Introduction to Python Programming

1. What is Python?

Python is a high-level, interpreted programming language known for its simplicity and readability. It allows developers to write clear programs for both small and large-scale projects.
Python is used for:

- Web development (e.g., Django, Flask)
- Data Science and Machine Learning (e.g., Pandas, TensorFlow)
- Automation (e.g., scripting tasks)
- Software development and more.

2. Why is Python Popular?

- **Easy to learn**: Python's syntax is simple and very close to natural language, making it a great language for beginners.
- **Community support**: It has a massive community, meaning you'll find lots of tutorials, resources, and libraries.
- **Cross-platform**: Python works on different operating systems (Windows, macOS, Linux, etc.).
- **Versatile**: Whether it's web development, data analysis, or automation, Python has libraries for almost everything.

3. Setting Up Python Environment

Step 1: Download Python

- **Visit**: Python's official website (https://www.python.org/)
- **Download the latest version** of Python for your operating system (Windows, macOS, or Linux).
- **Installation on Windows**:
  - Check the option **"Add Python to PATH"** during installation.
  - Choose "Install Now" or customize installation options if needed.

Step 2: Installing IDE (Integrated Development Environment)

- **IDE Options**: Python can be written in any text editor, but for ease, it's better to use an IDE. Some popular ones are:
  - **PyCharm**: A full-featured IDE (Download from here (https://www.jetbrains.com/pycharm/)).
  - **VS Code**: A lightweight editor with Python support (Download from here (https://code.visualstudio.com/)) - **Recommended**
  - **Jupyter Notebook**: Great for data science and learning Python interactively (Install with `pip install notebook`).

Step 3: Verify Installation

- Open the command prompt or terminal.
- Type `python --version` or `python3 --version` to verify that Python is successfully installed.

4. Writing Your First Python Program

Let's write a simple program to understand how Python works.

Step 1: Open a Text Editor or IDE

- Open any text editor like Notepad or an IDE like PyCharm/VS Code.

Step 2: Write Your First Python Code

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

This code will print "Hello, World!" on the screen.

Step 3: Run the Program

- **On IDE**: Click the "Run" button.
- **On Terminal**: Save the file as `hello.py` and navigate to the file location in the terminal. Then run:

```
python hello.py
```

5. Python as an Interpreted Language

Unlike other compiled languages like C or Java, Python executes the code line by line, which makes debugging easy. Python doesn't require you to compile your code into machine language; the Python interpreter takes care of it.

Benefits of Interpreted Language:

- **Easier debugging**: Errors are reported line by line.
- **Faster development**: You can directly run the code without worrying about compiling.

6. Key Features of Python

1. **Simple Syntax**: Easy to read and write, similar to English.
2. **Interpreted**: Python is executed line by line.
3. **Dynamically Typed**: No need to declare variable types explicitly.
4. **Object-Oriented**: Supports OOP (Object-Oriented Programming) like classes and objects.
5. **Rich Standard Library**: Comes with lots of built-in modules and functions.

Homework

1. **Download and install Python** on your system. - Reference Video (https://www.youtube.com/watch?v=Od3ltO2aKAY)
2. **If you don't have a laptop**, Install this app - Python Code-Pad (https://play.google.com/store/apps/details?id=com.markodevcic.python_code_pad&hl=en_IN)
3. **Write your first Python program** to print your name.

```
print("Namaste, nanna hesaru [Your Name]!")
```

3. **Practice** running your Python code through both an IDE and a terminal.
4. **Build in Public** Create a post on Linkedin/X and share that you are starting the course and its day 1. (Use #engineeringinkannada and mention me)

Python Basics - 1

1. Variables in Python

Variables in Python are used to store data values. They are created when you assign a value to them, and you don't need to declare their type (Python is dynamically typed).

Syntax for Variable Assignment:

```
x = 5  # Assigning an integer value to the variable x
y = "Hello"  # Assigning a string value to the variable y
```

Variable Naming Rules:

- Variable names can contain letters (a-z, A-Z), numbers (0-9), and underscores (_).
- Variable names must start with a letter or an underscore.
- Variable names are case-sensitive (`Name` and `name` are different).

Example:

```
age = 25
name = "John"
is_student = True
```

2. Data Types in Python

Python has various built-in data types. Some common ones are:

- **int**: For integers (e.g., 1, -3, 100)
- **float**: For floating-point numbers (e.g., 3.14, -0.001)
- **str**: For strings (e.g., "Hello", "Python")
- **bool**: For boolean values (True or False)

Type Checking:

You can use the `type()` function to check the type of a variable.

```
x = 10
print(type(x))  # Output: <class 'int'>
```

3. Type Conversion

Python allows you to convert between data types using functions like `int()`, `float()`, `str()`, etc.

Example:

```
x = "10"  # x is a string
y = int(x)  # Convert string to integer
z = float(y)  # Convert integer to float
print(z)  # Output: 10.0
```

4. Arithmetic Operators

Python supports basic arithmetic operations like addition, subtraction, multiplication, division, and more.

Common Operators:

- `+` (Addition)
- `-` (Subtraction)
- `*` (Multiplication)
- `/` (Division)
- `//` (Floor Division)
- `%` (Modulus)
- `**` (Exponentiation)

Examples:

```
a = 10
b = 3
print(a + b)  # Output: 13
print(a - b)  # Output: 7
print(a * b)  # Output: 30
print(a / b)  # Output: 3.3333...
print(a // b)  # Output: 3 (Floor Division)
print(a % b)  # Output: 1 (Modulus)
print(a ** b)  # Output: 1000 (Exponentiation)
```

5. Assigning Values to Multiple Variables

Python allows you to assign values to multiple variables in a single line.

Example:

```
x, y, z = 10, 20, 30
print(x)  # Output: 10
print(y)  # Output: 20
print(z)  # Output: 30
```

You can also assign the same value to multiple variables in one line:

```
x = y = z = 100
print(x, y, z)  # Output: 100 100 100
```

6. Variable Reassignment

You can change the value of a variable at any point in your program.

Example:

```
x = 5
print(x)  # Output: 5
x = 10
print(x)  # Output: 10
```

Homework

1. **Arithmetic Practice:** Write a Python program that performs basic arithmetic operations (addition, subtraction, multiplication, and division) on two numbers. Define the two numbers as variables within the code and print the results for each operation.
2. **Swap Two Variables:** Write a Python program that swaps the values of two variables with and without using a third variable.
3. **Build in Public:** Create a post on Linkedin/X and share that you learn variables and data types in python and its day 1. (Use #engineeringinkannada and mention me)

Input/Output, String Manipulation, and Comments

1. Input and Output in Python

1.1 Input from the User:

In Python, we use the `input()` function to take input from the user. The data entered by the user is always received as a string, so if you want to use it as a different data type (e.g., integer or float), you'll need to convert it using type conversion functions like `int()` or `float()`.

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))  # Convert input to integer
```

1.2 Output to the Console:

The `print()` function is used to display output to the console. You can use it to display text, variables, or results of expressions.

```
print("Hello, " + name + "! You are " + str(age) + " years old.")
```

You can also use **f-strings** (formatted string literals) for more readable code:

```
print(f"Hello, {name}! You are {age} years old.")
```

---

2. String Manipulation

Strings are sequences of characters. Python provides many useful methods to manipulate strings.

2.1 Common String Operations:

- **Concatenation**: Joining two or more strings together using the `+` operator.

