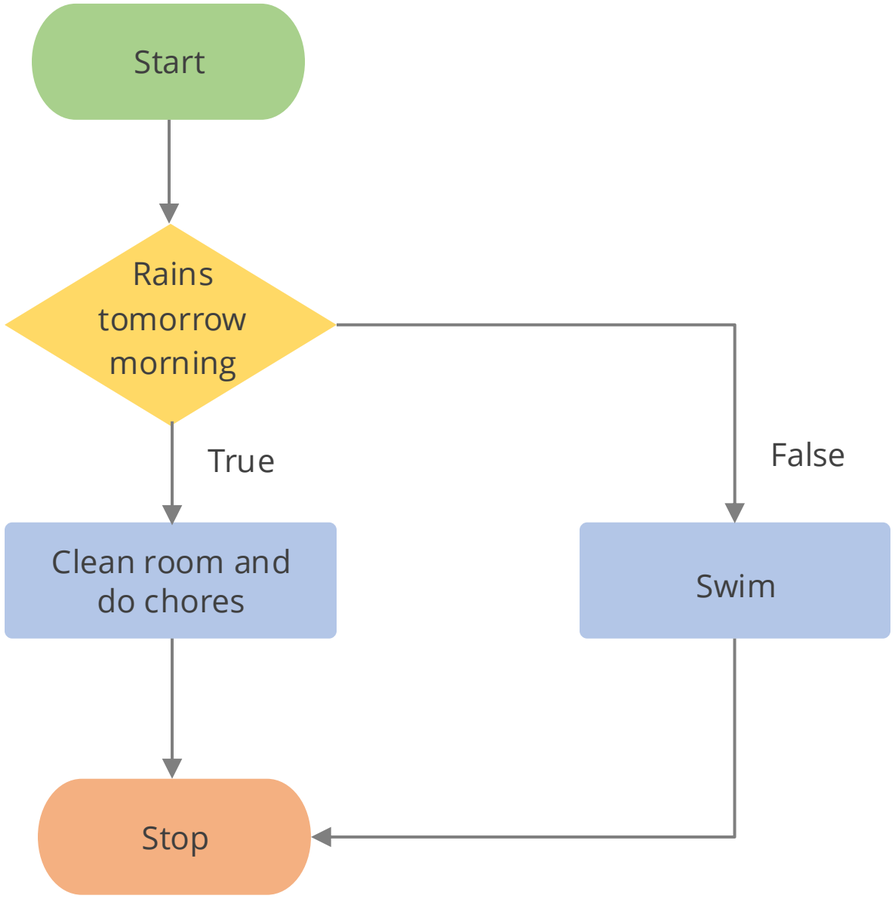
They execute a block of code based on whether a condition is True or False, controlling program flow and decision-making.



Conditional statements, also known as decision control structures, evaluate variables or expressions that return either True or False.

Conditional Statements: Scenario

- Based on the weather, I can choose my activities for the day.
- **If** it rains tomorrow morning, I will clean my room and do chores.
- **Else**, I will go swimming.



Decision Control Structures

Python offers four types of decision control structures. They are:

- If statement
- If-else statement
- Nested-if statement
- If-elif-else statement

If Statement

Python uses the if statement to change the program's control flow.
The indentation marks the block of code.

Syntax

```
if condition:
    statement
    statement
    .......
```

- A colon indicates the beginning of the block.
- The block is usually indented by four spaces.
- Each statement within the block must have the same indentation.

If Statement: Example

```
inp = input("Nationality ? ")

if inp == "French":
    print("Préférez vous parler francais?")
```

If-Else Statement

This evaluates the condition and executes the *if* block only when the test condition is True. Otherwise, it executes the *else* block.

Syntax

```
if condition:
    statement 1
    statement 2
else:
    statement 3
    statement 4
```

The else statement is optional in the if-else construct.

If-Else Statement: Example

```
num = int(input('Enter a number : '))

if num > 0:
    print(num, 'is a positive number.')
else:
    print(num, 'is a negative number.')
```

If-Elif-Else Statement

This allows checking multiple conditions. If the *if* condition is False, it checks the next *elif* condition, and so on.

Syntax

```
if condition 1:
    statement
elif condition 2:
    statement
elif condition 3:
    statement
else:
    statement
```

Only one block among the if-elif-else blocks is executed.
If all conditions are False, the else block is executed.

