Day 2: Python Fundamentals

- String Operations: Immutability, accessing, slicing, and methods
- Conditional Statements: Utilizing `if`, `elif`, and `else`
- Logical and Membership Operators: Simplifying conditions
- String Manipulation: Uncovering hidden messages, case conversion, trimming
- List Manipulation Techniques: Mutability, combining, slicing, sorting
- `.split()` Method: Dividing strings into lists, handling new lines
- Unscrambling Sentences: Correcting scrambled messages step-by-step
- Loops in Python: `while` and `for` loops, list comprehension

String Literals in Python

Immutable sequences of characters that cannot be changed once created.

Accessing Characters

```
quote = "I love python"
print(quote[0])  # Output: I
print(quote[2])  # Output: l
```

Accessing Characters

```
quote = "I love python"
# Demonstrating character access by index
print(quote[0])  # Output: I
print(quote[2])  # Output: l
```

Immutability of Strings

```
quote = "I love python"
# quote[2] = 'x' # Uncommenting this will raise an error
```

Immutability of Strings

```
quote = "I love python"
# Python strings are immutable
# Attempting to directly modify them will raise an error
```

String Methods

```
quote = "I love python"
print(quote.upper())  # Output: I LOVE PYTHON
print(quote)          # Output: I love python
```

String Methods

```
quote = "I love python"
# Using string methods does not change the original string
upper_quote = quote.upper()
print(upper_quote)  # Output: I LOVE PYTHON
 print(quote)        # Output: I love python
```

Slicing Operator

```python
quote = "I love python"
print(quote[2:6])  # Output: love
print(quote[7:])   # Output: python
```

Slicing Operator

```
quote = "I love python"
# Using slicing to extract parts of the string
slice1 = quote[2:6]  # love
slice2 = quote[7:]   # python
print(slice1)
print(slice2)
```

Advanced Slicing

```
quote = "I love python"
print(quote[-1])    # Output: n
print(quote[-6:])   # Output: python
print(quote[::3])   # Output: Ilphn
```

Advanced Slicing

```python
quote = "I love python"
# Exploring advanced slicing features
reverse_first_char = quote[-1]
substring = quote[-6:]
skip_characters = quote[::3]
print(reverse_first_char)  # n
print(substring)           # python
print(skip_characters)     # Ilphn
```

Reversing a String

```
quote = "I love python"
print(quote[::-1])  # Output: nohtyp evol I
```

Reversing a String

```
quote = "I love python"
# Reversing the string using slicing
reversed_quote = quote[::-1]
print(reversed_quote)  # Output: nohtyp evol I
```

Summary and Conclusion

- Python strings are powerful and flexible.
- Operations like slicing and accessing individual characters are straightforward.
- Despite their immutability, strings in Python can be manipulated using various methods to create new string values.

Python String Methods

Exploring the versatility and power of string manipulation in Python.

Basic String Operations

```
msg = "Hi, all"
print(msg.upper())  # HI, ALL
print(msg.lower())  # hi, all
print(msg.title())  # Hi, All
print(msg.capitalize())  # Hi, all
```

Trimming Whitespace

```
quote = "      Dream... you sleep"
trimmed_quote = quote.strip()
print(trimmed_quote) # Dream... you sleep
```

.strip() method removes leading and trailing whitespace, making it useful for cleaning up strings.

Stripping Specific Characters

```
quote1 = "---Dream... sleep---"
print(quote1.strip("-"))  # Dream... you sleep
print(quote1.lstrip("-")) # Dream... you sleep---
print(quote1.rstrip("-")) # ---Dream... you sleep
```

Using .strip(), .lstrip(), and .rstrip() to remove specific characters from the start, end, or both sides of a string.

Finding Substrings

```
quote3 = "Dream... you sleep"
print(quote3.find('something'))  # 20
print(quote3.find('Dream'))  # 0
print(quote3.find('**'))  # -1
```

The .find() method is used to locate substrings within a string, returning the index of the first occurrence or -1 if not found.