  ```
  first_name = "John"
  last_name = "Doe"
  full_name = first_name + " " + last_name
  print(full_name)  # Output: "John Doe"
  ```

- **Repetition**: Repeating a string multiple times using the `*` operator.

  ```
  greeting = "Hello! " * 3
  print(greeting)  # Output: "Hello! Hello! Hello! "
  ```

2.2 String Methods:

- `upper()`: Converts a string to uppercase.
- `lower()`: Converts a string to lowercase.
- `strip()`: Removes leading and trailing spaces from a string.
- `replace(old, new)`: Replaces a substring with another string.

```
message = "  Hello, World!  "
print(message.strip())  # Output: "Hello, World!"
print(message.upper())  # Output: "HELLO, WORLD!"
print(message.replace("World", "Python"))  # Output: "Hello, Python!"
```

2.3 Accessing String Characters:

You can access individual characters in a string using **indexing**. Python uses zero-based indexing, so the first character has an index of 0.

```
text = "Python"
print(text[0])  # Output: P
print(text[2])  # Output: t
```

You can also use **negative indexing** to start counting from the end of the string.

```
print(text[-1])  # Output: n
print(text[-3])  # Output: h
```

2.4 Slicing Strings:

You can extract a portion (substring) of a string using slicing.

```
text = "Python Programming"
print(text[0:6])  # Output: Python (extracts from index 0 to 5)
print(text[:6])  # Output: Python (same as above)
print(text[7:])  # Output: Programming (from index 7 to the end)
```

---

3. Comments in Python

Comments are ignored by the Python interpreter and are used to explain the code or leave notes for yourself or others. They do not affect the execution of the program.

- **Single-line comments** start with `#`:

  ```
  # This is a single-line comment
  print("Hello, World!")
  ```

- **Multi-line comments** can be written using triple quotes (`"""` or `'''`). These are often used to write detailed explanations or temporarily block sections of code:

  ```
  """
  This is a multi-line comment.
  It can span multiple lines.
  """
  print("Hello, Python!")
  ```

---

4. Escape Sequences

Escape sequences are special characters in strings that start with a backslash (`\`). They are used to represent certain special characters.

Some commonly used escape sequences:

- `\n`: New line
- `\t`: Tab space
- `\\`: Backslash

Example:

```
print("Hello\nWorld")  # Output:
# Hello
# World

print("Hello\tPython")  # Output: Hello    Python
```

Homework

1. **Simple Greeting Program**: Write a Python program that asks the user for their name and age, then prints a personalized greeting message. Use both the `+` operator and f-strings for output.

   Example:

```
Enter your name: Alice
Enter your age: 25
Output: Hello, Alice! You are 25 years old.
```

2. **String Manipulation Exercise**: Write a Python program that:
    - Takes a sentence as input from the user.
    - Prints the sentence in all uppercase and lowercase.
    - Replaces all spaces with underscores.
    - Removes leading and trailing whitespace.

    Example:

```
Input: "   Python is awesome!   "
Output:
Uppercase: "PYTHON IS AWESOME!"
Lowercase: "python is awesome!"
Replaced: "___Python_is_awesome!___"
Stripped: "Python is awesome!"
```

3. **Character Counter**: Write a Python program that:
    - Asks the user for a string.
    - Prints how many characters are in the string, excluding spaces.

    Example:

```
Input: "Hello World"
Output: "Number of characters (excluding spaces): 10"
```

4. **Escape Sequence Practice**: Write a Python program that uses escape sequences to print the following output:

    Example:

```
Hello
    World
This is a backslash: \
```

Operators in Python

1. Assignment Operators

Assignment operators are used to assign values to variables. The simplest one is `=` which assigns the value on the right to the variable on the left. There are also compound assignment operators that combine arithmetic operations with assignment.

Common Assignment Operators:

- `=`: Assigns value on the right to the variable on the left.
- `+=`: Adds right operand to the left operand and assigns the result to the left operand.
- `-=`: Subtracts the right operand from the left operand and assigns the result to the left operand.
- `*=`: Multiplies the left operand by the right operand and assigns the result to the left operand.
- `/=`: Divides the left operand by the right operand and assigns the result to the left operand.
- `%=`: Takes modulus of left operand by right operand and assigns the result to the left operand.

Examples:

```
x = 5  # Assigns 5 to x
x += 3  # Equivalent to x = x + 3, now x is 8
x -= 2  # Equivalent to x = x - 2, now x is 6
x *= 4  # Equivalent to x = x * 4, now x is 24
x /= 6  # Equivalent to x = x / 6, now x is 4.0
```

---

2. Comparison Operators

Comparison operators are used to compare two values. They return either `True` or `False` depending on the condition.

Common Comparison Operators:

- `==`: Checks if two values are equal.
- `!=`: Checks if two values are not equal.
- `>`: Checks if the left operand is greater than the right operand.
- `<`: Checks if the left operand is less than the right operand.
- `>=`: Checks if the left operand is greater than or equal to the right operand.
- `<=`: Checks if the left operand is less than or equal to the right operand.

Examples:

```
a = 10
b = 20

print(a == b)  # Output: False
print(a != b)  # Output: True
print(a > b)  # Output: False
print(a < b)  # Output: True
print(a >= 10)  # Output: True
print(b <= 25)  # Output: True
```

---

3. Logical Operators

Logical operators are used to combine conditional statements. They evaluate expressions and return either `True` or `False`.

Common Logical Operators:

- `and`: Returns `True` if both conditions are true.
- `or`: Returns `True` if at least one condition is true.
- `not`: Reverses the logical state of its operand (True becomes False, and vice versa).

Examples:

```
x = 5
y = 10
z = 15

# and operator
print(x > 0 and y > 5)  # Output: True (both conditions are True)

# or operator
print(x > 10 or z > 10)  # Output: True (one of the conditions is True)

# not operator
print(not(x > 10))  # Output: True (reverses False to True)
```

---

4. Membership Operators

Membership operators test for membership within a sequence, such as a list, string, or tuple. They return `True` or `False` based on whether the value is found in the sequence.

Membership Operators:

- `in`: Returns `True` if the specified value is found in the sequence.
- `not in`: Returns `True` if the specified value is not found in the sequence.

Examples:

```
my_list = [1, 2, 3, 4, 5]
my_string = "Python"

print(3 in my_list)  # Output: True (3 is in the list)
print(6 not in my_list)  # Output: True (6 is not in the list)
print("P" in my_string)  # Output: True ("P" is in the string)
print("z" not in my_string)  # Output: True ("z" is not in the string)
```

---

5. Bitwise Operators

Bitwise operators perform operations on binary representations of integers. These operators are useful for low-level programming tasks like working with bits and bytes.

Common Bitwise Operators:

- `&`: Bitwise AND (sets each bit to 1 if both bits are 1).
- `|`: Bitwise OR (sets each bit to 1 if one of the bits is 1).
- `^`: Bitwise XOR (sets each bit to 1 if only one of the bits is 1).
- `~`: Bitwise NOT (inverts all the bits).
- `<<`: Left shift (shifts bits to the left by a specified number of positions).
- `>>`: Right shift (shifts bits to the right by a specified number of positions).

Examples:

