Installing Python and Setting Up IDEs (PyCharm, VS Code, etc.)

1. Installing Python

Step 1: Download Python

- Go to the official Python website: python.org.
- Download the latest version of Python for your operating system (Windows, macOS, or Linux).

Step 2: Install Python

- **Windows:**
  - Run the downloaded installer.
  - Ensure to check the box that says “Add Python to PATH” before clicking “Install Now.”
  - Follow the prompts to complete the installation.

- **macOS:**
  - You can use the downloaded installer or install Python using Homebrew by running the command:

    ```
    bash
    brew install python
    ```

- **Linux:**
  - Use your package manager to install Python. For example, on Ubuntu, you can run:

    ```
    bash
    sudo apt update
    ```

sudo apt install python3

Step 3: Verify Installation

- Open a terminal or command prompt and type:

  bash
  python --version

  or

  bash
  python3 --version

- You should see the installed version of Python displayed.

2. Setting Up Integrated Development Environments (IDEs)

A. PyCharm

Step 1: Download PyCharm

- Go to the official JetBrains website: jetbrains.com/pycharm.
- Choose the Community version (free) or the Professional version (paid) and download it for your operating system.

Step 2: Install PyCharm

- Run the installer and follow the installation instructions.
- Choose your desired installation options (e.g., creating desktop shortcuts).

Step 3: Configure PyCharm

- Launch PyCharm after installation.
- On the welcome screen, you can create a new project by clicking on “New Project.”
- Specify the project location and the interpreter to use (Python version).
- Click “Create” to set up your project.

B. Visual Studio Code (VS Code)

Step 1: Download VS Code

- Go to the official Visual Studio Code website: code.visualstudio.com.
- Download the installer for your operating system.

Step 2: Install VS Code

- Run the installer and follow the setup instructions.

Step 3: Install Python Extension

- Launch VS Code after installation.
- Go to the Extensions view by clicking on the Extensions icon in the Activity Bar on the side or pressing Ctrl+Shift+X.
- Search for “Python” and install the official Python extension by Microsoft.

Step 4: Configure VS Code

- Open a new folder or create a new file with a .py extension.
- VS Code may prompt you to install the Python interpreter if it’s not already installed.
- Set the Python interpreter by clicking on the Python version shown in the bottom-left corner and selecting the desired interpreter from the list.

3. Verifying Setup

For Both IDEs:

- Create a simple Python file (e.g., hello.py) with the following code:

  ```
  python
  print("Hello, World!")
  ```

- Run the script:
  - **In PyCharm:** Right-click the file and select "Run 'hello'" or use the run button.
  - **In VS Code:** Open the terminal and run:

bash

python hello.py

- You should see the output Hello, World! in the console.

Conclusion

Now you have Python installed and set up in either PyCharm or VS Code, and you're ready to start developing Python applications! If you have any specific questions or need help with something else, feel free to ask!

Basic Python syntax: statements, indentation, comments

1. Statements

In Python, a statement is a piece of code that performs an action. Statements can be as simple as a single line of code or complex involving multiple components.

Example of a simple statement:

Example

```
print("Hello, World!")
```

This statement prints the string "Hello, World!" to the console.

Multiple statements: You can write multiple statements on the same line by separating them with a semicolon (;), but this is not commonly used for readability

Example:

```
x = 5; y = 10;
print(x + y)
```

2. Indentation

Indentation refers to the spaces at the beginning of a code line.

Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.

Python uses indentation to indicate a block of code.

Example:

```
if 7 > 4:
  print("Seven is
greater than Four!")
```

an error if you skip the indentation:

Example:

Syntax Error:

```
if 7 > 4:
print("Seven is greater than
Four!")
```

The number of spaces is up to you as a programmer, the most common use is four, but it has to be at least one.

Example:

```
if 5 > 2:
print("Five is greater than two!")
if 5 > 2:
      print("Five is greater than two!")
```

You have to use the same number of spaces in the same block of code, otherwise Python will give you an error:

Example:

Syntax Error:

```
if 5 > 2:
 print("Five is greater than two!")
      print("Five is greater than two!")
```

Day 2: Variables and Data Types

Variables –

In Python, a **variable** is a container used to store data or values. You can use a name to hold the data, and this name is called a variable.

