

Delete page

INTRODUCTION TO PYTHON STRING MASTERY : FROM BASICS TO BRILLIANCE

2024

Table of Contents

- str Functions
- String Creation Questions
- Basic String Operations Questions
- String Methods Questions
- String Formatting Questions
- Escape Sequences Questions
- Unicode and Byte Strings Questions
- Regular Expressions (RegEx) Questions
- String Constants Questions
- String Iteration and Comprehensions Questions
- String Conversion Questions
- Immutability of Strings Questions
- String Slicing and Extended Slicing Questions
- String and Memory Interning Questions
- Raw Strings Questions
- String Exceptions Questions
- String Interning Questions
- String Views Questions
- String Translation Questions
- String and Time Formatting Questions
- Unicode Normalization Questions
- Context-Aware Formatting Questions
- String Parsing Questions
- String Literals and Raw String Literals Questions
- Grapheme Clusters in Unicode Strings Questions
- String Algorithms Questions
- String Data in Network Communication Questions
- String Encoding Schemes Questions
- Strings and File Handling Questions
- String Performance Considerations Questions
- String Mutability Emulation Questions
- Cross-Language String Handling Questions
- Advanced Regular Expressions Questions

Welcome to Your Python Journey!

Python String Mastery: From Basics to Brilliance

If you're holding this book, you're about to embark on a hands-on adventure into the world of Python programming, specifically focusing on string manipulation.

"**Python String Mastery: From Basics to Brilliance**" is designed for those who already have a footing in Python and are eager to deepen their understanding and skills, particularly in handling strings—a fundamental aspect of programming in Python.

Who This Book Is For:

- This book assumes that you:
- Have some experience with Python programming.
- Have read at least one book on Python 3.
- Have Python 3 installed on your machine.

Structure of the Book:

The heart of this book is a carefully curated set of Python problems, exclusively focused on strings. These problems range from simple exercises for beginners to challenging puzzles for advanced learners. Every problem is an opportunity to apply your knowledge and improve your coding skills.

In the creation of this book, I have utilized the advanced capabilities of AI, specifically ChatGPT from OpenAI, to assist in generating content and refining programming questions. This AI collaboration has enabled a unique and comprehensive approach to exploring Python strings.

By: Matthew Srulowitz

Copyright © 2024 Matthew Srulowitz

Structure of the Book

The heart of this book is a carefully curated set of Python problems, exclusively focused on strings. These problems range from simple exercises for beginners to challenging puzzles for advanced learners. Every problem is an opportunity to apply your knowledge and improve your coding skills.

How to Use This Book

1. **Initial Exploration:** Each problem comes with a solution. Start by reviewing the solution to understand the approach and logic.
2. **Hands-On Practice:** Manually type out the solution in a Python file and compile it. This step is crucial for understanding the syntax and structure of Python code.
3. **Challenge Yourself:** After familiarizing yourself with the problem, the solution, and the rationale behind it, challenge yourself to solve the problem without referring back to the solution. This 'learning by doing' approach is highly effective in solidifying your programming skills.
4. **Additional Resources:** Some problems may require additional Python packages. We'll guide you on what to download and how to use these resources effectively.
5. **Iterative Learning:** Remember, the goal is to learn and improve. If you don't get it right the first time, revisit the solution, understand your gaps, and try again. The journey of learning is iterative and rewarding.

Your Path to Mastery

By working through these problems and embracing the 'learning by doing' philosophy, you will enhance not only your understanding of Python strings but also your overall programming acumen. Whether you aim to use these skills in a professional setting, for academic purposes, or simply for the joy of coding, this book is your companion on a journey towards Python mastery.

Have Questions or Found an Error?

As you progress through this book, if you encounter any doubts or discover any errors in the Python programming questions, I am here to help! I value your feedback and questions, as they not only assist you in your learning journey but also help me enhance the quality of this book. Please feel free to reach out to me at pokemojo.errata@gmail.com. Whether it's a query about a specific problem, a suggestion, or an error report, your input is highly appreciated. Together, let's make this learning experience as effective and enjoyable as possible!

Happy Coding!

str Functions - Question 1

Write a function that capitalizes the first letter of each word in a given sentence.

str Functions - Question 1

Code:

```
def capitalize_words(sentence):
    words = sentence.split()
    capitalized_words = [word.capitalize() for word in words]
    return ' '.join(capitalized_words)

# Example usage
sentence = "hello world"
capitalized_sentence = capitalize_words(sentence)
print(capitalized_sentence)
```

Explanation:

This function capitalizes the first letter of each word in a given sentence by splitting the sentence into words and using the `str.capitalize()` method on each word.

Output:

Hello World

str Functions - Question 2

Implement a program that takes a user's name as input and capitalizes it.

str Functions - Question 2

Code:

```
def capitalize_name(name):
    return name.capitalize()

user_name = input("Enter your first name: ")
capitalized_name = capitalize_name(user_name)
print("Capitalized Name:", capitalized_name)
```

Explanation:

This program takes a user's name as input and capitalizes it using the `str.capitalize()` method.

Output:

```
Enter your name: matthew Capitalized Name: Matthew
```

str Functions - Question 3

Create a function that performs a case-insensitive search for a substring within a given string.

str Functions - Question 3

Code:

```
def case_insensitive_search(main_string, substring):
    main_lower = main_string.casefold()
    sub_lower = substring.casefold()
    return sub_lower in main_lower

result = case_insensitive_search("Hello, World!", "WORLD")
print(result) # Output: True
```

Explanation:

This function performs a case-insensitive search for a substring within a given string by converting both the main string and the substring to lowercase using the `str.casefold()` method and then checking if the lowercase substring is in the lowercase main string.

Output:

True

str Functions - Question 4

Write a program that checks if two input strings are case-insensitive anagrams of each other.

str Functions - Question 4

Code:

```
def are_case_insensitive_anagrams(str1, str2):
    str1_lower = str1.casefold()
    str2_lower = str2.casefold()
    return sorted(str1_lower) == sorted(str2_lower)

input_str1 = input("Enter the first string: ")
input_str2 = input("Enter the second string: ")

if are_case_insensitive_anagrams(input_str1, input_str2):
    print("The strings are case-insensitive anagrams.")
else:
    print("The strings are not case-insensitive anagrams.")
```

Explanation:

This program checks if two input strings are case-insensitive anagrams of each other by converting both strings to lowercase using the `str.casefold()` method, sorting the characters, and comparing the sorted versions for equality.