If-Elif-Else Statement: Example

```
marks = int(input('Enter Marks : '))

if marks >= 90:
    print('Grade A')
elif marks >= 70:
    print('Grade B')
elif marks >= 55:
    print('Grade C')
elif marks >= 35:
    print('Grade D')
else:
    print('Grade F')
```

Nested-If

Python allows an *if* statement inside another *if* statement.

Syntax

```
if (condition 1):
    statement
    # Executes when condition 1 is True
    if (condition 2):
        # Executes when condition 2 is also True
    # inner if Block ends here
# outer if Block ends here
```

- This format is called nesting.
- Indentation defines the level of nesting.

Nested-If: Example

```
num = 15

if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative number")
```

True or False

Python evaluates the following objects as False:

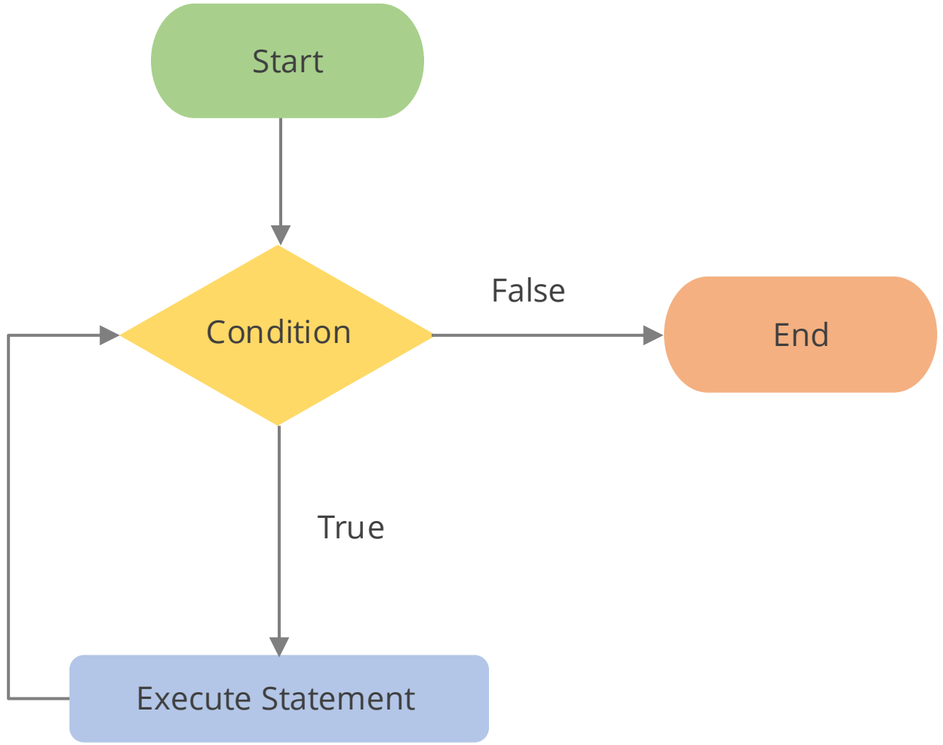
- Numerical zero values
- Boolean value False
- Empty strings
- Empty list, tuples, and dictionaries
- None

All other values are considered True in Python.

Loops

Loops: Introduction

A loop statement allows the repeated execution of a statement or group of statements.



Loops: Types

for loop

The *for* loop iterates over a sequence list, tuple, string, or other objects.
Its syntax is:

```
for a in iteration_object:
 Body
 of
 loop
```