Here are some examples:

Python Variable Declaration:

```
x = 10      # integer type
name = "Sayem"  # string type
pi = 3.14    # float type
```

Variable Naming Rules:

1. A variable name must start with a letter or an underscore _.
2. A variable name cannot start with a number, for example, 2name is invalid.
3. There should be no spaces in the variable name.
4. You cannot use Python keywords as variable names.

Data Types-

In Python, data types represent the kind of data a variable can hold. Different types of data are stored in different ways, and Python automatically assigns a data type when you assign a value to a variable.

Here are the most common Python data types:

1. Numeric Types

- **int**: Integer, whole numbers (positive or negative), e.g., `x = 10`
- **float**: Floating-point number, used for decimal numbers, e.g., `y = 3.14`
- **complex**: Complex numbers, written with a real and an imaginary part, e.g., `z = 3 + 4j`

2. Text Type

- **str**: String, used for textual data, enclosed in single or double quotes, e.g., `name = "Sayem"`

3. Sequence Types

- **list**: Ordered and mutable collection, e.g., `fruits = ["apple", "banana", "cherry"]`
- **tuple**: Ordered and immutable collection, e.g., `colors = ("red", "green", "blue")`
- **range**: Generates a sequence of numbers, often used in loops, e.g., `range(0, 10)`

4. Mapping Type

- **dict**: Dictionary, stores data as key-value pairs, e.g., `person = {"name": "Sayem", "age": 25}`

5. Set Types

- **set**: Unordered and unique collection, e.g., `unique_nums = {1, 2, 3}`
- **frozenset**: Immutable set, similar to a set but cannot be changed, e.g., `frozen = frozenset([1, 2, 3])`

6. Boolean Type

- **bool**: Represents `True` or `False`, e.g., `is_adult = True`

7. Binary Types

- **bytes**: Immutable sequence of bytes, e.g., b = b"Hello"
- **bytearray**: Mutable sequence of bytes, e.g., b_arr = bytearray(5)
- **memoryview**: Provides memory access to certain objects like bytes.

```
x = 10          # int
y = 3.14        # float
name = "Sayem"  # str
is_valid = True  # bool
```

Understanding dynamic typing

Dynamic typing is a programming paradigm where a variable's type is determined at runtime instead of being explicitly declared at compile time, allowing for greater flexibility and adaptive code structures. Key characteristics include late binding, where variable bindings occur during execution; duck typing, which assesses compatibility based on necessary methods or attributes rather than declared types; reduced boilerplate code, as explicit type declarations are unnecessary; and increased flexibility for fluid code design. Advantages of dynamic typing include rapid development, suitable for prototyping and experimentation without recompilation, and its prevalence in scripting languages like Python, JavaScript, and Ruby. However, it also has drawbacks, such as the potential for runtime errors due to a lack of compile-time checks, debugging challenges since type-related errors may surface only at runtime, and possible performance overhead from runtime type checks. Ultimately, the choice between static and dynamic typing depends on project requirements, team preferences, and the characteristics of the programming language, balancing type safety and performance against flexibility and development speed.

Example:

```
def greet(name):
  print("Hello, " + name + "!")
```

Day 3: Introduction to Type Casting and Basic Operations

Type casting is the process of converting a variable from one data type to another. In programming, this is often necessary when you want to perform operations on values of different types or when you need to ensure that a value is in the correct format for processing. Python provides built-in functions for type casting, allowing developers to convert between types easily.

Basic Type Casting Functions in Python:

1. int(): Converts a value to an integer.

```
x = int(3.14)   # x will be 3
y = int("10")   # y will be 10
```

2. float(): Converts a value to a float.

```
a = float(10)     # a will be
```

3.str(): Converts a value to a string.

```
name = str(100)   # name will be "100"
status = str(True)   # status will be "True"
```

4. list(): Converts a sequence (like a string or a tuple) to a list

```
chars = list("hello")   # chars will be ['h', 'e', 'l', 'l',
'o']
```

5. tuple(): Converts a sequence to a tuple.

```
numbers = tuple([1, 2, 3])   # numbers will be (1, 2, 3)
```

Basic Operations:

Python supports various basic operations that can be performed on different data types:

1. **Arithmetic Operations**:
    - Addition (+)
    - Subtraction (-)
    - Multiplication (*)
    - Division (/)
    - Floor Division (//)
    - Modulus (%)
    - Exponentiation (**)

```
a = 10
b = 3
sum = a + b              # 13
division = a / b         # 3.333...
integer_division = a // b  # 3
modulus = a % b          # 1
power = a ** b           # 1000
```


Comparison Operations: These operations return Boolean values (True or False).