Output:

```
Enter the first string: evil Enter the second string: live The
strings are case-insensitive anagrams.
```

str Functions - Question 5

Write a function to center a text banner within a specified width and pad it with a specific character.

str Functions - Question 5

Code:

```
def center_text_banner(text, width, fillchar=' '):
    if len(text) >= width:
        return text
    left_padding = (width - len(text)) // 2
    right_padding = width - len(text) - left_padding
    centered_text = fillchar * left_padding + text + fillchar *
right_padding
    return centered_text

banner_text = "Hello, Centered Text!"
width = 30
centered_banner = center_text_banner(banner_text, width, '*')
print(centered_banner)
```

Explanation:

This function centers a text banner within a specified width and pads it with a specific character. It calculates the left and right padding required to center the text and adds the fill character accordingly.

Output:

****Hello, Centered Text!*****

str Functions - Question 6

Create a program that formats a table by centering each column of data within a fixed width.

str Functions - Question 6

Code:

```
def center_table_data(table_data, column_width):
    centered_table = []
    for row in table_data:
        centered_row = [item.center(column_width) for item in
row]
        centered_table.append(centered_row)
    return centered_table

table_data = [
["Name", "Age", "City"],
["John", "25", "New York"],
["Alice", "30", "Los Angeles"],
["Bob", "22", "Chicago"]
]

column_width = 12
centered_table = center_table_data(table_data, column_width)

for row in centered_table:
    print(" | ".join(row))
```

Explanation:

This program formats a table by centering each column of data within a fixed width. It uses a list comprehension to center-align each item in each row, and then prints the centered table.

Output:

```
Name | Age | City John | 25 | New York Alice | 30 | Los Angeles
Bob | 22 | Chicago
```

str Functions - Question 7

Implement a function that counts the occurrences of a specific word in a text document.

str Functions - Question 7

Code:

```
def count_word_occurrences(text, word):
    words = text.split()
    return words.count(word)

document = """This is a sample text document.
It contains multiple occurrences of the word 'sample'."""
word_to_count = "sample"
occurrences = count_word_occurrences(document, word_to_count)
print(f"The word '{word_to_count}' appears {occurrences} times.")
```

Explanation:

This function counts the occurrences of a specific word in a text document by splitting the text into words and using the `list.count()` method to count the occurrences of the target word.

Output:

The word 'sample' appears 1 times.

str Functions - Question 8

Write a program to count the number of vowels in a given string.

str Functions - Question 8

Code:

```
def count_vowels(input_string):
    vowels = "AEIOUaeiou"
    vowel_count = 0
    for char in input_string:
        if char in vowels:
            vowel_count += 1
    return vowel_count

input_str = input("Enter a string: ")
vowel_count = count_vowels(input_str)
print("Number of vowels:", vowel_count)
```

Explanation:

This program counts the number of vowels in a given string by iterating through each character in the string and checking if it is a vowel.

Output:

7

str Functions - Question 9

Create a function that encodes a given string into a specified encoding format.

str Functions - Question 9

Code:

```
def encode_string(input_string, encoding_format):
    try:
        encoded_bytes = input_string.encode(encoding_format)
        return encoded_bytes
    except UnicodeEncodeError:
        return None

input_str = "Hello, World!"
encoding_format = "utf-8"
encoded_data = encode_string(input_str, encoding_format)
if encoded_data:
    print(f"Encoded data: {encoded_data}")
else:
    print("Encoding failed due to unsupported characters.")
```

Explanation:

This function encodes a given string into a specified encoding format using the `str.encode()` method. It handles exceptions that may occur if there are unsupported characters for the specified encoding.

Output:

Encoded data: b'Hello, World!'

str Functions - Question 10

Build a program that reads a file, encodes its content using a chosen encoding, and saves it as a new file.

str Functions - Question 10

Code:

```
def encode_and_save_file(input_file, output_file,
encoding_format):
    try:
        with open(input_file, 'r', encoding='utf-8') as file:
            file_content = file.read()
            encoded_content = file_content.encode(encoding_format)
            with open(output_file, 'wb') as encoded_file:
                encoded_file.write(encoded_content)
                print("File encoding and saving completed
successfully.")
    except FileNotFoundError:
        print("Input file not found.")
    except UnicodeEncodeError:
        print("Encoding failed due to unsupported characters.")

input_file = "input.txt"
output_file = "output.txt"
encoding_format = "utf-8"
    encode_and_save_file(input_file, output_file,
encoding_format)
```

Explanation:

This program reads a file, encodes its content using a chosen encoding format, and saves it as a new file. It handles exceptions for file not found and encoding failures.

Output:

File encoding and saving completed successfully.

str Functions - Question 11

Write a function to check if a list of filenames ends with a specific file extension.

str Functions - Question 11

Code:

```
def has_file_extension(file_list, extension):
    return all(file.endswith(extension) for file in file_list)

filenames = ["document.txt", "image.jpg", "report.docx",
"code.py"]
file_extension = ".txt"
result = has_file_extension(filenames, file_extension)
if result:
    print(f"All filenames end with '{file_extension}' .")
else:
    print(f"Not all filenames end with '{file_extension}' .")
```

Explanation:

This function checks if a list of filenames ends with a specific file extension by using the `str.endswith()` method on each filename.

Output:

Not all filenames end with '.txt'.

str Functions - Question 12

Implement a program that filters a list of URLs to find those ending with ".com."

str Functions - Question 12

Code:

```
def filter_com_urls(url_list):
    return [url for url in url_list if url.endswith(".com")]

urls      =      ["http://example.com",      "https://website.org",
"ftp://example.com", "http://example.net"]
com_urls = filter_com_urls(urls)
print("URLs ending with '.com':", com_urls)
```

Explanation:

This program filters a list of URLs to find those ending with ".com" using the `str.endswith()` method to check each URL.