```
a = 5  # In binary: 101
b = 3  # In binary: 011

# Bitwise AND
print(a & b)  # Output: 1 (binary: 001)

# Bitwise OR
print(a | b)  # Output: 7 (binary: 111)

# Bitwise XOR
print(a ^ b)  # Output: 6 (binary: 110)

# Bitwise NOT
print(~a)  # Output: -6 (inverts all bits)

# Left shift
print(a << 1)  # Output: 10 (binary: 1010)

# Right shift
print(a >> 1)  # Output: 2 (binary: 010)
```

---

6. **Arithmetic Operators** (Already Covered in Chapter 2 (https://github.com/chandansgowda/learn-python-in-kannada/blob/main/notes/2.md))

Python supports basic arithmetic operations like addition, subtraction, multiplication, division, and more.

Common Operators:

- `+` (Addition)
- `-` (Subtraction)
- `*` (Multiplication)
- `/` (Division)
- `//` (Floor Division)
- `%` (Modulus)
- `**` (Exponentiation)

Examples:

```
a = 10
b = 3
print(a + b)  # Output: 13
print(a - b)  # Output: 7
print(a * b)  # Output: 30
print(a / b)  # Output: 3.3333...
print(a // b)  # Output: 3 (Floor Division)
print(a % b)  # Output: 1 (Modulus)
print(a ** b)  # Output: 1000 (Exponentiation)
```

Homework

1. **Logical Operator Practice**: Write a Python program that takes two numbers as input from the user and checks if:
   - Both numbers are greater than 10 (using `and`).
   - At least one of the numbers is less than 5 (using `or`).
   - The first number is not greater than the second (using `not`).
2. **Comparison Operator Challenge**: Create a Python program that asks the user for their age and prints:
   - "You are an adult" if the age is greater than or equal to 18.
   - "You are a minor" if the age is less than 18.
   - Use >= and < comparison operators.
3. **Membership Operator Exercise**: Write a Python program that:
   - Takes a string as input from the user.
   - Checks if the letter 'a' is in the string (using `in`).
   - Checks if the string doesn't contain the word "Python" (using `not in`).
4. **Bitwise Operator Task**: Given two integers, write a Python program that:
   - Prints the result of `a & b`, `a | b`, and `a ^ b`.
   - Shifts the bits of `a` two positions to the left (`a << 2`).
   - Shifts the bits of `b` one position to the right (`b >> 1`).

Lists in Python

1. What is a List?

A list is a collection of items that are **ordered**, **mutable** (changeable), and **allow duplicate elements**. Lists can hold items of different data types, such as integers, strings, or even other lists.

Syntax:

```
my_list = [element1, element2, element3, ...]
```

Example:

```
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
mixed = ["apple", 3, True]
```

---

2. Accessing List Elements

You can access individual elements in a list using **indexing**. Remember that Python uses **zero-based indexing**, so the first item is at index 0.

Syntax:

```
list_name[index]
```

Example:

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0])  # Output: apple
print(fruits[2])  # Output: cherry
```

You can also use **negative indexing** to access elements from the end of the list:

```
print(fruits[-1])  # Output: cherry
print(fruits[-2])  # Output: banana
```

---

3. Modifying Lists

Lists are mutable, which means you can change the value of items in a list.

Changing a specific element:

```
fruits[1] = "orange"
print(fruits)  # Output: ['apple', 'orange', 'cherry']
```

Adding elements:

- `append()`: Adds an element to the **end** of the list.

  ```
  fruits.append("grape")
  print(fruits)  # Output: ['apple', 'orange', 'cherry', 'grape']
  ```

- `insert()`: Inserts an element at a **specific index**.

  ```
  fruits.insert(1, "kiwi")
  print(fruits)  # Output: ['apple', 'kiwi', 'orange', 'cherry']
  ```

Removing elements:

- `remove()`: Removes the first occurrence of an element.

  ```
  fruits.remove("orange")
  print(fruits)  # Output: ['apple', 'kiwi', 'cherry']
  ```

- `pop()`: Removes the element at a specific index (or the last item if no index is provided).

  ```
  fruits.pop()  # Removes the last item
  print(fruits)  # Output: ['apple', 'kiwi']

  fruits.pop(0)  # Removes the first item
  print(fruits)  # Output: ['kiwi']
  ```

- `clear()`: Removes all elements from the list.

  ```
  fruits.clear()
  print(fruits)  # Output: []
  ```

---

4. Slicing Lists

You can extract a portion of a list using **slicing**.

Syntax:

```
list_name[start:stop:step]
```

- `start`: The index to start the slice (inclusive).
- `stop`: The index to stop the slice (exclusive).

- `step`: The number of steps to skip elements (default is 1).

Examples:

```
numbers = [0, 1, 2, 3, 4, 5, 6]
print(numbers[1:4])  # Output: [1, 2, 3] (from index 1 to 3)
print(numbers[:4])  # Output: [0, 1, 2, 3] (from start to index 3)
print(numbers[2:])  # Output: [2, 3, 4, 5, 6] (from index 2 to end)
print(numbers[::2])  # Output: [0, 2, 4, 6] (every 2nd element)
```

---

5. List Functions and Methods

Python provides several built-in functions and methods for working with lists.

5.1 Common Functions:

- `len(list)`: Returns the number of elements in the list.

```
print(len(fruits))  # Output: 3
```

- `sorted(list)`: Returns a new sorted list without changing the original list.

```
numbers = [5, 2, 9, 1]
print(sorted(numbers))  # Output: [1, 2, 5, 9]
print(numbers)  # Original list remains unchanged: [5, 2, 9, 1]
```

- `sum(list)`: Returns the sum of elements in a list (for numerical lists).

```
numbers = [1, 2, 3, 4]
print(sum(numbers))  # Output: 10
```

5.2 Common Methods:

- `index(element)`: Returns the index of the first occurrence of the specified element.

```
print(fruits.index("apple"))  # Output: 0
```

- `count(element)`: Returns the number of occurrences of an element in the list.

```
numbers = [1, 2, 3, 1, 1]
print(numbers.count(1))  # Output: 3
```

- `reverse()`: Reverses the elements of the list in place.

```
fruits.reverse()
print(fruits)  # Output: ['cherry', 'orange', 'apple']
```

- `sort()`: Sorts the list in place (ascending by default).

```
numbers = [5, 2, 9, 1]
numbers.sort()
print(numbers)  # Output: [1, 2, 5, 9]
```

---

6. Nested Lists

Lists can contain other lists, allowing you to create **nested lists**. This can be useful for storing matrix-like data structures.

Example:

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

# Accessing elements in a nested list
print(matrix[0])  # Output: [1, 2, 3] (first row)
print(matrix[1][1])  # Output: 5 (element in the second row, second column)
```

Homework

1. **List Manipulation Exercise**:
    - Create a list of 5 items (strings or numbers).
    - Add a new item to the end of the list and another at the second position.
    - Remove the third item from the list.
    - Print the list after each operation.
2. **Reverse and Sort a List**: Create a list of numbers and:
    - Sort it in descending order.
    - Reverse the sorted list and print it.

Tuples and Sets in Python

1. Tuples in Python

A tuple is a collection of items that is **ordered** and **immutable** (unchangeable). Tuples are similar to lists, but once a tuple is created, you cannot modify it. They are often used to group related data together.

Syntax:

```
my_tuple = (element1, element2, element3, ...)
```

Example:

```
my_tuple = ("apple", "banana", "cherry")
numbers_tuple = (1, 2, 3, 4)
```

Creating a Tuple with One Element:

To create a tuple with only one element, include a trailing comma:

```
single_element_tuple = ("apple",)
```

---

2. Accessing Tuple Elements

You can access elements in a tuple using **indexing**, just like with lists. Tuples also support negative indexing.

Example:

```
fruits = ("apple", "banana", "cherry")
print(fruits[0])  # Output: apple
print(fruits[-1])  # Output: cherry
```

Slicing Tuples:

You can also slice tuples to access a subset of the elements.