- Equal (==)
- Not Equal (!=)
- Greater than (>)
- Less than (<)
- Greater than or equal to (>=)
- Less than or equal to (<=)

```
x = 8
y = 5

# Equal (==)
print(x == y)  # False, because 8 is not equal to 5

# Not Equal (!=)
print(x != y)  # True, because 8 is not equal to 5
```

Day 4: Strings - Introduction

Strings - Introduction

A **string** in Python is a sequence of characters enclosed in single quotes ('...'), double quotes ("..."), or triple quotes ('''...''' or """..."""") for multiline strings. Strings are widely used to store and manipulate text.

*1. **String Declaration and Basic Operations***

Declaring Strings:

```
# Using single quotes
greeting = 'Hello'

# Using double quotes
name = "Sayem"

# Using triple quotes for multiline strings
multiline_string = '''This is a
multiline string'''
```

Basic String Operations:

- **Concatenation** (joining two or more strings):

```
full_greeting = greeting + ", " + name  # "Hello, Sayem"
```

- **Repetition** (repeating a string multiple times):

```
repeated_string = greeting * 3
#"HelloHelloHello"
```

- **Length** (finding the number of characters in a string):

```
length = len(greeting)  # 5
```


2. Accessing and Updating Strings

Accessing characters in a string: Strings are indexed, with the first character having index 0. You can access individual characters or slices of a string using square brackets.

```
greeting = "Hello"

# Accessing individual characters
first_letter = greeting[0]  # 'H'
last_letter = greeting[-1]  # 'o' (negative index counts from the end)

# Slicing a string
substring = greeting[1:4]  # 'ell' (from index 1 to 3)
```

Updating strings: Strings in Python are **immutable**, meaning their content cannot be changed directly. However, you can create a new string by modifying an existing one.

```
new_greeting = greeting.replace("Hello", "Hi")  # 'Hi'
```

2. String Formatting

```
new_greeting = greeting.replace("Hello", "Hi")
# 'Hi'
```

Python provides several ways to format strings, allowing you to insert values into a string.

A. Using f-strings (formatted string literals): F-strings are introduced in Python 3.6+ and are an easy and readable way to embed expressions inside string literals.

```
name = "Sayem"
age = 25

formatted_string = f"My name is {name} and I am {age} years old."
print(formatted_string)  # "My name is Sayem and I am 25 years
old."
```

Day 5: Advanced String Operations

List of String Methods:

1. **upper()**

2. **lower()**
3. **capitalize()**
4. **replace(old, new)**
5. **strip()**
6. **split(delimiter)**
7. **join()**
8. **find(substring)**
9. **count(substring)**
10. **len()** (built-in function)
11. **str()** (built-in function)

Examples for Each String Method:

```
# Example string
text = "  Hello, Python World!  "
word_list = ['Python', 'is', 'awesome']

# 1. upper(): Converts all characters to uppercase
print(text.upper())  # Output: "  HELLO, PYTHON WORLD!  "

# 2. lower(): Converts all characters to lowercase
print(text.lower())  # Output: "  hello, python world!  "

# 3. capitalize(): Capitalizes the first character of the string
print(text.capitalize())  # Output: "  hello, python world!  "

# 4. replace(old, new): Replaces occurrences of 'old' with 'new'
print(text.replace("Python", "Programming"))  # Output: "  Hello,
Programming World!  "

# 5. strip(): Removes leading and trailing whitespace
print(text.strip())  # Output: "Hello, Python World!"

# 6. split(delimiter): Splits the string into a list of substrings
based on the delimiter (default is whitespace)
print(text.split())  # Output: ['Hello,', 'Python', 'World!']

# 7. join(): Joins a list of strings with a specified delimiter
print(" ".join(word_list))  # Output: "Python is awesome"

# 8. find(substring): Returns the index of the first occurrence of
'substring', or -1 if not found
print(text.find("Python"))  # Output: 8

# 9. count(substring): Counts the number of times 'substring' appears
in the string
print(text.count("o"))  # Output: 3

# 10. len(): Returns the length of the string (number of characters)
print(len(text))  # Output: 23

# 11. str(): Converts other data types to string
number = 123
print(str(number))  # Output: "123"
```