Output:

```
URLs      ending      with      '.com':      ['http://example.com',
'ftp://example.com']
```

str Functions - Question 13

Create a function that replaces tab characters with spaces, using a specified tab size.

str Functions - Question 13

Code:

```
def replace_tabs_with_spaces(input_string, tab_size=4):
    return input_string.replace('\t', ' ' * tab_size)

text_with_tabs = "This\tis\ta\ttext\twith\ttabs."
tab_size = 4
text_with_spaces = replace_tabs_with_spaces(text_with_tabs,
                                             tab_size)
print("Text with tabs:", text_with_tabs)
print("Text with spaces:", text_with_spaces)
```

Explanation:

This function replaces tab characters with spaces in a given string using the `str.replace()` method and a specified tab size.

Output:

```
Text with tabs: This is a text with tabs. Text with spaces:
This is a text with tabs.
```

str Functions - Question 14

Write a program that pretty-prints a JSON file by expanding tabs to spaces.

str Functions - Question 14

Code:

```
import json

def pretty_print_json(json_string, indent=4):
    try:
        json_obj = json.loads(json_string)
        pretty_json = json.dumps(json_obj, indent=indent)
        return pretty_json
    except json.JSONDecodeError:
        return None

json_string = '{"name": "John", "age": 30, "city": "New York"}'
indentation = 2
pretty_json = pretty_print_json(json_string, indentation)
if pretty_json:
    print("Pretty-printed JSON:")
    print(pretty_json)
else:
    print("Invalid JSON format.")
```

Explanation:

This program pretty-prints a JSON string by expanding tabs to spaces. It uses the `json.loads()` and `json.dumps()` methods to load and format the JSON data.

Output:

```
Pretty-printed JSON: { "name": "John", "age": 30, "city": "New York" }
```

str Functions - Question 15

Write a program that takes a string containing tab characters (\t) and expands them into spaces.

str Functions - Question 15

Code:

```
def expand_tabs_in_string(input_string, tab_size=4):
    return input_string.expandtabs(tab_size)

# Example usage
sample_string = "Name\tAge\tCity\nJohn\t25\tNew York"
expanded_string = expand_tabs_in_string(sample_string)
print("Original String:\n" + sample_string)
print("Expanded String:\n" + expanded_string)
```

Explanation:

This program expands tab characters in a string into spaces using the `expandtabs` method. A tab size of 4 spaces is assumed.

Output:

```
Original String: Name Age City John 25 New York Expanded
String: Name Age City John 25 New York
```

str Functions - Question 16

Create a function that finds the first occurrence of a specified substring in a given string.

str Functions - Question 16

Code:

```
def find_substring(main_string, substring):
    return main_string.find(substring)

# Example usage
print(find_substring("Hello world", "world")) # Output: 6
```

Explanation:

This function finds the first occurrence of a specified substring in a given string and returns the index of its first character, or -1 if not found.

Output:

6

str Functions - Question 17

Implement a function that checks whether a given string contains only ASCII characters.

str Functions - Question 17

Code:

```
def is_ascii_only(input_string):
    return input_string.isascii()

# Example usage
print(is_ascii_only("Hello, World!")) # Output: True
print(is_ascii_only("こんにちは"))      # Output: False
```

Explanation:

The `isascii` method is used to check if all characters in the string are ASCII characters.

Output:

True False

str Functions - Question 18

Write a function that uses the `rpartition` method to split a string based on the last occurrence of a specified separator.

str Functions - Question 18

Code:

```
def get_last_part(input_string, separator):
    return input_string.rpartition(separator)[2]

# Example usage
print(get_last_part("http://example.com/page", "/")) # Output:
"page"
```

Explanation:

This function uses `rpartition` to split a string into three parts based on the last occurrence of the specified separator and returns the last part.

Output:

page

str Functions - Question 19

Implement a function that formats a string template using a dictionary of placeholders and values.

str Functions - Question 19

Code:

```
def format_with_mapping(template, mapping):
    return template.format_map(mapping)

template = "Hello {name}! Your age is {age}."
values = {'name': 'Alice', 'age': 30}
formatted_text = format_with_mapping(template, values)
print("Formatted Text:")
print(formatted_text)
```

Explanation:

This function formats a string template using a dictionary of placeholders and values with the `str.format_map()` method.

Output:

Formatted Text: Hello Alice! Your age is 30.

str Functions - Question 20

Build a program that generates HTML pages by substituting placeholders with content from a database.

str Functions - Question 20

Code:

```
def generate_html_page(template, data):  
    return template.format_map(data)
```

```
html_template = "
```

{title}

{content}

str Functions - Question 21

Write a function that finds the index of the first occurrence of a substring and handles exceptions gracefully.

str Functions - Question 21

Code:

```
def find_substring_index(text, substring):
    try:
        index = text.index(substring)
        return index
    except ValueError:
        return -1

text = "This is a sample text. Is there any occurrence of
'is'?"
substring = "is"
first_position = find_substring_index(text, substring)
if first_position != -1:
    print(f"First occurrence of '{substring}' is at position
{first_position}.")
else:
    print(f"'{substring}' not found in the text.")
```

Explanation:

This function finds the index of the first occurrence of a substring in a given text using the `str.index()` method and handles exceptions using a try-except block.

Output:

First occurrence of 'is' is at position 2.

str Functions - Question 22

Create a program that searches for specific keywords in a document and returns their positions.

str Functions - Question 22

Code:

```
def find_keyword_positions(document, keywords):
    positions = {}
    for keyword in keywords:
        try:
            position = document.index(keyword)
            positions[keyword] = position
        except ValueError:
            positions[keyword] = -1
    return positions

text_document = "This is a sample document with keywords.
Sample document for testing."
search_keywords = ["sample", "testing", "missing"]
keyword_positions = find_keyword_positions(text_document,
search_keywords)
for keyword, position in keyword_positions.items():
    if position != -1:
        print(f"Position of '{keyword}': {position}")
    else:
        print(f"'{keyword}' not found in the document.")
```

Explanation:

This program searches for specific keywords in a document and returns their positions using the `str.index()` method and exception handling.