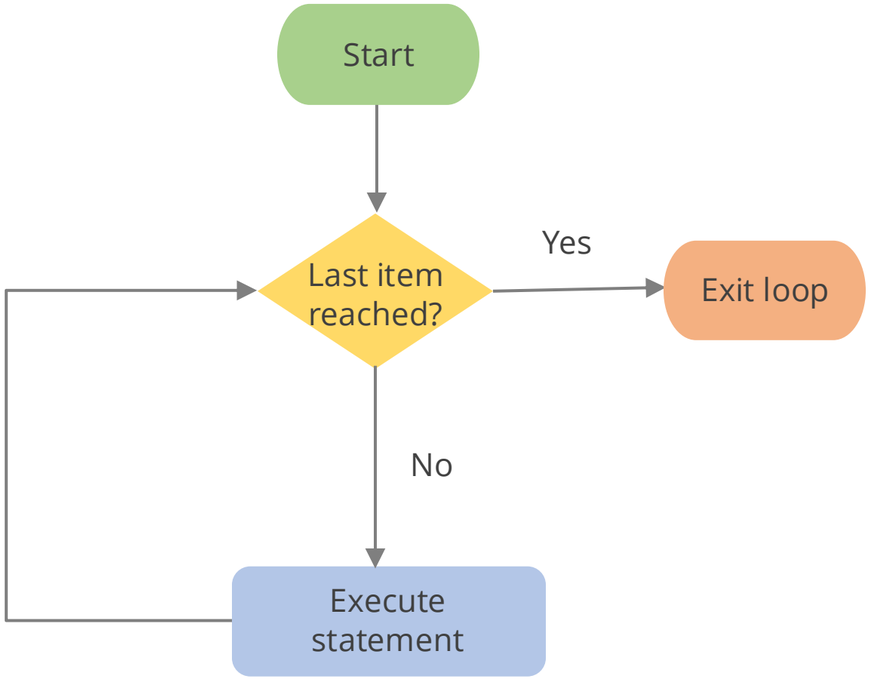
while loop

The *while* loop iterates over a block of code if the test expression is true.
Its syntax is:

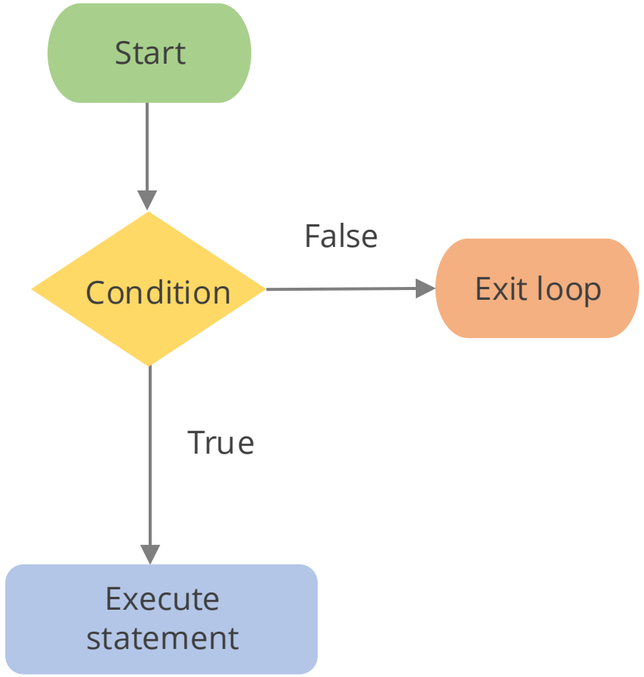
```
while test_expression:
 Body
 of
 loop
```

Loops: Types

Python supports **for** and **while** loops.



For loop



While loop

Loops in Python: Example

for loop

```
string = 'Python'
for s in string:
    print(s)
```

while loop

```
counter = 0
while counter < 5:
print(counter)
counter += 1
```

Nested Loop

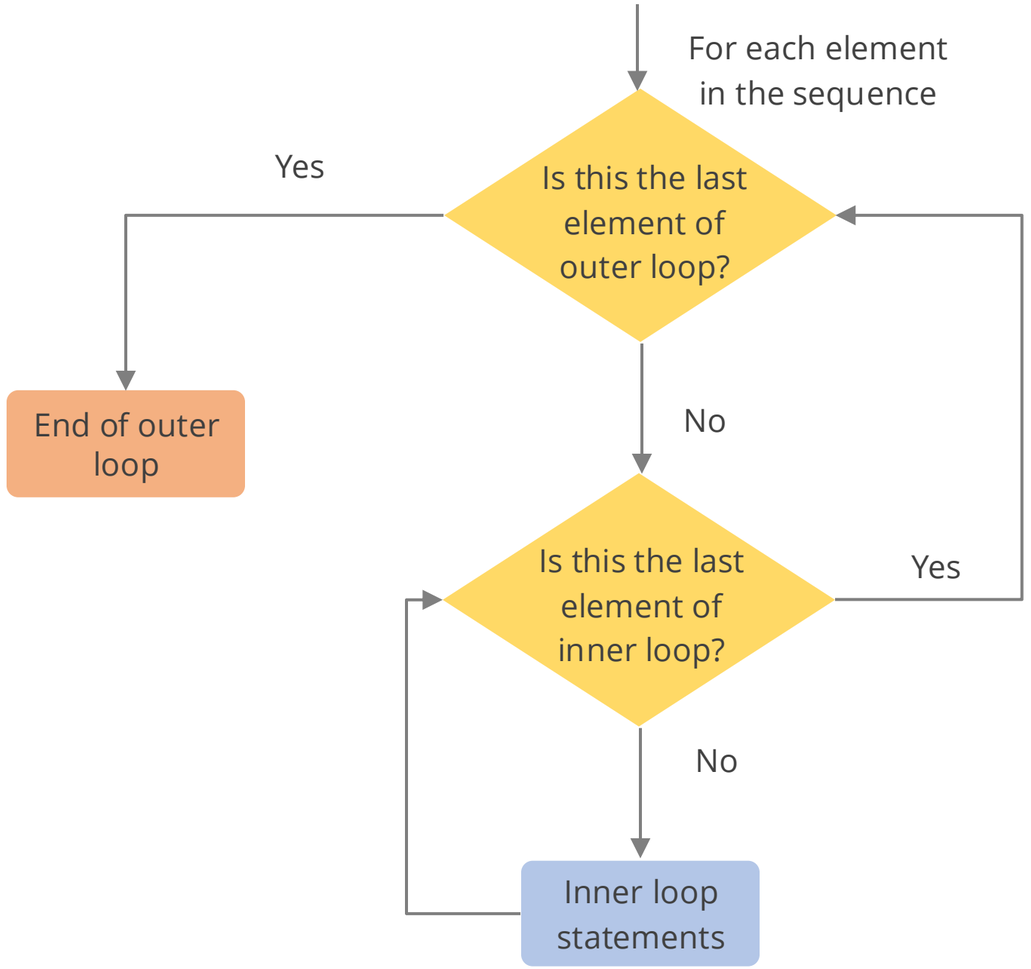
It is a loop inside the body of the outer loop.