Explanation:

1. **upper()**: Converts the string to all uppercase letters.
2. **lower()**: Converts the string to all lowercase letters.
3. **capitalize()**: Capitalizes only the first letter of the string.
4. **replace(old, new)**: Replaces all occurrences of old substring with new substring.
5. **strip()**: Removes spaces from the beginning and the end of the string.
6. **split(delimiter)**: Splits the string into a list of substrings. By default, it splits by whitespace.
7. **join()**: Joins a list of strings into a single string, with the specified separator.
8. **find(substring)**: Returns the index where the substring first occurs in the string, or -1 if not found.
9. **count(substring)**: Counts how many times a substring occurs within the string.
10. **len()**: Returns the total number of characters in the string.
11. **str()**: Converts a number or other data types into a string format.

Day 6: Operators Operands

Operators and Operands in Python

In Python, **operators** are symbols used to perform operations on variables and values (known as **operands**). Python supports various types of operators, including **arithmetic**, **comparison**, **logical**, **assignment**, **boolean**, and **membership** operators.

1. Arithmetic Operators

- + : Addition
- - : Subtraction
- * : Multiplication
- / : Division (float)
- // : Integer Division (floor division)
- % : Modulus (remainder)
- ** : Exponentiation (power)

```
x = 10
y = 3

addition = x + y          # 13
subtraction = x - y       # 7
multiplication = x * y    # 30
division = x / y          # 3.333...
integer_division = x // y # 3
modulus = x % y           # 1
power = x ** y            # 1000
```

2. Comparison Operators

- == : Equal to
- != : Not equal to
- > : Greater than
- < : Less than
- >= : Greater than or equal to
- <= : Less than or equal to

```
x = 10
y = 5

print(x == y)   # False
print(x != y)   # True
print(x > y)    # True
print(x < y)    # False
print(x >= y)   # True
print(x <= y)   # False
```

3. Logical Operators

- and : Logical AND
- or : Logical OR
- not : Logical NOT

```
x = 10
y = 5

print(x > 3 and y < 10)   # True, both conditions are True
print(x > 10 or y < 10)   # True, one condition is True
print(not(x > 5))         # False, since x > 5 is True, `not` makes it False
```

4. A

- -= : Subtract and assign
- *= : Multiply and assign
- /= : Divide and assign
- //= : Floor divide and assign
- %= : Modulus and assign
- **= : Exponent and assign

```
x = 10

x += 5  # Equivalent to: x = x + 5 (x becomes 15)
x -= 3  # Equivalent to: x = x - 3 (x becomes 12)
x *= 2  # Equivalent to: x = x * 2 (x becomes 24)
x /= 4  # Equivalent to: x = x / 4 (x becomes 6.0)
```

5. Boolean Operators

- True : Boolean True value
- False : Boolean False value

```
is_sunny = True
is_raining = False

print(is_sunny)     # Output: True
print(is_raining)   # Output: False
```

6. Membership Operators

- in : Returns True if a value is found in a sequence
- not in : Returns True if a value is not found in a sequence

```
fruit = "apple"
print("a" in fruit)       # True, 'a' is in "apple"
print("b" not in fruit)   # True, 'b' is not in
"apple"
```

Day 7 : Loops – Introduction

Conditional Statements in Python

Conditional statements allow you to execute specific blocks of code based on certain conditions. These conditions typically involve comparison or logical operators, which return either True or False.

1. if Statement

The if statement allows you to execute a block of code only if a condition is True.

```
x = 10
if x > 5:
    print("x is greater than 5")
```

Explanation: In this example, the condition x > 5 evaluates to True, so the code inside the if block runs, printing the message.

2. else Statement

The else statement is used to run a block of code if the condition in the if statement is False

```
x = 3
if x > 5:
    print("x is greater than 5")
else:
    print("x is less than or equal to 5")
```

Explanation: Since x > 5 is False, the else block is executed, printing "x is less than or equal to 5."

3. elif Statement

The elif (short for "else if") statement allows you to check multiple conditions. If the if condition is False, Python checks the elif condition.

```
x = 5
if x > 5:
    print("x is greater than 5")
elif x == 5:
    print("x is equal to 5")
else:
    print("x is less than 5")
```

Explanation: The elif statement checks if x == 5. Since this condition is True, the corresponding block of code runs, and "x is equal to 5" is printed.