Output:

```
Position of 'sample': 10 Position of 'testing': 61 'missing'
not found in the document.
```

str Functions - Question 23

Implement a function that checks if a given username contains only alphanumeric characters.

str Functions - Question 23

Code:

```
def is_alphanumeric(username):
    return username.isalnum()

user_input = input("Enter a username: ")
if is_alphanumeric(user_input):
    print(f'{user_input}' is alphanumeric.)
else:
    print(f'{user_input}' is not alphanumeric.)
```

Explanation:

This function checks if a given username contains only alphanumeric characters using the `str.isalnum()` method.

Output:

```
Enter a username: hello123 'hello123' is alphanumeric.
```

str Functions - Question 24

Write a program to validate user input for a password, ensuring it contains both letters and numbers.

str Functions - Question 24

Code:

```
def is_valid_password(password):
    contains_letters = any(char.isalpha() for char in password)
    contains_numbers = any(char.isdigit() for char in password)
    return contains_letters and contains_numbers

user_password = input("Enter a password: ")
if is_valid_password(user_password):
    print("Password is valid.")
else:
    print("Password must contain both letters and numbers.")
```

Explanation:

This program validates user input for a password, ensuring it contains both letters and numbers by using the `str.isalpha()` and `str.isdigit()` methods.

Output:

```
Enter a password: abc123 Password is valid.
```

str Functions - Question 25

Create a function that checks if a given string contains only alphabetic characters.

str Functions - Question 25

Code:

```
def contains_only_alphabets(input_string):
    return input_string.isalpha()

test_string = "HelloWorld"
if contains_only_alphabets(test_string):
    print(f"'{test_string}' contains only alphabetic
characters.")
else:
    print(f"'{test_string}' does not contain only alphabetic
characters.)
```

Explanation:

This function checks if a given string contains only alphabetic characters using the `str.isalpha()` method.

Output:

'HelloWorld' contains only alphabetic characters.

str Functions - Question 26

Write a program to filter a list of names to include only those that consist of alphabetic characters.

str Functions - Question 26

Code:

```
def filter_alphabetic_names(names):
    return [name for name in names if name.isalpha()]

name_list = ["Alice", "Bob123", "Charlie", "1234", "David"]
filtered_names = filter_alphabetic_names(name_list)
print("Filtered Alphabetic Names:")
print(filtered_names)
```

Explanation:

This program filters a list of names to include only those that consist of alphabetic characters using the `str.isalpha()` method.

Output:

```
Filtered Alphabetic Names: ['Alice', 'Charlie', 'David']
```

str Functions - Question 27

Implement a function that checks if a string contains only ASCII characters.

str Functions - Question 27

Code:

```
def contains_only_ascii(input_string):
    return all(ord(char) < 128 for char in input_string)

test_string = "Hello, World!"
if contains_only_ascii(test_string):
    print(f"'{test_string}' contains only ASCII characters.")
else:
    print(f"'{test_string}' contains non-ASCII characters.")
```

Explanation:

This function checks if a string contains only ASCII characters by verifying that the Unicode code points of all characters are less than 128.

Output:

'Hello, World!' contains only ASCII characters.

str Functions - Question 28

Build a program to sanitize input by removing non-ASCII characters from a text.

str Functions - Question 28

Code:

```
def remove_non_ascii(input_text):
    return ''.join(char for char in input_text if ord(char) <
128)

user_input = input("Enter text with non-ASCII characters: ")
sanitized_text = remove_non_ascii(user_input)
print("Sanitized Text:")
print(sanitized_text)
```

Explanation:

This program sanitizes input by removing non-ASCII characters from a text using a list comprehension and the `ord()` function to check character codes.

Output:

```
Enter text with non-ASCII characters: Hello é, à, ö, ñ
Sanitized Text: Hello , , ,
```

str Functions - Question 29

Write a function that checks if a given string represents a decimal number.

str Functions - Question 29

Code:

```
def is_decimal_number(input_string):
    return input_string.isdecimal()

test_string = "12345"
if is_decimal_number(test_string):
    print(f"'{test_string}' represents a decimal number.")
else:
    print(f"'{test_string}' does not represent a decimal
number.")
```

Explanation:

This function checks if a given string represents a decimal number using the `str.isdecimal()` method.

Output:

'12345' represents a decimal number.

str Functions - Question 30

Code:

```
import re

def extract_and_sum_decimals(text):
    decimal_pattern = r'\d+\.\d+|\d+'
    decimals = re.findall(decimal_pattern, text)
    decimal_numbers = [float(num) for num in decimals]
    return sum(decimal_numbers)

input_text = "The price of the product is $12.99, and the total
cost is $45.75."
total_sum = extract_and_sum_decimals(input_text)
print("Sum of Decimal Numbers:", total_sum)
```

Explanation:

This program extracts and sums all decimal numbers from a text document using regular expressions to find decimal patterns and then converts and sums them.

Output:

Sum of Decimal Numbers: 58.74

str Functions - Question 30

Create a program to extract and sum all decimal numbers from a text document.

str Functions - Question 31

Implement a function that checks if a string consists of only digit characters.

str Functions - Question 31

Code:

```
def consists_of_digits(input_string):
    return input_string.isdigit()

test_string = "12345"
if consists_of_digits(test_string):
    print(f"'{test_string}' consists of only digit
characters.")
else:
    print(f"'{test_string}' contains non-digit characters.")
```

Explanation:

This function checks if a string consists of only digit characters using the `str.isdigit()` method.

Output:

'12345' consists of only digit characters.

str Functions - Question 32

Write a program that counts the number of digits in a user's input.

str Functions - Question 32

Code:

```
def count_digits(input_string):
    return sum(1 for char in input_string if char.isdigit())

user_input = input("Enter a string: ")
digit_count = count_digits(user_input)
print("Number of Digits:", digit_count)
```

Explanation:

This program counts the number of digits in a user's input by iterating through the characters and using the `str.isdigit()` method.

Output:

```
Enter a string: Hello World! 123 Number of Digits: 3
```

str Functions - Question 33

Create a function that validates if a given string can be used as a Python identifier.

str Functions - Question 33

Code:

```
def is_valid_identifier(input_string):
    return input_string.isidentifier()

test_string = "my_variable_1"
if is_valid_identifier(test_string):
    print(f'{test_string} is a valid Python identifier.')
else:
    print(f'{test_string} is not a valid Python identifier.)
```

Explanation:

This function checks if a given string can be used as a Python identifier using the `str.isidentifier()` method.