```
print(fruits[1:3])  # Output: ('banana', 'cherry')
```

---

3. Tuple Operations

Although tuples are immutable, you can perform various operations with them.

Tuple Concatenation:

You can combine two or more tuples using the `+` operator.

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
combined_tuple = tuple1 + tuple2
print(combined_tuple)  # Output: (1, 2, 3, 4, 5, 6)
```

Tuple Repetition:

You can repeat a tuple multiple times using the `*` operator.

```
repeated_tuple = (1, 2) * 3
print(repeated_tuple)  # Output: (1, 2, 1, 2, 1, 2)
```

Checking Membership:

You can check if an item exists in a tuple using the `in` operator.

```
print("apple" in fruits)  # Output: True
```

---

4. Tuple Methods

Though tuples are immutable, Python provides some built-in methods for working with tuples.

- `count()`: Returns the number of times an element appears in the tuple.

  ```
  my_tuple = (1, 2, 3, 1, 1)
  print(my_tuple.count(1))  # Output: 3
  ```

- `index()`: Returns the index of the first occurrence of an element.

  ```
  my_tuple = ("apple", "banana", "cherry")
  print(my_tuple.index("banana"))  # Output: 1
  ```

---

5. Advantages of Using Tuples

- **Immutable**: This property ensures that tuple data cannot be modified after creation, making them useful for fixed data.
- **Faster than Lists**: Due to immutability, tuples are generally faster than lists.
- **Can Be Used as Keys in Dictionaries**: Since tuples are hashable, they can be used as keys in dictionaries, unlike lists.

---

6. Sets in Python

A set is a collection of **unique** items that is **unordered** and **unindexed**. Sets do not allow duplicate values. Sets are useful for performing operations like union, intersection, and difference.

Syntax:

```
my_set = {element1, element2, element3, ...}
```

Example:

```
fruits_set = {"apple", "banana", "cherry"}
numbers_set = {1, 2, 3, 4, 5}
```

Empty Set:

To create an empty set, use the `set()` function (not `{}`, which creates an empty dictionary).

```
empty_set = set()
```

---

7. Set Operations

Sets support mathematical operations like union, intersection, and difference.

Union:

The union of two sets combines all elements from both sets, removing duplicates.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1 | set2  # Output: {1, 2, 3, 4, 5}
```

Intersection:

The intersection of two sets returns elements that are common to both sets.

```
intersection_set = set1 & set2  # Output: {3}
```

Difference:

The difference between two sets returns elements that are in the first set but not in the second.

```
difference_set = set1 - set2  # Output: {1, 2}
```

Symmetric Difference:

The symmetric difference returns elements that are in either of the sets but not in both.

```
sym_diff_set = set1 ^ set2  # Output: {1, 2, 4, 5}
```

---

8. Set Methods

Sets come with several useful methods for performing common tasks.

- `add()`: Adds an element to the set.

  ```
  fruits_set.add("orange")
  print(fruits_set)  # Output: {'apple', 'banana', 'cherry', 'orange'}
  ```

- `remove()`: Removes a specified element from the set. Raises an error if the element does not exist.

  ```
  fruits_set.remove("banana")
  print(fruits_set)  # Output: {'apple', 'cherry'}
  ```

- `discard()`: Removes a specified element without raising an error if it does not exist.

  ```
  fruits_set.discard("banana")  # No error if "banana" is not in the set
  ```

- `pop()`: Removes a random element from the set.

  ```
  fruits_set.pop()
  ```

- `clear()`: Removes all elements from the set.

  ```
  fruits_set.clear()
  ```

9. Differences Between Lists, Tuples, and Sets

Feature	List	Tuple	Set
Ordering	Ordered	Ordered	Unordered
Mutability	Mutable	Immutable	Mutable
Duplicates	Allows duplicates	Allows duplicates	No duplicates
Indexing	Supports indexing	Supports indexing	No indexing
Operations	List operations	Tuple operations	Set operations
Common Uses	General collection	Fixed data	Unique items

Homework

1. **Tuple Operations**:
    - Create a tuple with 5 elements.
    - Try to modify one of the elements. What happens?
    - Perform slicing on the tuple to extract the second and third elements.
    - Concatenate the tuple with another tuple.
2. **Set Operations**:
    - Create two sets: one with your favorite fruits and another with your friend's favorite fruits.
    - Find the union, intersection, and difference between the two sets.
    - Add a new fruit to your set.
    - Remove a fruit from your set using both `remove()` and `discard()`. What happens when the fruit doesn't exist?
3. **Tuple and Set Comparison**:
    - Create a list of elements and convert it into both a tuple and a set.
    - Print both the tuple and the set.
    - Try to add new elements to the tuple and set. What differences do you observe?

Dictionaries in Python

A dictionary in Python is a collection of **key-value pairs**. Each key in a dictionary is associated with a value, and you can retrieve or manipulate data using the key. Unlike lists and tuples, dictionaries are **unordered** and **mutable** (changeable).

1. Creating a Dictionary

You can create a dictionary using curly braces `{}` or the `dict()` function.

Syntax:

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

Example:

Let's create a dictionary of famous cities in Karnataka and their popular dishes.

```
karnataka_food = {
    "Bengaluru": "Bisi Bele Bath",
    "Mysuru": "Mysore Pak",
    "Mangaluru": "Neer Dosa"
}
```

2. Accessing Dictionary Elements

To access the values stored in a dictionary, you use the key.

Example:

```
print(karnataka_food["Mysuru"])  # Output: Mysore Pak
```

You can also use the `get()` method to access values, which is safer because it doesn't throw an error if the key doesn't exist.

```
print(karnataka_food.get("Mangaluru"))  # Output: Neer Dosa
print(karnataka_food.get("Shivamogga", "Not Found"))  # Output: Not Found
```

3. Adding and Updating Dictionary Elements

You can add new key-value pairs or update existing values in a dictionary.

Adding an Item:

```
karnataka_food["Shivamogga"] = "Kadubu"
print(karnataka_food)
```

Updating an Item:

```
karnataka_food["Bengaluru"] = "Ragi Mudde"
```

4. Removing Elements from a Dictionary

You can remove items from a dictionary using several methods:

- `pop()`: Removes the specified key and returns the associated value.

  ```
  mysuru_food = karnataka_food.pop("Mysuru")
  print(mysuru_food)  # Output: Mysore Pak
  ```

- `del`: Removes the specified key.

  ```
  del karnataka_food["Mangaluru"]
  ```

- `clear()`: Empties the dictionary.

  ```
  karnataka_food.clear()
  ```

5. Dictionary Methods

Here are some common methods available for dictionaries:

- `keys()`: Returns all the keys in the dictionary.

  ```
  print(karnataka_food.keys())  # Output: dict_keys(['Bengaluru', 'Mysuru', 'Mangaluru'])
  ```

- `values()`: Returns all the values in the dictionary.

  ```
  print(karnataka_food.values())  # Output: dict_values(['Bisi Bele Bath', 'Mysore Pak', 'Neer Dosa'])
  ```

- `items()`: Returns key-value pairs as tuples.

  ```
  print(karnataka_food.items())  # Output: dict_items([('Bengaluru', 'Bisi Bele Bath'), ('Mysuru', 'Mysore Pak'), ('Mangaluru', '
  ```

- `update()`: Updates the dictionary with another dictionary or iterable.