4. Nested Conditionals

You can nest if, elif, and else statements within one another to check multiple conditions in a hierarchical manner.

Explanation: In this example, the outer if checks whether x > 5. Since this is True, the inner if checks if y < 10. Since both conditions are True, "x is greater than 5 and y is less than 10" is printed.

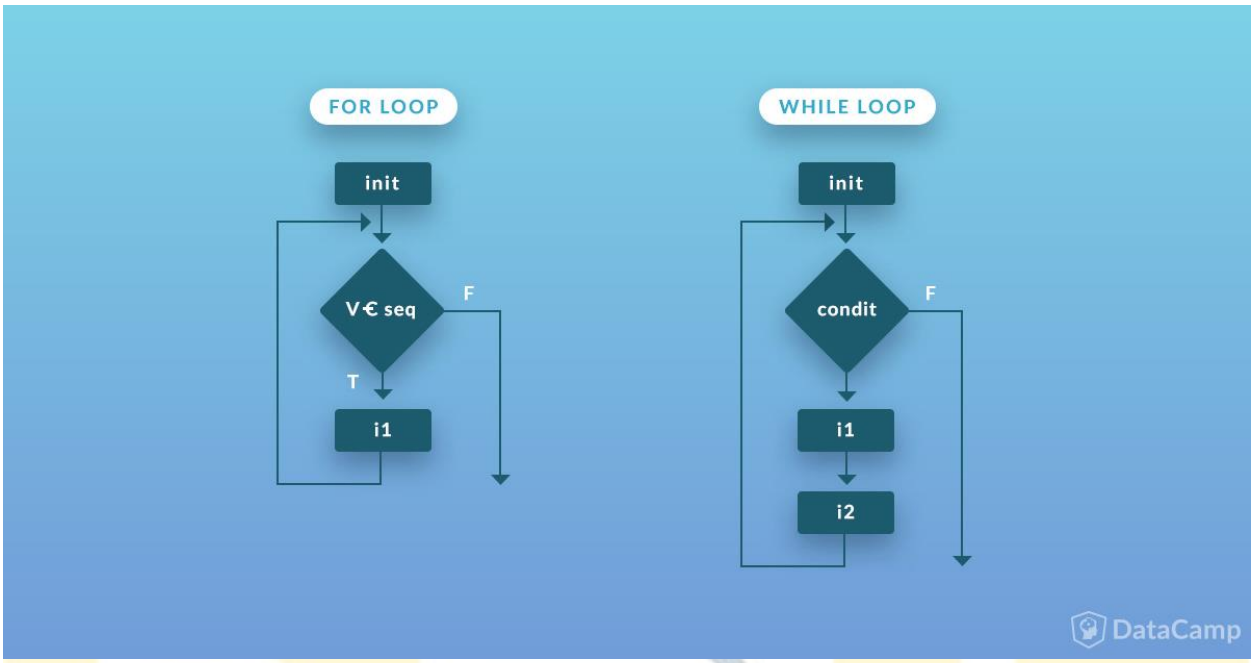
```
x = 10
y = 5

if x > 5:
    if y < 10:
        print("x is greater than 5 and y is less than 10")
    else:
        print("y is 10 or greater")
else:
    print("x is 5 or less")
```

Summary:

- **if:** Executes a block of code if the condition is True.
- **else:** Executes a block of code if the condition in the if statement is False.
- **elif**: Checks additional conditions if the first if condition is False.
- **Nested conditionals**: Allows combining multiple conditions within one another.

Day 8 : Loops – Introduction

Loops allow you to execute a block of code repeatedly as long as a condition is true. Python supports two main types of loops: **for loops** and **while loops**. Each type is useful for different scenarios.



Loops are essential in Python, as well as in any programming language, because they allow you to execute a block of code repeatedly. In many situations, you'll need to run the same code multiple times without manually writing it out each time. Loops help you automate this process, making your programs efficient and concise.

In this Python loops tutorial, we'll cover the following key concepts:

- **The Python while loop**: You'll learn how to create and use a while loop, especially in data science applications, by going through some hands-on coding challenges.
- **The for loop**: This section will show you how to construct and use a for loop in real-life programming scenarios.
- **Difference between while and for loops**: You'll explore when to use each type of loop and understand the differences between them.
- **Nested loops**: Learn how to use loops inside other loops to handle more complex iterations.
- **Using the break and continue keywords**: These powerful tools allow you to control the flow of loops by breaking out of a loop or skipping specific iterations.
- **The range() vs xrange() functions**: Understand the differences between these two functions and their applications.