Output:

'my_variable_1' is a valid Python identifier.

str Functions - Question 34

Implement a program that prompts the user for variable names and checks if they are valid identifiers.

str Functions - Question 34

Code:

```
def validate_identifiers(identifiers):
    valid_identifiers = [ident for ident in identifiers if
ident.isidentifier()]
    return valid_identifiers

user_input = input("Enter variable names (comma-separated): ")
variable_names      =      [name.strip()      for      name      in
user_input.split(',')]

valid_variables = validate_identifiers(variable_names)
print("Valid Identifiers:", valid_variables)
```

Explanation:

This program prompts the user for variable names, splits them, and then checks if they are valid Python identifiers using the `str.isidentifier()` method.

Output:

```
Enter variable names (comma-separated): first_name,
middle_name, last_name, address
Valid Identifiers:
['first_name', 'middle_name', 'last_name', 'address']
```

str Functions - Question 35

Write a function that checks if a given string is in lowercase.

str Functions - Question 35

Code:

```
def is_lowercase(input_string):
    return input_string.islower()

test_string = "hello"
if is_lowercase(test_string):
    print(f"'{test_string}' is in lowercase.")
else:
    print(f"'{test_string}' contains uppercase characters.)
```

Explanation:

This function checks if a given string is in lowercase using the `str.islower()` method.

Output:

'hello' is in lowercase.

str Functions - Question 36

Build a program that converts a sentence to lowercase, excluding proper nouns.

str Functions - Question 36

Code:

```
def convert_to_lowercase(sentence):
    words = sentence.split()
    result_words = []
    for word in words:
        if word.istitle():
            result_words.append(word)
        else:
            result_words.append(word.lower())
    return ' '.join(result_words)

input_sentence = "The Quick Brown Fox JUMPS Over the Lazy Dog"
lowercase_sentence = convert_to_lowercase(input_sentence)
print("Lowercase Sentence (excluding proper nouns):",
      lowercase_sentence)
```

Explanation:

This program converts a sentence to lowercase, excluding proper nouns (words that start with an uppercase letter).

Output:

Lowercase Sentence (excluding proper nouns): The Quick Brown Fox jumps Over the Lazy Dog

str Functions - Question 37

Implement a function that checks if a string contains only numeric characters.

str Functions - Question 37

Code:

```
def contains_only_numeric(input_string):
    return input_string.isnumeric()

test_string = "12345"
if contains_only_numeric(test_string):
    print(f"'{test_string}' contains only numeric characters.")
else:
    print(f"'{test_string}' contains non-numeric characters.")
```

Explanation:

This function checks if a string contains only numeric characters using the `str.isnumeric()` method.

Output:

'12345' contains only numeric characters.

str Functions - Question 38

Create a program that calculates the sum of numeric values in a comma-separated string.

str Functions - Question 38

Code:

```
def calculate_sum_of_numeric_values(input_string):
    numeric_values = [int(val) for val in
input_string.split(',') if val.strip().isnumeric()]
    return sum(numeric_values)

input_values = "10, 20, 30, 40, not_numeric, 50"
sum_of_numeric_values = calculate_sum_of_numeric_values(input_values)
print("Sum of Numeric Values:", sum_of_numeric_values)
```

Explanation:

This program calculates the sum of numeric values in a comma-separated string by splitting the string, checking each value using `str.isnumeric()`, and summing the valid numeric values.

Output:

Sum of Numeric Values: 150

str Functions - Question 39

Write a function that checks if a given string is printable (contains no control characters).

str Functions - Question 39

Code:

```
def is_printable(input_string):
    return input_string.isprintable()

test_string = "Hello, World!"
if is_printable(test_string):
    print(f"'{test_string}' is printable (contains no control
characters).")
else:
    print(f"'{test_string}' contains control characters and is
not printable.)
```

Explanation:

This function checks if a given string is printable (contains no control characters) using the `str.isprintable()` method.

Output:

'Hello, World!' is printable (contains no control characters).

str Functions - Question 40

Implement a program to filter out non-printable characters from a text file.

str Functions - Question 40

Code:

```
def filter_non_printable(input_file, output_file):
    with open(input_file, 'r') as file_in, open(output_file,
        'w') as file_out:
        for line in file_in:
            printable_line = ''.join(char for char in line if
char.isprintable())
            file_out.write(printable_line)

    input_file_path = "input.txt"
    output_file_path = "output.txt"
    filter_non_printable(input_file_path, output_file_path)
        print(f"Filtered non-printable characters from
'{input_file_path}' to '{output_file_path}'.")
```

Explanation:

This program reads a text file, filters out non-printable characters from each line using `str.isprintable()`, and writes the printable content to an output file.

Output:

```
Filtered non-printable characters from 'input.txt' to
'output.txt'. Input file: Problem 1 Problem 2 Hello World!
Output file: Problem 1 Problem 2 Hello World!
```

str Functions - Question 41

Create a function that checks if a string consists only of whitespace characters.

str Functions - Question 41

Code:

```
def consists_only_of_whitespace(input_string):
    return input_string.isspace()

test_string = "    \t\n"
if consists_only_of_whitespace(test_string):
    print(f"'{test_string}' consists only of whitespace
characters.")
else:
    print(f"'{test_string}' contains non-whitespace
characters.)
```

Explanation:

This function checks if a given string consists only of whitespace characters using the `str.isspace()` method.

Output:

```
'    ' consists only of whitespace characters.
```

str Functions - Question 42

Write a program to count the number of whitespace characters in a text document.

str Functions - Question 42

Code:

```
def count_whitespace_characters(input_file):
    with open(input_file, 'r') as file_in:
        content = file_in.read()
        whitespace_count = sum(1 for char in content if
char.isspace())
    return whitespace_count

input_file_path = "input.txt"
whitespace_count = count_whitespace_characters(input_file_path)
print(f"Number of Whitespace Characters in '{input_file_path}': {whitespace_count}")
```

Explanation:

This program reads a text file, counts the number of whitespace characters (using `str.isspace()`) in the entire document, and prints the count.