  ```
  new_dishes = {"Hubballi": "Girmit"}
  karnataka_food.update(new_dishes)
  ```

6. Dictionary Characteristics

- **Unordered**: Dictionary keys are not stored in any particular order.
- **Mutable**: You can change, add, or remove items.
- **Keys Must Be Immutable**: Keys in a dictionary must be of a data type that is immutable, such as a string, number, or tuple.
- **Unique Keys**: A dictionary cannot have duplicate keys. If you try to add a duplicate key, the latest value will overwrite the previous one.

7. Differences Between Lists, Tuples, Sets, and Dictionaries

Feature	List	Tuple	Set	Dictionary
Ordering	Ordered	Ordered	Unordered	Unordered
Mutability	Mutable	Immutable	Mutable	Mutable
Duplicates	Allows duplicates	Allows duplicates	No duplicates	Keys: No duplicates
Indexing	Supports indexing	Supports indexing	No indexing	Uses keys
Structure	Indexed collection	Indexed collection	Unordered collection	Key-value pairs

---

Homework

1. **Basic Dictionary Operations**:
    - Create a dictionary to store information about 5 cities in Karnataka and their famous dishes.
    - Add a new city and its dish to the dictionary.
    - Update the dish for Bengaluru.
    - Remove one city from the dictionary.
    - Use the `keys()` method to print all city names in the dictionary.
    - Use the `values()` method to print all dishes in the dictionary.
2. **Nested Dictionary Practice** (Simple for now):
    - Create a dictionary to store details of two of your friends, including their names, favorite subject, and favorite food.
    - Access and print the favorite food of one friend.

Conditional Statements in Python: `if`, `elif`, and `else`

In programming, **conditional statements** are used to perform different actions based on different conditions. Python uses `if`, `elif`, and `else` statements to allow your program to make decisions.

1. The `if` Statement

The `if` statement is used to test a condition. If the condition is **True**, the block of code under the `if` statement is executed.

Syntax:

```
if condition:
    # Code block to execute if the condition is True
```

Example:

Let's say you want to check if it's time for dinner (assuming dinner time is 8 PM).

```
time = 20  # 20 represents 8 PM in 24-hour format
if time == 20:
    print("It's time for dinner!")
```

Here, the program checks if the variable `time` is equal to 20 (8 PM). If it's 20, the message `"It's time for dinner!"` is printed.

2. The `else` Statement

The `else` statement provides an alternative block of code to execute when the `if` condition is **False**.

Syntax:

```
if condition:
    # Code block if the condition is True
else:
    # Code block if the condition is False
```

Example:

Let's extend the dinner example by adding an alternative action if it's not 8 PM.

```
time = 18  # 6 PM
if time == 20:
    print("It's time for dinner!")
else:
    print("It's not dinner time yet.")
```

If the condition (`time == 20`) is False (because the time is 6 PM), the program prints `"It's not dinner time yet."`

3. The `elif` Statement

The `elif` (short for "else if") statement checks another condition if the previous `if` or `elif` condition was False. You can have multiple `elif` statements to test various conditions.

Syntax:

```
if condition1:
    # Code block if condition1 is True
elif condition2:
    # Code block if condition2 is True
else:
    # Code block if none of the above conditions are True
```

Example:

Let's create a system to check meal times based on the time of the day:

```
time = 15  # 3 PM

if time == 8:
    print("It's breakfast time!")
elif time == 13:
    print("It's lunch time!")
elif time == 20:
    print("It's dinner time!")
else:
    print("It's not a meal time.")
```

Here, the program checks multiple conditions:

- If the time is 8 AM, it prints `"It's breakfast time!"`.
- If the time is 1 PM, it prints `"It's lunch time!"`.
- If the time is 8 PM, it prints `"It's dinner time!"`.
- If none of these conditions are true, it prints `"It's not a meal time."`

4. Comparison Operators in `if` Statements

You can use **comparison operators** to compare values in `if` statements:

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

Example:

Let's check if someone is eligible to vote in Karnataka (minimum age for voting is 18).

```
age = 19

if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not eligible to vote.")
```

Here, the condition `age >= 18` checks if the age is greater than or equal to 18. If True, it prints that the person is eligible to vote. Otherwise, it prints that they are not eligible.

5. Logical Operators in `if` Statements

You can also use **logical operators** to combine multiple conditions in `if` statements:

- `and`: Returns True if both conditions are True
- `or`: Returns True if at least one condition is True
- `not`: Reverses the result of a condition

Example:

Let's say you want to check if someone is eligible for a student discount. The person must be both under 18 years of age and have a student ID.

```
age = 16
has_student_id = True

if age < 18 and has_student_id:
    print("You are eligible for the student discount!")
else:
    print("You are not eligible for the student discount.")
```

Here, the condition `age < 18 and has_student_id` checks if both conditions are True. If so, the message `"You are eligible for the student discount!"` is printed.

---

6. Example: Checking Bus Ticket Prices

Let's create an example based on ticket prices for a Karnataka KSRTC bus. If the passenger is under 5 years old, the ticket is free. If the passenger is between 5 and 12 years old, they get a child discount. If the passenger is 60 years or older, they get a senior citizen discount. Otherwise, they pay the full fare.

```
age = 65

if age < 5:
    print("Ticket is free.")
elif age <= 12:
    print("You get a child discount.")
elif age >= 60:
    print("You get a senior citizen discount.")
else:
    print("You pay the full fare.")
```

In this example:

- If the passenger is younger than 5 years, the output is `"Ticket is free."`
- If they are 5 to 12 years old, it prints `"You get a child discount."`
- If they are 60 or older, it prints `"You get a senior citizen discount."`
- For all other ages, it prints `"You pay the full fare."`

---

7. Nested `if` Statements

You can also use `if` statements inside other `if` statements. This is called **nesting**.

Example:

Let's say you're planning to visit Mysuru. You want to decide whether to go based on the day of the week and the weather.

```
day = "Saturday"
is_raining = False

if day == "Saturday" or day == "Sunday":
    if not is_raining:
        print("Let's visit Mysuru!")
    else:
        print("It's raining, let's stay home.")
else:
    print("It's a weekday, let's wait for the weekend.")
```

Here, the program first checks if it's a weekend. If it is, it checks the weather. If it's not raining, it prints `"Let's visit Mysuru!"`, otherwise, it prints `"It's raining, let's stay home."` On weekdays, it prints `"It's a weekday, let's wait for the weekend."`

---

8. Indentation in Python

Python uses **indentation** (spaces at the beginning of a line) to define blocks of code. The indented code after an `if`, `elif`, or `else` statement belongs to that condition. Make sure to use consistent indentation to avoid errors.

Example:

```
age = 19

if age >= 18:
    print("You are eligible to vote.")
    print("Remember to bring your voter ID.")
else:
    print("You are not eligible to vote.")
```

In the example above, the two `print()` statements are part of the `if` block because they are indented. Be careful to maintain the correct indentation for your code to run correctly.

Homework

1. **Basic Conditions**:
    - Write a program to check if someone is eligible for a bus pass. If they are below 5 years, the bus pass is free. If they are 60 years or older, they get a senior citizen discount. Otherwise, they pay the full price.
2. **Meal Time Checker**:
    - Create a program that checks the time of day (24-hour format) and prints whether it's time for breakfast, lunch, or dinner.
        - Breakfast: 8 AM
        - Lunch: 1 PM
        - Dinner: 8 PM
        - If none of these times, print "It's not meal time."
3. **Simple Eligibility Check**:
    - Write a program that checks whether a person is eligible for a library membership. If they are under 18, they get a student membership. If they are 60 or older, they get a senior citizen membership. Otherwise, they get a regular membership.

While Loops in Python

A **loop** is a programming structure that repeats a set of instructions as long as a specified condition is True. In Python, the **`while` loop** allows you to repeatedly execute a block of code as long as the condition is True.

1. The Basic Structure of a `while` Loop

The `while` loop repeatedly executes a block of code as long as the condition is `True`.

Syntax:

```
while condition:
    # Code to execute as long as condition is True
```

Example:

Let's print numbers from 1 to 5 using a `while` loop.