Immutable Strings and Replacement

```
quote3 = "Dream... you sleep"
new_quote = quote3.replace("Dream", "💤")
print(new_quote) # 💤... you sleep
print(quote3)  # Original string is unchanged
```

Highlighting string immutability with .replace(), showing how original strings remain unchanged after modifications.

Counting, Validating, and Length

```
quote3 = "Dream... you sleep"
print(quote3.count("Dream"))  # 2
print(quote3.startswith("Dream"))  # True
print(quote3.endswith("sleep"))  # True
bage_no = "45445"
print(bage_no.isdigit())  # True
name = "ark"
print(name.islower())  # True
print(len(name))  # 3
```

Conditionals in Python

Understanding how to make decisions in your code with `if`, `elif`, and `else` statements.

Basic Comparison

```python
# Example of a simple comparison
number1 = 5
number2 = 10

if number1 > number2:
    print("number1 is greater")
else:
    print("number2 is greater")
```

Basic Comparison

```
number1 = 5
number2 = 10

# Comparing two numbers and printing the result
if number1 > number2:
    print("number1 is greater")
else:
    print("number2 is greater")
```

Handling Equal Values

```
# Checking for equality
number1 = 10
number2 = 10

if number1 > number2:
    print("number1 is greater")
elif number1 == number2:
    print("Both numbers are equal")
else:
    print("number2 is greater")
```

Handling Equal Values

```
number1 = 10
number2 = 10

# Using elif to handle equal values
if number1 > number2:
    print("number1 is greater")
elif number1 == number2:
    print("Both numbers are equal")
else:
    print("number2 is greater")
```

Applying Conditionals to User Input

```
person1 = input('Please give your name: ')
height1 = int(input("Provide your height in cm: "))
person2 = input('Please give your name: ')
height2 = int(input("Provide your height in cm: "))

if height1 > height2:
    print(f'{person1} is taller')
elif height1 == height2:
    print(f'{person1} and {person2} have the same height')
else:
    print(f'{person2} is taller')
```

Applying Conditionals to User Input

```
person1 = input('Please give your name: ')
height1 = int(input("Provide your height in cm: "))
person2 = input('Please give your name: ')
height2 = int(input("Provide your height in cm: "))

# Implementing conditional logic based on user input
if height1 > height2:
    print(f'{person1} is taller')
elif height1 == height2:
    print(f'{person1} and {person2} have the same height')
else:
    print(f'{person2} is taller')
```

Understanding elif and else

```
# Demonstration of elif and else
choice = input("Choose one option: A, B, or C: ")

if choice == 'A':
    print("You chose A.")
elif choice == 'B':
    print("You chose B.")
else:
    print("You chose C.")
```

Understanding elif and else

```
choice = input("Choose one option: A, B, or C: ")

# Explaining the use of elif for multiple conditions and else for the final default condition
if choice == 'A':
    print("You chose A.")
elif choice == 'B':
    print("You chose B.")
else:
    print("You chose C.")
```

Summary and Conclusion

- Python's `if`, `elif`, and `else` statements provide a powerful way to control the flow of your program based on conditions.
- `elif` allows for multiple conditions to be checked sequentially
- `else` serves as a catch-all for any cases not covered by preceding `if` or `elif` statements.

Ice Cream Stock Checker

Learning to use conditional statements and operators in Python for practical problem-solving.

```python
# Initial approach with multiple if/elif statements
fav_flavour = input("Please tell me your flavour: ")
stock1 = "vanilla"
stock2 = "lime"
stock3 = "chocolate"

if fav_flavour == stock1:
    print("Yes, we do have it")
elif fav_flavour == stock2:
    print("Yes, we do have it")
elif fav_flavour == stock3:
    print("Yes, we do have it")
else:
    print("No, we ran out of stock")
```

```python
# Simplified approach using logical OR
fav_flavour = input("Please tell me your flavour: ")

if fav_flavour == "vanilla" or fav_flavour == "lime" or fav_flavour == "chocolate":
    print("Yes, we do have it")
else:
    print("No, we ran out of stock")
```

```python
shop_stock = "vanilla, lime, chocolate"
fav_flavour = input("Please tell me your flavour: ")

# Using the logical OR to simplify the condition
if fav_flavour in shop_stock:
    print("Yes, we do have it")
else:
    print("No, we ran out of stock")
```


