Lists in Python

In Python, a list is a versatile and mutable data structure that can store a collection of items.

Lists are ordered, indexed, and can contain elements of different data types.

They are defined by enclosing comma-separated values within square brackets [] .

Here's a detailed explanation of lists in Python with examples:

Lists in Python

Creating Lists:

You can create a list in Python using square brackets [ ] and separating elements with commas.

```
# Creating a list of integers
numbers = [1, 2, 3, 4, 5]

# Creating a list of strings
fruits = ["apple", "banana", "orange"]

# Creating a list with mixed data types
mixed_list = [1, "apple", True, 3.14]
```

Lists in Python

Accessing Elements:

Elements in a list are accessed using indices. Python uses 0-based indexing, meaning the first element is at index 0.

```
# Accessing elements by index
print(numbers[0])   # Output: 1
print(fruits[1])    # Output: "banana"

# Negative indexing (accessing elements from the end)
print(numbers[-1])  # Output: 5 (last element)
```

Lists in Python

Slicing Lists:

You can extract a portion of a list using slicing. Slicing syntax is list[start:end:step], where start is inclusive and end is exclusive.

```
# Slicing a list
print(numbers[1:4])  # Output: [2, 3, 4]

# Slicing with step
print(numbers[::2])  # Output: [1, 3, 5]
```

Lists in Python

Modifying Lists:

Lists are mutable, meaning you can change their elements after creation.

```
# Modifying elements
fruits[0] = "grape"
print(fruits)  # Output: ["grape", "banana", "orange"]

# Appending elements
fruits.append("kiwi")
print(fruits)  # Output: ["grape", "banana", "orange", "kiwi"]

# Removing elements
fruits.remove("banana")
print(fruits)  # Output: ["grape", "orange", "kiwi"]
```

Lists in Python

List Operations:

Lists support various operations such as concatenation (+), repetition (*), length (len()), membership (in), and iteration.

```
# Concatenation
combined_list = numbers + fruits
print(combined_list)  # Output: [1, 2, 3, 4, 5, "grape", "orange", "kiwi"]

# Repetition
repeated_list = fruits * 2
print(repeated_list)  # Output: ["grape", "orange", "kiwi", "grape", "orange", "kiwi"

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

# Membership
print("apple" in fruits)  # Output: False
```

Lists in Python

List Methods:

Python provides several built-in methods to manipulate lists efficiently.

```
# Sorting
numbers.sort()
print(numbers)  # Output: [1, 2, 3, 4, 5]

# Reversing
numbers.reverse()
print(numbers)  # Output: [5, 4, 3, 2, 1]

# Counting occurrences
print(fruits.count("orange"))  # Output: 1

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

Lists in Python

Nested Lists:

Lists can contain other lists, allowing for the creation of nested data structures.

```
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(nested_list[0][1])  # Output: 2
```

List Comprehensions:

List comprehensions provide a concise way to create lists based on existing lists.

```
# Squaring numbers
squared_numbers = [x ** 2 for x in numbers]
print(squared_numbers)  # Output: [25, 16, 9, 4, 1]
```

Lists in Python

Exercise :

Write a program to create sum of elements in a list

Write a program to create two lists and use concatenation

Write a program to reverse elements in a list

Module 3 – Flow Control

Python Operators

Operators in Python are symbols or keywords that perform operations on operands. Here are the main types of operators in Python:

1. **Arithmetic Operators:**
   - Perform basic arithmetic operations.

python Copy code

```python
a = 5
b = 2

print(a + b)  # Addition
print(a - b)  # Subtraction
print(a * b)  # Multiplication
print(a / b)  # Division
print(a % b)  # Modulus
print(a ** b) # Exponentiation
print(a // b) # Floor Division
```

Python Operators

2. Comparison Operators:

- Compare values and return Boolean results.