```
i = 1
while i <= 5:
    print(i)
    i += 1  # Incrementing i by 1 after each iteration
```

- The loop starts with `i = 1` and checks if `i <= 5`.
- As long as this condition is `True`, it prints the value of `i` and increases it by 1 (`i += 1`).
- The loop ends when `i` becomes 6, as the condition `i <= 5` becomes `False`.

Output:

```
1
2
3
4
5
```

2. Common Example: Counting Sheep

Let's relate this to a common example: Imagine you're counting sheep to fall asleep.

```
sheep_count = 1
while sheep_count <= 10:
    print(f"Sheep {sheep_count}")
    sheep_count += 1
```

This prints `"Sheep 1"`, `"Sheep 2"`, and so on, until `"Sheep 10"`. After that, the loop stops.

3. Avoiding Infinite Loops

A **`while` loop** can run indefinitely if the condition is always `True`. To prevent this, ensure that the condition eventually becomes `False`.

Example of an Infinite Loop:

```
i = 1
while i <= 5:
    print(i)
    # Forgot to update i, so the condition remains True forever!
```

In this case, the loop will keep printing `1` forever because `i` is never incremented, so the condition `i <= 5` will always be `True`.

To avoid this, make sure to **update the variable** that controls the condition within the loop.

4. Using `break` to Exit a `while` Loop

You can use the `break` statement to exit a loop when a certain condition is met.

Example:

Let's stop counting sheep after 5 sheep, even though the condition allows counting up to 10:

```
sheep_count = 1
while sheep_count <= 10:
    print(f"Sheep {sheep_count}")
    if sheep_count == 5:
        print("That's enough counting!")
        break
    sheep_count += 1
```

- The loop stops after `"Sheep 5"` because of the `break` statement, even though the condition was `sheep_count <= 10`.

Output:

```
Sheep 1
Sheep 2
Sheep 3
Sheep 4
Sheep 5
That's enough counting!
```

5. Using `continue` to Skip an Iteration

The `continue` statement is used to skip the current iteration and move on to the next one.

Example:

Let's say you want to skip counting sheep that are number 4:

```
sheep_count = 1
while sheep_count <= 5:
    if sheep_count == 4:
        sheep_count += 1
        continue
    print(f"Sheep {sheep_count}")
    sheep_count += 1
```

Here, when `sheep_count` is 4, the `continue` statement skips printing `"Sheep 4"`, and the loop continues with `sheep_count = 5`.

Output:

```
Sheep 1
Sheep 2
Sheep 3
Sheep 5
```

6. Using `while` Loops for User Input

You can use a `while` loop to repeatedly ask the user for input until they provide valid data.

Example:

Let's ask the user for a PIN until they enter the correct one:

```
pin = ""
correct_pin = "1234"
while pin != correct_pin:
    pin = input("Enter your PIN: ")
    if pin != correct_pin:
        print("Incorrect PIN. Try again.")
print("PIN accepted. You can proceed.")
```

- The loop keeps running until the user enters the correct PIN.
- If the user enters an incorrect PIN, they are prompted to try again.

7. Real-life Example: KSRTC Bus Seats Availability

Let's say you want to simulate a KSRTC bus seat booking system. The bus has 5 available seats. Each time a seat is booked, the available seats decrease.

```
available_seats = 5

while available_seats > 0:
    print(f"{available_seats} seats available.")
    booking = input("Do you want to book a seat? (yes/no): ").lower()

    if booking == "yes":
        available_seats -= 1
        print("Seat booked!")
    else:
        print("No booking made.")

print("All seats are booked!")
```

Here, the loop keeps running until all seats are booked. It checks the available seats and asks the user if they want to book one. The loop stops when there are no more seats available.

Output Example:

```
5 seats available.
Do you want to book a seat? (yes/no): yes
Seat booked!
4 seats available.
Do you want to book a seat? (yes/no): yes
Seat booked!
...
1 seats available.
Do you want to book a seat? (yes/no): yes
Seat booked!
All seats are booked!
```

8. Nested `while` Loops

You can also nest `while` loops inside each other. This can be useful in more complex scenarios, such as checking multiple conditions or dealing with multi-level data.

Example:

Let's simulate a snack machine that allows users to buy snacks as long as both the machine has snacks and the user has money:

```
snacks_available = 3
money = 10

while snacks_available > 0 and money > 0:
    print(f"Snacks available: {snacks_available}. Money: ₹{money}")
    buy = input("Do you want to buy a snack for ₹5? (yes/no): ").lower()

    if buy == "yes" and money >= 5:
        snacks_available -= 1
        money -= 5
        print("Snack purchased!")
    else:
        print("No purchase made.")

print("Either snacks are sold out or you are out of money.")
```

This loop will continue as long as there are snacks available and the user has money. Once one condition is no longer True, the loop stops.

Homework

1. **Basic Counting with `while` Loop**:
    - Write a program that counts from 1 to 10 using a `while` loop.
2. **Odd Numbers Printer**:

- Create a program that prints all odd numbers between 1 and 20 using a `while` loop.

3. **Ticket Booking Simulation**:
   - Write a program that simulates a bus ticket booking system. The bus has 8 seats. Each time a seat is booked, the available seats decrease. When there are no seats left, the loop stops and displays a message saying "All seats are booked."
4. **Countdown Timer**:
   - Write a program that counts down from 10 to 1 using a `while` loop and prints "Happy New Year!" after the countdown is over.

For Loops in Python

In Python, a **`for` loop** is used to iterate over a sequence (like a list, tuple, string, or range) and execute a block of code repeatedly for each element in the sequence.

1. The Basic Structure of a `for` Loop

A **`for` loop** allows you to repeat a block of code a fixed number of times, or once for each element in a sequence.

Syntax:

```
for item in sequence:
    # Code to execute for each item in the sequence
```

Example:

Let's print each name in a list of Kannada cities:

```
cities = ["Bengaluru", "Mysuru", "Hubballi", "Mangaluru"]
for city in cities:
    print(city)
```

Output:

```
Bengaluru
Mysuru
Hubballi
Mangaluru
```

- Here, `cities` is a list, and the `for` loop iterates over each item (city) in that list.

2. Using `range()` with `for` Loops

The `range()` function generates a sequence of numbers, which you can use in a `for` loop when you want to repeat a block of code a specific number of times.

Syntax of `range()`:

```
range(start, stop, step)
```

- `start`: The starting value (inclusive).
- `stop`: The ending value (exclusive).
- `step`: The increment (optional, default is 1).

Example: Counting from 1 to 10

```
for i in range(1, 11):
    print(i)
```

This loop will print the numbers from 1 to 10.

Output:

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

Example: Counting by 2s from 1 to 10

```
for i in range(1, 11, 2):
    print(i)
```

This loop prints only the odd numbers between 1 and 10.

Output:

```
1
3
5
7
9
```

3. Looping Over Strings

You can also loop over each character in a string using a `for` loop.

Example: Printing each character in a string

```
name = "Karnataka"
for letter in name:
    print(letter)
```

Output:

```
K
a
r
n
a
t
a
k
a
```

This loop goes through the string `"Karnataka"` one character at a time.

4. Nested `for` Loops

You can also have **nested `for` loops**, which means a loop inside another loop. This is useful when working with multi-level data, like lists inside lists.

Example: Multiplication Table

Let's print the multiplication table from 1 to 5 using a nested `for` loop.