Improving with Membership Operator (in)

```python
# Applying the ternary operator for compactness
shop_stock = "vanilla, lime, chocolate"
fav_flavour = input("Please tell me your flavour: ")

result = "Yes, we do have it" if fav_flavour in shop_stock else "No, we ran out of stock"
print(result)
```

Operators in Python

- **Unary Operators**: Operate on a single operand (e.g., `not`, `~` for bitwise NOT).
- **Binary Operators**: Perform operations on two operands (e.g., arithmetic, comparison, logical).
- **Ternary Operator**: A concise way to perform an if-else in a single line.

Summary and Conclusion

- Python offers various ways to handle conditional logic, from basic `if` statements to using operators for more concise code.
- The membership operator (`in`) simplifies checking for the presence of a value in a collection.
- The ternary operator provides a compact way to write simple conditional assignments.

Secret Code Extractor

Demonstrating string manipulation techniques in Python to uncover hidden messages.

```python
# Extracting the secret code after the 🔑 emoji
message = "    🚨🔍🚦🔑secret_code✌️"
code = "SECRET_CODE✌️"

key_idx = message.find('🔑')
secret_msg = message[(key_idx + 1):]
output = secret_msg.upper()
print(output)

if (output == code):
  print("You are an hacker 🎊")
else:
  print("Try again")
```

```python
# Corrected approach to handle when no 🔑 is present
message = "    🚨🔍📱secret_code✌️"
code = "SECRET_CODE✌🏿"

key_idx = message.find('🔑')

if key_idx != -1:
  secret_msg = message[(key_idx + 1):]
  output = secret_msg.upper()
  print(output)

  if (output == code):
    print("You are an hacker 🎊")
  else:
    print("Try again")
else:
  print('No, secret is present')
```

```python
# Adjusted for correct conditional check in Task 2
message = "    🚨🔍🚦🔑****secret_code✌️(((("
code = "SECRET_CODE✌🏾"

key_idx = message.find('🔑')

if key_idx != -1:
  secret_msg = message[(key_idx + 1):]
  output = secret_msg.upper().strip('(').strip('*')
  print(output)

  if (output == code):
    print("You are an hacker 🎊")
  else:
    print("Try again")
else:
  print('No. secret is present')
```

Task 3: Cleaning the Secret Code

```
# Updated to reflect the corrected logical flow from Task 2
message = "    🚨🔍🧵🔑****secret_code✌️(((((("
code = "SECRET_CODE✌🏾"

key_idx = message.find('🔑')

if key_idx != -1:
    output = message[(key_idx + 1):].upper().strip('(').strip('*')
    print(output)

    if (output == code):
        print("You are an hacker 🖇️")
    else:
        print("Try again")
else:
```

Dot Chaining for Efficiency

String Manipulation in Python

- **Finding Substrings**: Using `.find()` to locate specific characters or sequences.
- **Case Conversion**: `.upper()` to convert to uppercase for matching or processing.
- **Trimming Characters**: Using `.strip()` to remove unwanted characters from the start and end of strings.
- **Dot Chaining**: Combining multiple string operations in a single expression for efficiency.

Advanced List Manipulation Techniques in Python

Deep dive into Python lists: exploring their mutable nature, operation categories, and best practices for efficient data handling.