```python
x = 10
y = 20

print(x == y)  # Equal to
print(x != y)  # Not equal to
print(x > y)   # Greater than
print(x < y)   # Less than
print(x >= y)  # Greater than or equal to
print(x <= y)  # Less than or equal to
```

Python Operators

3. Logical Operators:

- Perform logical operations on Boolean values.

python Copy code

```python
a = True
b = False

print(a and b)  # Logical AND
print(a or b)   # Logical OR
print(not a)    # Logical NOT
```

Python Operators

4. Assignment Operators:

- Assign values to variables.

python Copy code

```python
x = 10
y = 5

x += y  # Equivalent to x = x + y
x -= y  # Equivalent to x = x - y
x *= y  # Equivalent to x = x * y
x /= y  # Equivalent to x = x / y
```

Python Operators

5. Bitwise Operators:

- Perform bitwise operations on integers.

python Copy code

```python
a = 5
b = 3

print(a & b)  # Bitwise AND
print(a | b)  # Bitwise OR
print(a ^ b)  # Bitwise XOR
print(~a)     # Bitwise NOT
print(a << 1)  # Left shift
print(a >> 1)  # Right shift
```

Python Operators

6. Identity Operators:

- Compare the memory addresses of two objects.

python Copy code

```python
x = [1, 2, 3]
y = [1, 2, 3]

print(x is y)     # Identity (False)
print(x is not y) # Not identity (True)
```

Conditional control Statements

Conditional statements in Python are used to make decisions in the code based on certain conditions.

The primary conditional statement in Python is the ‘**if’** statement.

Here's an explanation of the key components:

‘if’ statements

Syntax:

- The basic syntax of an **if** statement is as follows:

```
if condition:
    # code to be executed if the condition is True
```

Example:

```
x = 10

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

- In this example, the condition **x > 5** is checked.
- If it evaluates to **True**, the indented block of code beneath it (in this case, the **print** statement) will be executed.

‘else’ statements

You can extend the **if** statement with an **else** statement to specify what should happen if the condition is **False**.

The **else** block is executed when the **if** condition is not satisfied.

```
x = 3

if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
```

‘elif’ statements

Sometimes, you might want to check multiple conditions.

The **elif** (short for "else if") statement allows you to check additional conditions if the previous ones are not met.

```
x = 5

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

Here, the program will print the appropriate message based on the value of **x**.

Nested ‘if’ statements

You can also nest **if** statements within each other. This involves placing one **if** statement inside another.

Be cautious with indentation to ensure proper code structure.

```
x = 10
y = 5

if x > 5:
    if y > 3:
        print("Both x and y are greater than their respective threshold
```

In this example, the inner **if** statement is only checked if the outer **if** condition (**x > 5**) is **True**.

Loop control Statements

Loop control statements in Python allow you to alter the normal execution flow within loops.

The two primary loop control statements are **break** and **continue**.

‘break’ statement

The **break** statement is used to exit a loop prematurely.

When a **break** statement is encountered within a loop, the loop is immediately terminated, and the program continues with the next statement after the loop.

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

In this example, the loop prints values from 0 to 2. When **i** becomes 3, the **break** statement is encountered, and the loop is terminated.

‘continue’ statement

The **continue** statement is used to skip the rest of the code inside a loop for the current iteration and proceed to the next iteration.

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

This loop prints values from 0 to 4, but when **i** is 2, the **continue** statement is encountered, skipping the rest of the loop body for that iteration.

Looping Statements

Looping statements in Python are used to repeatedly execute a block of code as long as a certain condition is true.

There are two main types of looping statements in Python:

for loops

and

while loops.

‘for’ Loops

The **for** loop is used to iterate over a sequence (such as a list, tuple, string, or range) and execute a block of code for each item in the sequence.

```
fruits = ["apple", "banana", "orange"]

for fruit in fruits:
    print(fruit)
```

In this example, the **for** loop iterates over the **fruits** list, and the **print** statement is executed for each fruit in the list.

‘while’ Loops

The **while** loop repeatedly executes a block of code as long as a specified condition is true. The loop continues until the condition becomes false.

```
count = 0

while count < 5:
    print(count)
    count += 1
```

Here, the **while** loop prints the value of **count** and increments it by 1 in each iteration until **count** becomes 5.

Nested Loops

You can also have loops inside other loops. This is called nested looping. For example, a **for** loop inside another **for** loop.