```
for i in range(1, 6):
    for j in range(1, 6):
        print(f"{i} x {j} = {i * j}")
    print()  # To print an empty line after each table
```

Output:

```
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5

2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10

...

5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
```

Here, the outer loop controls the first number (`i`), and the inner loop controls the second number (`j`). Together, they generate the multiplication table.

5. Using `break` in a `for` Loop

The `break` statement is used to exit a loop early when a certain condition is met.

Example: Stop the loop when you find a specific item

Let's say you are searching for a specific city in a list:

```
cities = ["Bengaluru", "Mysuru", "Hubballi", "Mangaluru"]
for city in cities:
    if city == "Hubballi":
        print(f"Found {city}!")
        break
    print(city)
```

In this case, the loop stops when it finds `"Hubballi"` and prints `"Found Hubballi!"`.

Output:

```
Bengaluru
Mysuru
Found Hubballi!
```

6. Using `continue` in a `for` Loop

The `continue` statement is used to skip the current iteration of the loop and move on to the next one.

Example: Skip a specific item in a list

Let's skip `"Hubballi"` while looping through the cities:

```
cities = ["Bengaluru", "Mysuru", "Hubballi", "Mangaluru"]
for city in cities:
    if city == "Hubballi":
        continue
    print(city)
```

Output:

```
Bengaluru
Mysuru
Mangaluru
```

Here, `"Hubballi"` is skipped, and the loop continues with the next city.

7. Looping Through a List with `enumerate()`

The `enumerate()` function allows you to loop over a sequence and get both the index and the value of each item.

Example: Displaying the index and value of each city

```
cities = ["Bengaluru", "Mysuru", "Hubballi", "Mangaluru"]
for index, city in enumerate(cities):
    print(f"City {index + 1}: {city}")
```

Output:

```
City 1: Bengaluru
City 2: Mysuru
City 3: Hubballi
City 4: Mangaluru
```

8. Using `else` with `for` Loops

You can also use an **`else` clause** with a `for` loop. The code inside the `else` block will execute once the loop finishes, unless the loop is terminated by a `break` statement.

Example:

```
for city in cities:
    print(city)
else:
    print("No more cities!")
```

Output:

```
Bengaluru
Mysuru
Hubballi
Mangaluru
No more cities!
```

In this case, after the loop has finished going through all the cities, it prints `"No more cities!"`.

9. Real-Life Example: Distributing Laddus

Imagine you have 5 laddus to distribute among friends. You can use a `for` loop to give each friend one laddu.

```
laddus = 5
friends = ["Rahul", "Sneha", "Aman", "Priya"]

for friend in friends:
    if laddus > 0:
        print(f"{friend} gets a laddu!")
        laddus -= 1
    else:
        print("No laddus left!")
```

Output:

```
Rahul gets a laddu!
Sneha gets a laddu!
Aman gets a laddu!
Priya gets a laddu!
No laddus left!
```

Here, the loop goes through the list of friends and distributes the laddus one by one.

Homework

1. **Multiples of 3**:
    - Write a `for` loop that prints all multiples of 3 between 1 and 30.
2. **Sum of First 10 Numbers**:
    - Write a program using a `for` loop that calculates the sum of numbers from 1 to 10.
3. **Print Your Name Letter by Letter**:
    - Write a program that takes your name as input and prints each letter of your name using a `for` loop.
4. **Count Vowels in a String**:
    - Write a program that counts how many vowels are in a given string using a `for` loop.

Lists and Dictionaries with For Loops, List Comprehension, and Dictionary Comprehension

In this video, we will learn how to use `for` loops with **Lists** and **Dictionaries**, followed by advanced techniques using **List Comprehension** and **Dictionary Comprehension**. These tools are essential for writing concise and efficient Python code.

---

1. Looping Through Lists

A `for` loop is the most common way to iterate through items in a list.

Example: Sum of all numbers in a list

```
numbers = [10, 20, 30, 40, 50]
total = 0

for num in numbers:
    total += num

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

Output:

```
Total sum: 150
```

Example: Doubling each number in a list

```
numbers = [1, 2, 3, 4, 5]
doubled = []

for num in numbers:
    doubled.append(num * 2)

print("Doubled List:", doubled)
```

Output:

```
Doubled List: [2, 4, 6, 8, 10]
```

Example: Printing Kannada food items

```
foods = ["Dosa", "Idli", "Vada", "Bisibelebath"]

for food in foods:
    print(f"I like {food}")
```

Output:

```
I like Dosa
I like Idli
I like Vada
I like Bisibelebath
```

---

2. Looping Through Dictionaries

You can use `for` loops to iterate over dictionaries by accessing both keys and values.

Example: Iterating over dictionary keys

```
student_marks = {"Anand": 85, "Geetha": 90, "Kumar": 78}

for student in student_marks:
    print(student)
```

Output:

```
Anand
Geetha
Kumar
```

Example: Iterating over dictionary values

```
student_marks = {"Anand": 85, "Geetha": 90, "Kumar": 78}

for marks in student_marks.values():
    print(marks)
```

Output:

```
85
90
78
```

Example: Iterating over both keys and values

```
student_marks = {"Anand": 85, "Geetha": 90, "Kumar": 78}

for student, marks in student_marks.items():
    print(f"{student} scored {marks} marks")
```

Output:

```
Anand scored 85 marks
Geetha scored 90 marks
Kumar scored 78 marks
```

---

3. `for` Loops with `range()`

You can also use `for` loops with the `range()` function to loop through a sequence of numbers.

Example: Adding marks to students using index values

```
students = ["Anand", "Geetha", "Kumar"]
marks = [85, 90, 78]

student_marks = {}

for i in range(len(students)):
    student_marks[students[i]] = marks[i]

print(student_marks)
```

Output:

```
{'Anand': 85, 'Geetha': 90, 'Kumar': 78}
```

---

4. List Comprehension

List comprehension provides a more concise way to create lists by applying an expression to each element in an existing list.

Syntax:

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

Example 1: Squaring numbers in a list

```
numbers = [1, 2, 3, 4, 5]
squares = [num ** 2 for num in numbers]
print(squares)
```

Output:

```
[1, 4, 9, 16, 25]
```

Example 2: Filtering even numbers

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = [num for num in numbers if num % 2 == 0]
print(even_numbers)
```

Output:

```
[2, 4, 6]
```

Example 3: Uppercasing Kannada city names

```
cities = ["Bengaluru", "Mysuru", "Hubballi", "Mangaluru"]
uppercased_cities = [city.upper() for city in cities]
print(uppercased_cities)
```

Output:

```
['BENGALURU', 'MYSURU', 'HUBBALLI', 'MANGALURU']
```

---

5. Dictionary Comprehension

Similar to list comprehension, dictionary comprehension provides a concise way to create dictionaries.

Syntax:

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

Example 1: Creating a dictionary of squares

```
numbers = [1, 2, 3, 4, 5]
squares_dict = {num: num ** 2 for num in numbers}
print(squares_dict)
```

Output:

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Example 2: Converting a list of names to a dictionary of name lengths