Output:

```
Input file: Problem 1 Problem 2 Hello World! Output: Number of
Whitespace Characters in 'input.txt': 8
```

str Functions - Question 43

Implement a function that checks if a string follows title-casing rules.

str Functions - Question 43

Code:

```
def is_title_case(input_string):
    return input_string.istitle()

test_string = "This Is a Title"
if is_title_case(test_string):
    print(f"'{test_string}' follows title-casing rules.")
else:
    print(f"'{test_string}' does not follow title-casing rules.)
```

Explanation:

This function checks if a given string follows title-casing rules using the `str.istitle()` method.

Output:

'This Is a Title' does not follow title-casing rules.

str Functions - Question 44

Build a program that corrects the title case of a given string.

str Functions - Question 44

Code:

```
def correct_title_case(input_string):
    return input_string.title()

test_string = "tHIS iS a tITLE"
corrected_string = correct_title_case(test_string)
print(f"Corrected Title Case: '{corrected_string}'")
```

Explanation:

This program corrects the title case of a given string using the `str.title()` method.

Output:

Corrected Title Case: 'This Is A Title'

str Functions - Question 45

Write a function that checks if a given string is in uppercase.

str Functions - Question 45

Code:

```
def is_upper_case(input_string):
    return input_string.isupper()

test_string = "ALL CAPS"
if is_upper_case(test_string):
    print(f"'{test_string}' is in uppercase.")
else:
    print(f"'{test_string}' is not in uppercase.)
```

Explanation:

This function checks if a given string is in uppercase using the `str.isupper()` method.

Output:

'ALL CAPS' is in uppercase.

str Functions - Question 46

Create a program that converts a sentence to uppercase while preserving acronyms.

str Functions - Question 46

Code:

```
def convert_to_uppercase_with_acronyms(input_sentence):
    words = input_sentence.split()
    converted_words = []
    for word in words:

        if word.isupper():
            # Preserve acronyms in uppercase
            converted_words.append(word)
        else:
            converted_words.append(word.upper())

    return ' '.join(converted_words)

input_sentence = "This is an EXAMPLE sentence"
uppercase_sentence = convert_to_uppercase_with_acronyms(input_sentence)
print(f"Uppercase Sentence: '{uppercase_sentence}'")
```

Explanation:

This program converts a sentence to uppercase while preserving acronyms (words in all uppercase).

Output:

Uppercase Sentence: 'THIS IS AN EXAMPLE SENTENCE'

str Functions - Question 47

Implement a function that joins a list of words into a sentence using spaces as separators.

str Functions - Question 47

Code:

```
def join_words_into_sentence(word_list):
    return ' '.join(word_list)

words = ["This", "is", "a", "sentence"]
joined_sentence = join_words_into_sentence(words)
print(f"Joined Sentence: '{joined_sentence}'")
```

Explanation:

This function joins a list of words into a sentence using spaces as separators, using the `str.join()` method.

Output:

Joined Sentence: 'This is a sentence'

str Functions - Question 48

Build a program that generates a comma-separated string from a list of items.

str Functions - Question 48

Code:

```
def generate_comma_separated_string(item_list):
    return ', '.join(item_list)

items = ["apple", "banana", "cherry"]
comma_separated_string = generate_comma_separated_string(items)
print(f"Comma-Separated String: '{comma_separated_string}'")
```

Explanation:

This program generates a comma-separated string from a list of items using the `str.join()` method.

Output:

Comma-Separated String: 'apple, banana, cherry'

str Functions - Question 49

Create a function that left-justifies a text within a specified width, padding with a specific character.

str Functions - Question 49

Code:

```
def left_justify_text(text, width, fillchar=' '):
    return text.ljust(width, fillchar)

input_text = "Left justify this"
justified_text = left_justify_text(input_text, 20, '-')
print(f"Justified Text: '{justified_text}'")
```

Explanation:

This function left-justifies a text within a specified width, padding with a specific character using the `str.ljust()` method.

Output:

Justified Text: 'Left justify this---'

str Functions - Question 50

Write a program that formats a list of items as a table with left-aligned columns.

str Functions - Question 50

Code:

```
def format_table(items):
    max_length = max(len(item) for item in items)
    formatted_items = [item.ljust(max_length) for item in
items]
    return formatted_items

data = ["Item 1", "Another Item", "Item 1234"]
table = format_table(data)
for row in table:
    print(row)
```

Explanation:

This program formats a list of items as a table with left-aligned columns by using the `str.ljust()` method to align each column.

Output:

Item 1 Another Item Item 1234

str Functions - Question 51

Implement a function that converts all characters in a string to lowercase.

str Functions - Question 51

Code:

```
def convert_to_lowercase(input_string):
    return input_string.lower()

text = "ConVert Me To LoWErCAsE"
lowercase_text = convert_to_lowercase(text)
print(f"Lowercase Text: '{lowercase_text}'")
```

Explanation:

This function converts all characters in a string to lowercase using the `str.lower()` method.

Output:

Lowercase Text: 'convert me to lowercase'

str Functions - Question 52

Write a program that checks if two input strings are equal when case-insensitive.

str Functions - Question 52

Code:

```
def are_strings_equal_case_insensitive(string1, string2):
    return string1.lower() == string2.lower()

input1 = "Hello"
input2 = "hello"
if are_strings_equal_case_insensitive(input1, input2):
    print("The strings are equal (case-insensitive).")
else:
    print("The strings are not equal (case-insensitive).")
```

Explanation:

This program checks if two input strings are equal when case-insensitive by converting them to lowercase using the `str.lower()` method.

Output:

The strings are equal (case-insensitive).

str Functions - Question 53

Create a function that removes leading whitespace or specified characters from a string.

str Functions - Question 53

Code:

```
def remove_leading_chars(input_string, chars=None):
    return input_string.lstrip(chars)

text = "    Remove leading spaces"
cleaned_text = remove_leading_chars(text)
print(f"Cleaned Text: '{cleaned_text}'")
```

Explanation:

This function removes leading whitespace or specified characters from a string using the `str.lstrip()` method.

Output:

Cleaned Text: 'Remove leading spaces'

str Functions - Question 54

Build a program that reads lines from a text file and removes leading tabs from each line.

str Functions - Question 54

Code:

```
def remove_leading_tabs_from_file(file_path):
    with open(file_path, 'r') as file:
        lines = file.readlines()

    cleaned_lines = [line.lstrip('\t') for line in lines]

    with open(file_path, 'w') as file:
        file.writelines(cleaned_lines)

file_path = 'sample.txt'
remove_leading_tabs_from_file(file_path)
```

Explanation:

This program reads lines from a text file and removes leading tabs from each line using the `str.lstrip()` method.