```
for i in range(3):
    for j in range(2):
        print(i, j)
```

This nested loop prints pairs of values, where **i** ranges from 0 to 2 and **j** ranges from 0 to 1.

Module 4 –
Functions in Python

User-Defined Functions in Python

Functions in Python are blocks of reusable code that perform a specific task. They help in organizing code, making it more modular, and promoting reusability.

In Python, functions are defined using the **def** keyword, followed by the function name, parameters in parentheses, a colon, and then the function body.

Here are some sub-topics related to Python functions:

Function Definition

1. **Function Definition:**

Syntax: **def function_name(parameters):**

Example:

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

Function Parameters

2. **Function Parameters:**

Functions can take parameters (inputs) to receive values that can be used within the function.

Example:

```
def add_numbers(x, y):
    return x + y
```

Return Statement

3. **Return Statement:**

The **return** statement is used to exit a function and return a value to the caller.

Example:

```
def square(x):
    return x * x
```

Recursion

A function calling itself is known as recursion. It can be an alternative to traditional iterative solutions.

Example:

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

Python Built-In Functions

Python provides a rich set of built-in functions that are readily available for use without the need for additional imports.

These functions cover a wide range of operations and are integral to various programming tasks.

Here are some common categories of built-in functions:

Python Built-In Functions

Mathematical Functions:

abs(): Returns the absolute value of a number.

max(): Returns the largest item in an iterable or the largest of two or more arguments.

min(): Returns the smallest item in an iterable or the smallest of two or more arguments.

pow(): Returns x to the power of y.

round(): Rounds a number to the nearest integer or specified number of decimals.

Python Built-In Functions

Type Conversion Functions:

- **int()**, **float()**, **str()**, **bool()**: Convert values to integers, floating-point numbers, strings, and boolean values, respectively.

- **list()**, **tuple()**, **set()**: Convert iterable objects to lists, tuples, and sets.

Python Built-In Functions

String Functions:

- **len()**: Returns the length of an object (e.g., string, list, tuple).
- **str()**: Converts an object to a string.
- **format()**: Formats a string using placeholders.
- **capitalize()**, **lower()**, **upper()**: Change the case of letters in a string.
- **split()**: Splits a string into a list based on a specified delimiter.

Python Built-In Functions

List and Tuple Functions:

- **len()**: Returns the number of elements in a list or tuple.
- **sorted()**: Returns a new sorted list from the elements of an iterable.
- **sum()**: Returns the sum of all elements in an iterable.
- **max()**, **min()**: Returns the maximum or minimum element in an iterable.

Python Built-In Functions

Set Functions:

- **len()**: Returns the number of elements in a set.
- **add()**: Adds an element to a set.
- **remove()**, **discard()**: Removes an element from a set (raises an error or ignores if the element is not present).
- **union()**, **intersection()**, **difference()**: Set operations.

Python Built-In Functions

Dictionary Functions:

- **len()**: Returns the number of key-value pairs in a dictionary.
- **keys()**, **values()**, **items()**: Return lists of keys, values, or key-value pairs.
- **get()**: Returns the value for a specified key or a default value if the key is not present.

‘range’ Function

The **range()** function in Python is used to generate a sequence of numbers.

It can be used in various ways, and its general syntax is:

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

- **start**: Optional. The starting value of the sequence (default is 0).
- **stop**: The end value of the sequence (not included).
- **step**: Optional. The step size between numbers (default is 1).

‘range’ Function

Examples:

1. Generating a sequence of numbers:

```
for i in range(5):
    print(i)
# Output: 0, 1, 2, 3, 4
```

2. Specifying start and end values:

```
for i in range(2, 8):
    print(i)
# Output: 2, 3, 4, 5, 6, 7
```

‘range’ Function

3. Adding a step size

```
for i in range(1, 10, 2):
    print(i)
# Output: 1, 3, 5, 7, 9
```

The **range()** function is commonly used in **for** loops to iterate over a sequence of numbers. It is memory-efficient because it generates values on-the-fly rather than creating a list in memory.