```
names = ["Anand", "Geetha", "Kumar"]
name_lengths = {name: len(name) for name in names}
print(name_lengths)
```

Output:

```
{'Anand': 5, 'Geetha': 6, 'Kumar': 5}
```

Example 3: Filtering cities with population above 10 lakhs (Localized Example)

```
city_population = {
    "Bengaluru": 84,
    "Mysuru": 11,
    "Hubballi": 9,
    "Mangaluru": 5
}
large_cities = {city: population for city, population in city_population.items() if population > 10}
print(large_cities)
```

Output:

```
{'Bengaluru': 84, 'Mysuru': 11}
```

---

6. Splitting Strings to Create Lists

In Python, the `split()` method allows you to break a string into a list of words or elements based on a specified separator (like space, comma, etc.). This is very useful when you have data in string format and want to convert it into a list.

Syntax:

```
string.split(separator, maxsplit)
```

- **separator** (optional): The character or substring on which to split the string (default is space).
- **maxsplit** (optional): The maximum number of splits (default is unlimited).

Example 1: Splitting a sentence into words

```
sentence = "I love coding in Python"
words = sentence.split()
print(words)
```

Output:

```
['I', 'love', 'coding', 'in', 'Python']
```

Here, the default separator (space) is used to split the string into individual words.

Example 2: Splitting a string with commas

```
data = "apple,banana,mango"
fruits = data.split(",")
print(fruits)
```

Output:

```
['apple', 'banana', 'mango']
```

Here, the comma (`,`) is used as the separator to split the string into different fruit names.

Example 3: Limiting the number of splits

```
sentence = "Python is fun to learn"
words = sentence.split(" ", 2)
print(words)
```

Output:

```
['Python', 'is', 'fun to learn']
```

In this example, the string is split only twice. The rest of the string remains as the final element.

Practical Application

You can use the `split()` method when reading data from a text file or processing input from a user where the data is separated by spaces, commas, or any other delimiters. Once you split the data, you can work with it as a list, applying any list manipulation techniques you've learned.

Homework

1. **List Manipulation**:
   - Create a list of Kannada foods. Use list comprehension to create a new list where each food name is in uppercase.
2. **Sum of Prices**:
   - Create a dictionary of 5 items with their prices. Write a program that calculates the total price of all items using a `for` loop.
3. **List of Squares**:
   - Create a list of numbers from 1 to 10. Use list comprehension to generate a list of their squares.
4. **Student Data Task**:
   - Create a list of 3 dictionaries, where each dictionary contains the name, age, and marks of a student. Loop through the list and print each student's information.
5. **Dictionary Comprehension**:
   - Create a dictionary where the keys are Kannada cities, and the values are their populations. Use dictionary comprehension to filter out cities with populations below 10 lakhs.
6. **Nested List Challenge**: Write a Python program that takes a list of lists (a 2D list) as input and:
   - Prints the entire matrix row by row.
   - Prints the sum of each row in the matrix.

   Example:

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

---

1. Basics of Functions

A function is a reusable block of code that performs a specific task when called. Functions are useful to organize code, make it reusable, and reduce redundancy.

---

2. Defining a Function

You define a function using the `def` keyword followed by the function name, parentheses, and a colon `:`.

Syntax:

```
def function_name(parameters):
    # Block of code
```

Example: Basic function to greet a user

```
def greet():
    print("Hello! Welcome to the Python course.")
greet()
```

Output:

```
Hello! Welcome to the Python course.
```

---

3. Function Parameters

Parameters are variables used to pass data into a function.

Example: Function with a parameter

```
def greet_user(name):
    print(f"Hello, {name}! Welcome to the Python course.")

greet_user("Anand")
```

Output:

```
Hello, Anand! Welcome to the Python course.
```

---

4. Returning Values from a Function

A function can return a value using the `return` keyword, which allows the output of the function to be reused elsewhere.

Example: Function that adds two numbers and returns the result

```
def add_numbers(a, b):
    return a + b

result = add_numbers(10, 20)
print("The sum is:", result)
```

Output:

```
The sum is: 30
```

---

5. Default Parameter Values

You can define a default value for a parameter, which is used if no argument is passed when the function is called.

Example: Function with a default parameter

```
def greet(name="Student"):
    print(f"Hello, {name}! Welcome to the Python course.")

greet()  # Uses default value "Student"
greet("Geetha")  # Uses passed value "Geetha"
```

Output:

```
Hello, Student! Welcome to the Python course.
Hello, Geetha! Welcome to the Python course.
```

Here are the sections for **Nested Functions** and **Local/Global Variables**:

---

6. Local and Global Variables

- **Local Variables** are defined inside a function and are only accessible within that function.
- **Global Variables** are defined outside all functions and are accessible from anywhere in the code.

Example: Local vs Global variables

```
name = "Global Name"

def greet():
    name = "Local Name"
    print(name)

greet()  # Prints local variable
print(name)  # Prints global variable
```

Output:

```
Local Name
Global Name
```

In this example, the local variable `name` inside the function does not affect the global variable `name`.

Homework:

1. **Greet Function**: Write a function `greet()` that takes no arguments and prints a greeting message.
2. **Parameterized Greet**: Write a function `greet_user()` that takes a name as input and prints a custom greeting.
3. **Sum Function**: Write a function `add_numbers(a, b)` that returns the sum of two numbers. Call this function with different values.

Functions - Advanced Concepts

In this part, we'll explore more advanced function concepts, including recursion, lambda functions, and variable-length arguments.

---

1. Keyword Arguments

With keyword arguments, you can pass values to a function by specifying the parameter names.

Example:

```
def display_info(name, age):
    print(f"Name: {name}, Age: {age}")

display_info(age=25, name="Kumar")
```

Output:

```
Name: Kumar, Age: 25
```

---

2. Variable-Length Arguments

You can use `*args` and `**kwargs` to accept a variable number of arguments in a function.

Example: Using `*args`

```
def total_sum(*numbers):
    result = 0
    for num in numbers:
        result += num
    return result

print(total_sum(1, 2, 3, 4))
```

Output:

```
10
```

Example: Using `**kwargs`

```
def student_info(**details):
    for key, value in details.items():
        print(f"{key}: {value}")

student_info(name="Anand", age=22, course="Python")
```

Output:

```
name: Anand
age: 22
course: Python
```

---

3. Lambda Functions

A **lambda function** is a small anonymous function that can take any number of arguments but has only one expression.

Syntax:

```
lambda arguments: expression
```

Example: Lambda function to double a number

```
double = lambda x: x * 2
print(double(5))
```

Output:

```
10
```

---

4. Recursion

Recursion occurs when a function calls itself. It's used to solve problems that can be broken down into smaller, similar problems.

Example: Recursive function to calculate factorial

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))
```

Output:

```
120
```

Here are the sections for **Nested Functions** and **Local/Global Variables**:

---

5. Nested Functions

A **nested function** is a function defined inside another function. The inner function is only accessible within the outer function, allowing for more modular and controlled code execution.

Example: Nested function in Python

```
def outer_function(name):
    def inner_function():
        print(f"Hello, {name}!")
    inner_function()

outer_function("Anand")
```

Output:

```
Hello, Anand!
```

In this example, the `inner_function()` is called within `outer_function()` and uses the `name` parameter of the outer function.

Homework:

1. **Lambda Function**: Write a lambda function that multiplies two numbers.
2. **Recursive Function**: Write a recursive function that calculates the sum of the first `n` numbers.
3. **Variable-Length Arguments**: Write a function that accepts any number of arguments and returns their average.