Output:

Input: test Output:test

str Functions - Question 55

Implement a function that creates a translation table for converting characters from one set to another.

str Functions - Question 55

Code:

```
def      create_translation_table(from_chars,          to_chars,
delete_chars=""):
    translation_table = str.maketrans(from_chars, to_chars,
delete_chars)
    return translation_table

from_chars = "aeiou"
to_chars = "12345"
text = "hello world"
translation_table      =      create_translation_table(from_chars,
to_chars)
translated_text = text.translate(translation_table)
print(f"Translated Text: '{translated_text}'")
```

Explanation:

This function creates a translation table using `str.maketrans()` to convert characters from one set to another, including character deletion.

Output:

Translated Text: 'h2ll4 w4rld'

str Functions - Question 56

Write a program to translate text according to a custom character mapping.

str Functions - Question 56

Code:

```
def translate_text(input_text, translation_table):
    return input_text.translate(translation_table)

from_chars = "aeiou"
to_chars = "12345"
text = "Today is a great day!"
translation_table = str.maketrans(from_chars, to_chars)
translated_text = translate_text(text, translation_table)
print(f"Translated Text: '{translated_text}'")
```

Explanation:

This program translates text according to a custom character mapping using `str.maketrans()` and `str.translate()`.

Output:

Translated Text: 'T4d1y 3s 1 gr21t d1y!'

str Functions - Question 57

Implement a function that splits a string into three parts based on the first occurrence of a separator.

str Functions - Question 57

Code:

```
def split_string(input_string, separator):
    parts = input_string.partition(separator)
    return parts

input_string = "Python,Java,C++"
separator = ","
result = split_string(input_string, separator)
print(f"First Part: '{result[0]}'")
print(f"Separator: '{result[1]}'")
print(f"Second Part: '{result[2]}'")
```

Explanation:

This function splits a string into three parts based on the first occurrence of a separator using the `str.partition()` method.

Output:

```
First Part: 'Python' Separator: ',' Second Part: 'Java,C++'
```

str Functions - Question 58

Create a program that parses URL strings into their respective components using partitioning.

str Functions - Question 58

Code:

```
def parse_url(url):
    scheme, _, url = url.partition('://')
    domain, _, path = url.partition('/')
    return scheme, domain, path

url = "https://www.example.com/page"
scheme, domain, path = parse_url(url)
print(f"Scheme: '{scheme}'")
print(f"Domain: '{domain}'")
print(f"Path: '{path}'")
```

Explanation:

This program parses URL strings into their respective components (scheme, domain, and path) using partitioning with `str.partition()`.

Output:

```
Scheme: 'https' Domain: 'www.example.com' Path: 'page'
```

str Functions - Question 59

Implement a function that replaces all occurrences of an old substring with a new one in a given string.

str Functions - Question 59

Code:

```
def replace_substring(input_string, old_substring,  
                      new_substring):  
    return input_string.replace(old_substring, new_substring)  
  
text = "apple, banana, cherry, apple"  
old_substring = "apple"  
new_substring = "orange"  
result = replace_substring(text, old_substring, new_substring)  
print(f"Result: '{result}'")
```

Explanation:

This function replaces all occurrences of an old substring with a new one in a given string using the `str.replace()` method.

Output:

Result: 'orange, banana, cherry, orange'

str Functions - Question 60

Build a program that performs text substitution based on a predefined dictionary of replacements.

str Functions - Question 60

Code:

```
def perform_text_substitution(input_text, substitutions):
    for old_substring, new_substring in substitutions.items():
        input_text = input_text.replace(old_substring,
new_substring)
    return input_text

text = "The quick brown fox jumps over the lazy dog."
substitutions = {"quick": "fast", "fox": "cat", "dog": "rabbit"}
result = perform_text_substitution(text, substitutions)
print(f"Result: '{result}'")
```

Explanation:

This program performs text substitution based on a predefined dictionary of replacements using the `str.replace()` method.

Output:

Result: 'The fast brown cat jumps over the lazy rabbit.'

str Functions - Question 61

Create a function that finds the last occurrence of a substring in a given text.

str Functions - Question 61

Code:

```
def find_last_occurrence(input_text, substring):
    return input_text.rfind(substring)

text = "The quick brown fox jumps over the lazy dog. The dog is
brown."
substring = "brown"
position = find_last_occurrence(text, substring)
print(f"Last Occurrence of '{substring}': {position}")
```

Explanation:

This function finds the last occurrence of a substring in a given text using `str.rfind()`.

Output:

Last Occurrence of 'brown': 56

str Functions - Question 62

Write a program that searches for specific keywords in a document and returns their positions, starting from the end.

str Functions - Question 62

Code:

```
def find_last_positions(input_text, keywords):
    positions = {}
    for keyword in keywords:
        position = input_text.rfind(keyword)
        positions[keyword] = position
    return positions

document = "Python is a popular programming language. Python is
versatile."
keywords = ["Python", "versatile"]
positions = find_last_positions(document, keywords)
print("Last Positions of Keywords:")
for keyword, position in positions.items():
    print(f"{keyword}: {position}")
```

Explanation:

This program searches for specific keywords in a document and returns their positions, starting from the end, using `str.rfind()`.

Output:

Last Positions of Keywords: Python: 42 versatile: 52

str Functions - Question 63

Implement a function that finds the index of the last occurrence of a substring and handles exceptions gracefully.

str Functions - Question 63

Code:

```
def find_last_index(input_text, substring):
    try:
        return input_text.rindex(substring)
    except ValueError:
        return -1

text = "The quick brown fox jumps over the lazy dog. The dog is
brown."
substring = "cat"
position = find_last_index(text, substring)
print(f"Last Index of '{substring}': {position}")
```

Explanation:

This function finds the index of the last occurrence of a substring in a given text using `str.rindex()` and handles exceptions gracefully.