Understanding List Mutability

```python
# Lists in Python are mutable, allowing for dynamic modifications
marks = [98, 75, 40, 80, 90]
print("Original list:", marks)

# Removing elements demonstrates mutability
marks.pop()
print("List after .pop():", marks)

# Modifying an element by index
marks[2] = 85  # Changing 40 to 85
print("List after modifying an element:", marks)
```

Adding and Removing Elements

```
# Adding and removing elements
subjects = ['Maths', 'Science']
subjects.append('English')  # Adding an element
print("After append:", subjects)

subjects.remove('Science')  # Removing a specific element
print("After remove:", subjects)
```

Using .append() and .remove()

Adding and Removing Elements

```
# Inserting at a specific index and popping
marks = [98, 75, 85]
marks.insert(1, 88)  # Insert 88 at index 1
print("After insert:", marks)

popped_mark = marks.pop(2)  # Pop element at index 2
print("After pop:", marks, "| Popped element:", popped_mark)
```

Combining, Duplicating, and Slicing Lists

```
# Combining two lists
list1 = [1, 2, 3]
list2 = [4, 5, 6]
combined_list = list1 + list2
print("Combined list:", combined_list)
```

Combining, Duplicating, and Slicing Lists

```
# Duplicating lists
duplicated_list = ['a', 'b'] * 3
 print("Duplicated list:", duplicated_list)
```

Combining, Duplicating, and Slicing Lists

```
# Slicing for copying and subsetting
marks = [98, 75, 85, 90, 95]
subset_marks = marks[1:4]  # Get subset from index 1 to 3
print("Subset of marks:", subset_marks)
```

Copying Lists and Understanding Memory Management

```python
# Creating a shallow copy of a list
original_list = [1, 2, 3]
shallow_copy = original_list[:]
shallow_copy.append(4)
print("Original list:", original_list)
print("Shallow copy with new element:", shallow_copy)
```

Copying Lists and Understanding Memory Management

```
# Demonstrating different memory addresses for copied lists
print("Memory address of original list:", id(original_list))
print("Memory address of shallow copy:", id(shallow_copy))
```

Sorting and Joining Lists

```python
# Sorting lists in ascending and descending order
numbers = [3, 1, 4, 1, 5, 9, 2, 6]
numbers.sort()  # Sorts in place
print("Sorted in ascending:", numbers)

numbers.sort(reverse=True)
print("Sorted in descending:", numbers)
```

Sorting and Joining Lists

```
# Joining elements of a list into a string
subjects = ['Maths', 'Science', 'English']
subjects_str = ", ".join(subjects)
print("Subjects as a string:", subjects_str)
```

Mastering the `.split()` Method in Python

Understanding how to divide strings into lists using `.split()` for effective data manipulation and processing.

Exploring .split(): From Basics to Advanced Usage

```python
# Splitting a string into a list by spaces (default behavior)
sentence = "Learning Python is fun"
words = sentence.split()
print(words)
# Output: ['Learning', 'Python', 'is', 'fun']
# Demonstrates basic string splitting into a list of words.
```

Basic Usage of .split()

Exploring .split(): From Basics to Advanced Usage

```
# Using a custom separator to split a string
data = "name:John;age:30;city:New York"
details = data.split(';')
print(details)
# Output: ['name:John', 'age:30', 'city:New York']
 # Illustrates splitting a string using a custom separator ';'.
```

Custom Separator

Exploring .split(): From Basics to Advanced Usage

```
# Limiting the number of splits
limited_split = "apple,banana,cherry,dragonfruit".split(',', 2)
print(limited_split)
# Output: ['apple', 'banana', 'cherry,dragonfruit']
# Shows how to limit the number of splits in the result.
```

Exploring .split(): From Basics to Advanced Usage

```python
# Splitting a multi-line string
multi_line_text = """Python
JavaScript
HTML
CSS"""
languages = multi_line_text.split('\n')
print(languages)
# Output: ['Python', 'JavaScript', 'HTML', 'CSS']
# Demonstrates splitting a string by new line characters.
```

Handling New Lines

Correcting a Scrambled Sentence Using Python

This slide demonstrates a step-by-step approach to unscrambling a sentence, leveraging Python's list capabilities.