Syntax

```
# outer loop
for element in sequence :
    outer loop statements
    # inner loop
    for element in sequence :
        body of inner loop
    additional outer loop statements
```

The inner and outer loops can be different or of the same type.

Nested Loop: Flowchart



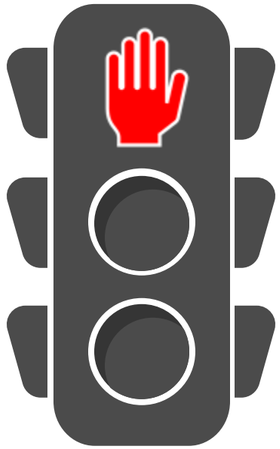
Nested Loop: Example

Here is the code to print multiplication tables (from 2 to 10)

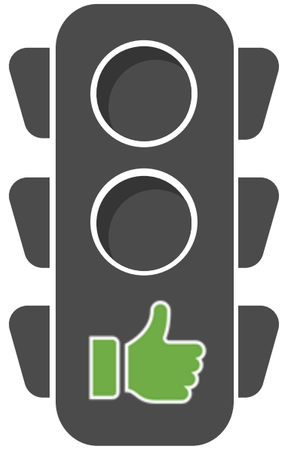
```
# Outer loop
#loop to iterate from 2 to 10
for i in range(2, 11):
    # Nested loop to iterate from 1 to 10
    for j in range(1, 11):
        # Print multiplication using f-string
        print(f"{i:2d} × {j:2d} = {i * j:2d}")
    print(f" End of multiplication table of {i}\n")
```

Loops Control Statements

They alter the flow of execution in loops. Python supports two such statements:



Break



Continue

Loop Control Statements: Break

Syntax

```
break
```

- The **break** statement exits the innermost enclosing of the **for** or **while** loop.
- It terminates the nearest enclosing loop and skips the optional **else** block.
- If a loop is terminated by a **break**, the loop variable retains its current value.

Break: Example

```
# Use of break statement inside the loop
for i in "Hello string":
    if i == "l":
        break
    print(i)

print("End of Loop")
```

Loop Control Statements: Continue

Syntax

```
continue
```

- The **continue** statement skips the current iteration and proceeds with the next one.
- It does not terminate the loop but moves control to the next iteration.
- As a result, the optional **else** block of the loop still executes.