The while Loop

The while loop is one of the most straightforward and intuitive loops you'll encounter when learning to program. The concept is simple: as long as a given condition remains true, the code inside the while loop will keep running.

Definition: A while loop is a programming construct that executes a block of code repeatedly while a given condition evaluates to True.

To implement a while loop in Python, you need three components:

1. The **while** keyword
2. A **condition** that evaluates to either True or False
3. A block of code to be executed repeatedly

1. for Loops

A for loop in Python is used to iterate over a sequence (such as a list, tuple, dictionary, set, or string) or any other iterable object.

Syntax:

```
# for variable in sequence:
#     # code block

fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Explanation: The for loop goes through each element in the fruits list and prints it. The loop iterates until all elements are covered.

2. while Loops

A while loop repeatedly executes the code block as long as a given condition is true.

Syntax:

```
#while condition:
    # code block
count = 1
while count <= 5:
    print("Count is:", count)
    count += 1
```

Explanation:

- The condition `count <= 5` controls the loop.
- As long as the condition is `True`, the code inside the loop runs.
- The value of `count` increases by 1 with each iteration, so the loop eventually stops when the condition becomes `False`.

This allows you to avoid writing repetitive code and helps manage tasks where multiple iterations are required based on a condition.

Day 9: Advanced Loops

Advanced Loops in Python

Loops are essential for repetitive tasks in programming, but they also come with advanced use cases, such as **infinite loops**, **nested loops**, and optimizations for performance. In this section, we'll explore these topics in detail and provide practical examples.

1. Infinite Loops

An **infinite loop** occurs when the loop's condition always evaluates to `True`, causing the loop to run indefinitely. This can happen accidentally, or it can be intentional if you want a program to keep running until interrupted.

Example of an Infinite Loop:

```
while True:
    print("This loop will run forever unless stopped")
    break  # You can manually break the loop to stop it
```

Without the break statement, this loop would continue running endlessly. Infinite loops are often used in systems like servers, where the code needs to run continuously until an external condition (like a user command) stops it.

Avoiding Unintentional Infinite Loops:
Always make sure the condition in your while or for loop changes during execution so that the loop can eventually terminate.

2. Nested Loops

A **nested loop** is a loop inside another loop. This is useful when you need to iterate over multiple dimensions of data, like in matrix operations or handling multi-level lists.

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

for row in matrix:
    for num in row:
        print(num, end=" ")
    print()  # Newline after each row
```

Explanation:

- The outer loop iterates through each row in the matrix.
- The inner loop iterates through each number in the row, printing it out.
- The result is a structured output of the matrix, row by row.

Nested loops can also be used for more complex tasks, such as handling 2D arrays, calculating combinations of items, or processing files in directories.

3. Practical Examples of Loops

1. Finding Prime Numbers

Using a for loop to check for prime numbers within a range:

```
for num in range(2, 20):
    is_prime = True
    for i in range(2, num):
        if num % i == 0:
            is_prime = False
            break
    if is_prime:
        print(num, "is a prime number")
```

2. Reading Files Line by Line

Using a `while` loop to read lines from a file until the end of the file:

```
with open("sample.txt", "r") as file:
    line = file.readline()
    while line:
        print(line.strip())
        line = file.readline()
```

4. Performance Considerations

While loops are powerful, they can impact performance if not optimized properly. Here are some key tips to keep in mind:

- **Avoiding unnecessary nesting**: Nested loops, especially when combined with large data sets, can lead to performance degradation. The time complexity increases with each layer of nesting. For example, a nested loop over two large lists may result in an O(n^2) complexity, which can be slow for large inputs.
- **Use `range()` instead of creating lists**: If you're looping through a range of numbers, use `range()` instead of constructing large lists. `range()` is a generator that doesn't consume memory unnecessarily.
- **Break Early**: Use the `break` statement to exit loops as soon as a condition is met, reducing unnecessary iterations.