From Scramble to Coherence: A Step-by-Step Correction

```
# Beginning with a scrambled message
scrambled_message = "world the save to time no is there"
# Objective: Reorder the words to form a coherent sentence.
```

From Scramble to Coherence: A Step-by-Step Correction

```
# Splitting the scrambled sentence into a list of words
scrambled_message = "world the save to time no is there"
scrambled_list = scrambled_message.split(' ')
# Now we have a list of words to work with.
```

Step 2: Splitting into Words

From Scramble to Coherence: A Step-by-Step Correction

```
# Reversing the list to correct the order of words
scrambled_message = "world the save to time no is there"
scrambled_list = scrambled_message.split(' ')
reversed_list = scrambled_list[::-1]
# The words are now in the correct order to form a meaningful sentence.
```

Step 3: Reversing the Word Order

From Scramble to Coherence: A Step-by-Step Correction

```
# Joining the reversed list back into a sentence
scrambled_message = "world the save to time no is there"
scrambled_list = scrambled_message.split(' ')
reversed_list = scrambled_list[::-1]
corrected_message = " ".join(reversed_list)
# We've reconstructed the sentence into its intended form.
```

Step 4: Forming the Corrected Sentence

From Scramble to Coherence: A Step-by-Step Correction

```
# Combining all steps to correct the scrambled message
scrambled_message = "world the save to time no is there"
scrambled_list = scrambled_message.split(' ')
reversed_list = scrambled_list[::-1]
corrected_message = " ".join(reversed_list)
print(corrected_message)  # Output: there is no time to save the world
```

Full Correction Process

Mastering Loops in Python

An in-depth exploration of loop constructs in Python:

- `while` loops for repeated execution based on a condition
- `for` loops for iterating over sequences.

Understanding While Loops

```
# Demonstrating a simple while loop
i = 0

while i < 3:
  print(i)
  i = i + 1

# This loop prints numbers 0 to 2, demonstrating basic loop control with a counter.
```

Basic While Loop Structure

Understanding While Loops

```
# Creating a pattern with input-controlled repetition
no_of_rows = int(input("How many rows you would like? "))
i = 1

while i <= no_of_rows:
  print("😈" * i)
  i += 1

# This example takes user input to determine the number of rows for the pattern.
```

Leveraging For Loops

```python
# Using range() with for loops for controlled iteration
for curr in range(3):
  print(curr)

# Demonstrates iterating over a sequence of numbers generated by range().
```

Leveraging For Loops

```python
# Another approach to generating patterns, now with for loops
no_of_rows = int(input("How many rows you would like? "))

for i in range(1, no_of_rows + 1):
  print("🤩" * i)

# Similar to the while loop example, but showcasing the for loop's iteration capabilities.
```

Manipulating Lists with Loops

```python
# Modifying list elements in place
player_stats = [10, 30, 60]
for i in range(len(player_stats)):
  player_stats[i] *= 2

print(player_stats)

# Showcases how to iterate over a list by index to modify each element.
```

Doubling Values in a List

Manipulating Lists with Loops

```python
# Modifying list elements in place
powered_up_stats = player_stats.copy()
for i in range(len(player_stats)):
  powered_up_stats[i] *= 2
print(powered_up_stats, player_stats)

# Creating a copy and not modifying the original
```

Doubling Values in a List

Manipulating Lists with Loops

```python
# Using list comprehension for efficient list manipulation
powered_up_stats = [stat * 2 for stat in player_stats]
print(powered_up_stats, player_stats)

# Demonstrates doubling the values in player_stats using list comprehension.
```

List Comprehension for Transformation

Advanced List Comprehension

```python
# Calculating the length of each string in a list
avengers = ["Hulk", "Iron man", "Black widow", "Captain america", "Spider man", "Thor"]
names_char_count = [len(avenger) for avenger in avengers]

print(names_char_count)

# This example showcases using list comprehension to calculate the character count of each string in a list.
```

Advanced List Comprehension

```python
# Filtering elements based on their characteristics
long_names = [avenger for avenger in avengers if len(avenger) > 10]

print(long_names)

# Filters the avengers list to include only names longer than 10 characters.
```

That's all folks 🙋‍♂️