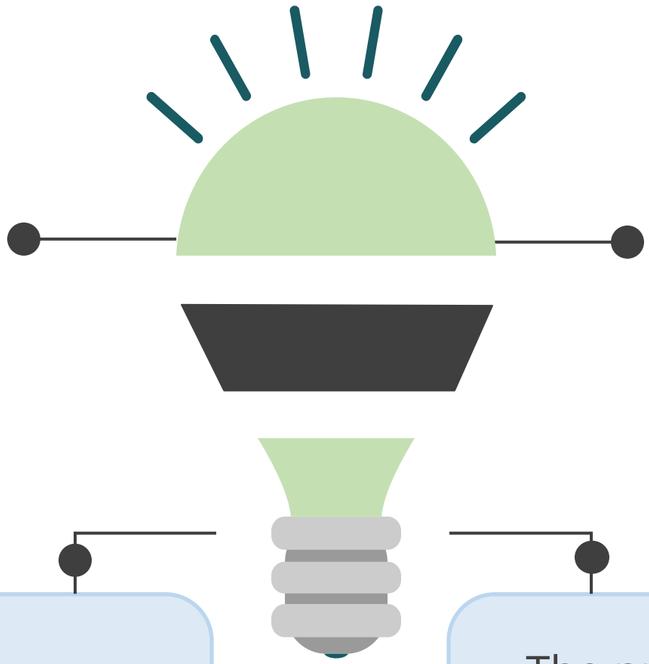
Continue: Example

```
# Use of continue statement inside the loop
for i in "Hello string":
    if i == "l":
        continue
    print(i)

print("End of Loop")
```

Loop Else Statement

Python allows the **else** keyword to be used with the **for** and **while** loops.



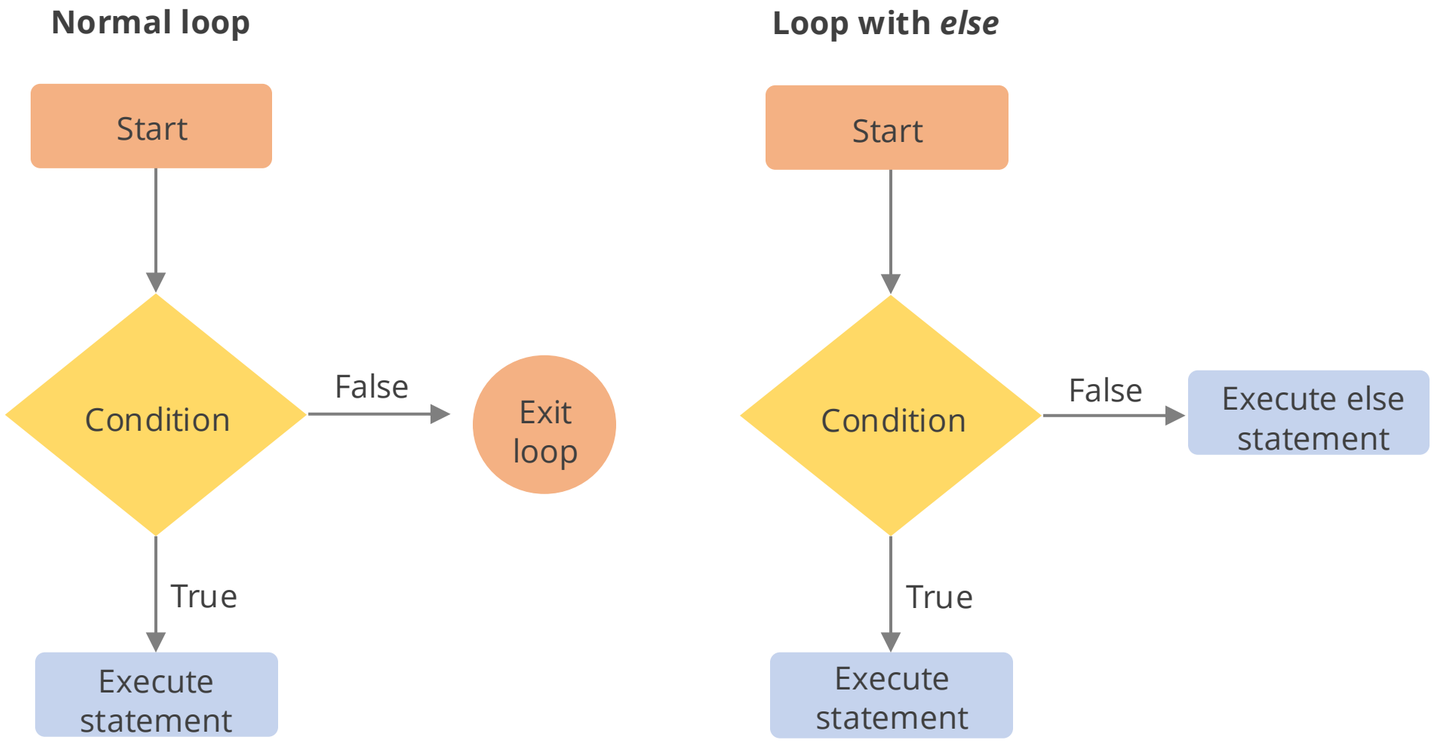
The statements of the **else** block are executed after all the iterations are completed.