Example: Optimizing a search loop with `break`:

```
numbers = [10, 20, 30, 40, 50]
target = 30
for num in numbers:
    if num == target:
        print(f"Found {target}")
        break  # Stop the loop early after finding
the target
```

Summary:

- **Infinite loops**: Can be used intentionally or occur accidentally if the loop's condition is always true.
- **Nested loops**: Allow you to handle multi-dimensional data or perform complex tasks but can be resource-intensive.
- **Practical applications**: Loops are used in tasks like finding prime numbers or reading files, but you should always consider performance.
- **Optimizing loops**: Break early, avoid deep nesting, and use efficient built-in functions to ensure optimal performance.

By understanding these advanced loop concepts, you can write more efficient and effective Python code, especially in data processing and repetitive tasks.

Day 10: Creation of lists

1. Indexing and Slicing Lists:

- **Indexing**: This allows you to access individual elements from a list using their index, where the first element is at index `0`, and you can also use negative indexing to access elements from the end of the list.
- **Slicing**: You can extract a portion of the list by specifying a start, stop, and an optional step in the form `[start:end:step]`. It creates a new sub-list without modifying the original.

2. List Operations:

- **Concatenation**: Using the `+` operator, you can combine two or more lists into a single list.
- **Repetition**: The `*` operator allows you to repeat the elements of a list a specified number of times, creating a longer list.

3. List Methods:

- **`append()`**: Adds a single element to the end of the list.
- **`extend()`**: Appends elements from another iterable (like another list) to the end of the list.
- **`insert()`**: Inserts an element at a specified index without replacing any existing elements.
- **`remove()`**: Removes the first occurrence of a specific element from the list.
- **`pop()`**: Removes and returns the element at the specified index. If no index is specified, it removes the last element.
- **`sort()`**: Sorts the list in ascending order by default.
- **`reverse()`**: Reverses the order of the elements in the list in place.

Example: . **List Methods**

Exa

```
# Example list
my_list = [1, 2, 3, 4]

# 1. append()
my_list.append(5)
print("After append:", my_list)

# 2. extend()
my_list.extend([6, 7])
print("After extend:", my_list)

# 3. insert()
my_list.insert(0, 0)
print("After insert:", my_list)

# 4. remove()
my_list.remove(3)
print("After remove:", my_list)

# 5. pop()
popped_element = my_list.pop()
print("Popped element:", popped_element)
print("After pop:", my_list)

# 6. sort()
my_list.sort()
print("After sort:", my_list)
```

```
# 7. reverse()
my_list.reverse()
print("After reverse:", my_list)

# 8. index()
index_of_2 = my_list.index(2)
print("Index of 2:", index_of_2)

# 9. count()
count_of_4 = my_list.count(4)
print("Count of 4:", count_of_4)

# 10. clear()
my_list.clear()
print("After clear:", my_list)

# 11. copy()
original_list = [1, 2, 3, 4]
copied_list = original_list.copy()
print("Copied list:", copied_list)
```

Day 11: Creation of tuples

1. **Usage and Advantages of Tuples Over Lists**:

   Immutability: Tuples are immutable, meaning once they are created, their elements cannot be changed, added, or removed. This makes tuples ideal for fixed collections of items where data integrity is crucial.

   Performance: Tuples are generally more memory-efficient than lists. Their immutability allows Python to optimize their storage, leading to better performance in some scenarios.

   Hash ability: Tuples can be used as keys in dictionaries and elements in sets because they are hash able, while lists cannot.

   Multiple Return Values: Functions can return multiple values packed in a tuple, allowing easy grouping of related data.

   Data Integrity: When you need to ensure that a collection remains unchanged, tuples enforce this requirement.

2. **Immutability of Tuples**:

   Tuples cannot be modified after creation. This means you cannot:

   - Change an element's value.
   - Add new elements.
   - Remove elements.

   This immutability provides safety for data that should remain constant throughout the program.

3. **Tuple Methods**:

   `count(value)`: Returns the number of occurrences of a specified value in the tuple.

   `index(value)`: Returns the index of the first occurrence of a specified value in the tuple. Raises a `Value Error` if the value is not found.

Example Code

Here's a small code snippet demonstrating tuples:

```
# Creating a tuple
my_tuple = (1, 2, 3, 4, 5)
print("Original tuple:", my_tuple)

# Accessing elements using indexing
print("First element:", my_tuple[0])   # Output: 1
print("Last element:", my_tuple[-1])    # Output: 5

# Slicing a tuple
sub_tuple = my_tuple[1:4]  # Extract elements from index 1 to 3
print("Sliced tuple:", sub_tuple)  # Output: (2, 3, 4)

# Using step in slicing
step_tuple = my_tuple[::2]  # Extract every second element
print("Every second element:", step_tuple)  # Output: (1, 3, 5)
```

```
# Creating a tuple
my_tuple = (1, 2, 3, 4, 5)

# Trying to change an element (this will raise an error)
# my_tuple[1] = 10  # Uncommenting this line will raise TypeError

# Using count() method
print("Count of 2:", my_tuple.count(2))

# Using index() method
print("Index of 3:", my_tuple.index(3))
```

Day 12: Creation of sets

1. **Usage and Advantages of Sets**:

   No Duplicates: Sets automatically eliminate duplicate elements, ensuring that each element is unique. This is useful for collecting unique items.

   Unordered: Sets do not maintain any specific order of elements, which can simplify certain operations and comparisons.

   Performance: Sets are optimized for membership testing. Checking if an item is in a set is faster than in a list.

   Mutability: Sets can be modified after creation. You can add or remove elements as needed, unlike tuples.

   Set Operations: Sets support mathematical operations like union, intersection, and difference, making them powerful for data analysis.

Example Code :

Here's a small code snippet demonstrating tuples:

```
# Creating a set
my_set = {1, 2, 3}

# Adding elements
my_set.add(4)
print("After add:", my_set)  # Output: {1, 2, 3, 4}

# Removing an element
my_set.remove(2)
print("After remove:", my_set)  # Output: {1, 3, 4}

# Discarding a non-existent element
my_set.discard(5)  # No error
print("After discard:", my_set)  # Output: {1, 3, 4}

# Popping an element
popped_element = my_set.pop()
print("Popped element:", popped_element)  # Output: (an arbitrary element)
print("After pop:", my_set)  # Output: set with one less element

# Set operations
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print("Union:", set1.union(set2))         # Output: {1, 2, 3, 4, 5}
print("Intersection:", set1.intersection(set2))  # Output: {3}
print("Difference:", set1.difference(set2))  # Output: {1, 2}
```

Day 13: Creation of dictionaries

1. Accessing, Updating, and Deleting Elements

Dictionaries store data as key-value pairs, allowing you to access values by their keys. You can update entries by reassigning values to keys and delete entries using the `del` statement to remove specific key-value pairs.

2. Dictionary Methods (keys, values, items, update)

Dictionaries offer methods like `keys()`, `values()`, and `items()` to access all keys, values, or key-value pairs, respectively. The `update()` method allows you to merge another dictionary into the current one, updating or adding entries as needed.

3. Use Cases (Fast Lookups, Key-Value Pairing)

Dictionaries are ideal for fast lookups, providing constant-time access to values by keys. They are commonly used for pairing related data (like names with numbers), making them

Example Code :

```
# Creating a dictionary
person = {"name": "Alice", "age": 30, "city": "New York"}

# Accessing an element
print(person["name"])  # Output: Alice

# Updating an element
person["age"] = 31

# Deleting an element
del person["city"]

# Using dictionary methods
print(person.keys())    # Output: dict_keys(['name', 'age'])
print(person.values())  # Output: dict_values(['Alice', 31])
print(person.items())   # Output: dict_items([('name', 'Alice'), ('age', 31)])

# Updating with another dictionary
person.update({"country": "USA", "profession": "Engineer"})

# Resulting dictionary
print(person)
# Output: {'name': 'Alice', 'age': 31, 'country': 'USA', 'profession': 'Engineer'}
```

```
# Initializing a dictionary
student = {
    "name": "John",
    "age": 21,
    "courses": ["Math", "Science"],
    "GPA": 3.8
}

# Accessing elements
print(student["name"])      # Output: John
print(student["courses"])   # Output: ['Math', 'Science']

# Updating an element
student["GPA"] = 3.9
print(student["GPA"])       # Output: 3.9

# Adding a new element
student["graduation_year"] = 2025
print(student)              # Output includes new key-value pair 'graduation_year': 2025

# Deleting an element
del student["age"]
print(student)              # Output excludes 'age'

# Dictionary methods
print(student.keys())       # Output: dict_keys(['name', 'courses', 'GPA', 'graduation_year'])
```