The **else** clause is defined after the body of the loop.

The program exits the loop only after the **else** block is executed.

Loop Else Statement

The loop else statement does not execute if the loop terminates due to a **break** statement.



For Else Statement: Example

```
numbers = [1, 2, 3, 4, 5, 6, 7]
for num in numbers:
    if num == 6:
        print("Number found!")
        break
else:
    print("Number not found!")
```

While Else Statement: Example

```
count = 0
while count < 5:
    print("Count:", count)
    if count == 3:
        print("Count reached 3!")
        break
    count += 1
else:
    print("Loop completed!")
```

Quick Check

Which of the following statements about loops in Python is true?

A. A for loop runs indefinitely unless a break statement is used.

B. A while loop executes only once, regardless of the condition.

C. The continue statement skips the rest of the current iteration and moves to the next iteration.

D. A for loop cannot iterate over strings.

Demo: Understanding If-Elif Conditions and Loops in Python

Duration: 10 minutes

Overview:

This demo covers if-elif conditions and loops in Python. You will learn how to implement decision-making, control looping structures, and handle different cases efficiently.

Note:

Please download the demo document from the reference material section for step-by-step guidance.

DEMONSTRATION

Comprehensions

List Comprehension: Introduction

It is a concise way to create lists in Python.

Syntax

```
new_list = [expression for item in iterable if condition]
```

output collection condition

List Comprehension: Example

The following is an example of creating a list comprehension:

```
squares = [x**2 for x in range(1, 6)]
print(squares)
# Output: [1, 4, 9, 16, 25]


numbers = [1, 2, 3, 4, 5, 6, 7, 8]
evens = [x for x in numbers if x % 2 == 0]
print(evens)
# Output: [2, 4, 6, 8]
```

Dictionary Comprehension: Introduction

It allows you to create dictionaries dynamically with a concise syntax.

Syntax

```
new_dict = {key_expression: value_expression for item in iterable if condition}
```

Dictionary Comprehension: Example

The following is an example of creating a dictionary comprehension:

```
#Creating a dictionary with numbers and their squares
squares_dict = {x: x**2 for x in range(1, 6)}
print(squares_dict)
# Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

#Filtering a dictionary to keep only even values
num_dict = {x: x**2 for x in range(10) if x % 2 == 0}
print(num_dict)
# Output: {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}
```

Set Comprehension: Introduction

They allow you to generate unique, unordered collections dynamically.

Syntax

```
new_set = {expression for item in iterable if condition}
```

Set Comprehension: Example

The following is an example of creating a set comprehension:

```
#Creating a set of squares
squares_set = {x**2 for x in range(1, 6)}
print(squares_set)
# Output: {1, 4, 9, 16, 25}


#Removing duplicates from a list using set comprehension
numbers = [1, 2, 2, 3, 4, 4, 5, 5, 6]
unique_numbers = {x for x in numbers}
print(unique_numbers)
# Output: {1, 2, 3, 4, 5, 6}
```

Quick Check

Which of the following statements about comprehensions in Python is true?

A. List comprehension can only be used with numeric data types.

B. Dictionary comprehension allows for the creation of a dictionary from an existing iterable.

C. Set comprehension allows duplicate values in the final set.

D. Dictionary comprehension does not support conditions.

Demo: Understanding Comprehensions in Python

Duration: 10 minutes

Overview:

This demo covers list, dictionary, and set comprehensions in Python. You will learn how to transform data efficiently, apply filtering, and create structured data using concise syntax.

Note:

Please download the demo document from the reference material section for step-by-step guidance.

DEMONSTRATION

Guided Practice

Overview

Duration: 20 minutes

You are working with customer data in a Python program. Your goal is to store, organize, and analyze customer feedback using basic Python data structures. You will use lists, tuples, dictionaries, and sets to manage customer records and loops and conditionals to analyze the data.

Key Takeaways

- Lists, Tuples, and Dictionaries are fundamental Python data structures used for storing and managing collections of data efficiently.
- Loops (for and while) help automate repetitive tasks by iterating over sequences like lists, tuples, and dictionaries.
- Comprehensions (List, Dictionary, and Set) provide a concise and efficient way to create and filter collections in a single line of code.
- Using loops, conditionals, and comprehensions together enhances code readability, optimizes performance, and simplifies data manipulation.

Q&A