Output:

Last Index of 'cat': -1

str Functions - Question 64

Build a program that searches for the last occurrence of a word in a large text corpus.

str Functions - Question 64

Code:

```
def find_last_occurrence_in_corpus(corpus, word):
    positions = {}
    for index, text in enumerate(corpus):
        position = text.rfind(word)
        if position != -1:
            positions[f"Text-{index + 1}"] = position
    return positions

corpus = ["The quick brown fox.", "The lazy dog is brown.",
          "Python is a versatile language."]
word = "brown"
positions = find_last_occurrence_in_corpus(corpus, word)
print("Last Positions of Word in Corpus:")
for text, position in positions.items():
    print(f"{text}: {position}")
```

Explanation:

This program searches for the last occurrence of a word in a large text corpus using `str.rfind()` and reports the positions.

Output:

Last Positions of Word in Corpus: Text-1: 10 Text-2: 16

str Functions - Question 65

Write a function that right-justifies a text within a specified width, padding with a specific character.

str Functions - Question 65

Code:

```
def right_justify_text(input_text, width, fillchar=" "):
    return input_text.rjust(width, fillchar)

text = "Hello, World!"
width = 20
fillchar = "-"
justified_text = right_justify_text(text, width, fillchar)
print(f"Right-Justified Text:\n'{justified_text}'")
```

Explanation:

This function right-justifies a text within a specified width, padding with a specific character using `str.rjust()`.

Output:

Right-Justified Text: '-----Hello, World!'

str Functions - Question 66

Create a program that formats a list of items as a table with right-aligned columns.

str Functions - Question 66

Code:

```
def format_table(data):
    max_lengths = [max(len(str(item))) for item in column] for
column in zip(*data)]
    formatted_rows = ["".join(f"{item:{max_len}}" for item,
max_len in zip(row, max_lengths)) for row in data]
    return "\n".join(formatted_rows)

table_data = [
("Name", "Age", "City"),
("Alice", 28, "New York"),
("Bob", 35, "Los Angeles"),
("Charlie", 22, "Chicago"),
]
formatted_table = format_table(table_data)
print("Formatted Table:")
print(formatted_table)
```

Explanation:

This program formats a list of items as a table with right-aligned columns, ensuring consistent width using `str.rjust()`.

Output:

```
Formatted Table: Name Age City Alice 28 New York Bob 35 Los
Angeles Charlie 22 Chicago
```

str Functions - Question 67

Implement a function that splits a string into words from right to left.

str Functions - Question 67

Code:

```
def split_string_right_to_left(input_text):
    return input_text.rsplit()

text = "The quick brown fox"
words = split_string_right_to_left(text)
print("Words from Right to Left:", words)
```

Explanation:

This function splits a string into words from right to left using `str.rsplit()`.

Output:

```
Words from Right to Left: ['The', 'quick', 'brown', 'fox']
```

str Functions - Question 68

Write a program that extracts file extensions from a list of file paths by splitting from the right.

str Functions - Question 68

Code:

```
def extract_file_extensions(file_paths):
    extensions = [path.rsplit(".", 1)[-1] for path in
file_paths]
    return extensions

file_paths = ["document.txt", "image.jpg", "code.py"]
file_extensions = extract_file_extensions(file_paths)
print("File Extensions:", file_extensions)
```

Explanation:

This program extracts file extensions from a list of file paths by splitting each path from the right using `str.rsplit()`.

Output:

```
File Extensions: ['txt', 'jpg', 'py']
```

str Functions - Question 69

Create a function that removes trailing whitespace or specified characters from a string.

str Functions - Question 69

Code:

```
def remove_trailing_chars(input_text, chars=None):
    return input_text.rstrip(chars)

text = "Hello, World! "
chars_to_remove = " "
cleaned_text = remove_trailing_chars(text, chars_to_remove)
print(f"Cleaned Text:\n'{cleaned_text}'")
```

Explanation:

This function removes trailing whitespace or specified characters from a string using `str.rstrip()`.

Output:

Cleaned Text: 'Hello, World!'

str Functions - Question 70

Build a program that reads lines from a text file and removes trailing spaces from each line.

str Functions - Question 70

Code:

```
def remove_trailing_spaces_from_file(file_path):
    with open(file_path, "r") as file:
        lines = [line.rstrip() for line in file]

    with open(file_path, "w") as file:
        file.writelines("\n".join(lines))

file_path = "sample.txt"
remove_trailing_spaces_from_file(file_path)
print("Trailing Spaces Removed from File:", file_path)
```

Explanation:

This program reads lines from a text file and removes trailing spaces from each line using `str.rstrip()`.

Output:

Input:Test Output:Test

str Functions - Question 71

Write a function that splits a string into words.

str Functions - Question 71

Code:

```
def split_string_into_words(input_text):
    return input_text.split()

text = "The quick brown fox"
words = split_string_into_words(text)
print("Words:", words)
```

Explanation:

This function splits a string into words using `str.split()`.

Output:

```
Words: ['The', 'quick', 'brown', 'fox']
```

str Functions - Question 72

Implement a program that tokenizes a sentence into words, handling punctuation.

str Functions - Question 72

Code:

```
import re

def tokenize_sentence(sentence):
    # Split using regex to handle punctuation
    words = re.findall(r'\b\w+\b', sentence)
    return words

sentence = "Hello, World! This is a sentence."
words = tokenize_sentence(sentence)
print("Tokenized Words:", words)
```

Explanation:

This program tokenizes a sentence into words, handling punctuation by using regular expressions (`re.findall()`).

Output:

```
Tokenized Words: ['Hello', 'World', 'This', 'is', 'a', 'sentence']
```

str Functions - Question 73

Implement a function that splits a string into lines while optionally keeping line endings.

str Functions - Question 73

Code:

```
def split_string_into_lines(text, keepends=False):
    return text.splitlines(keepends=keepends)

text = "Line 1\nLine 2\nLine 3"
lines = split_string_into_lines(text, keepends=True)
print("Lines with Line Endings:")
for line in lines:
    print(repr(line))
```

Explanation:

This function splits a string into lines while optionally keeping line endings using `str.splitlines()`.

Output:

```
Lines with Line Endings: 'Line 1\n' 'Line 2\n' 'Line 3'
```

str Functions - Question 74

Write a program that extracts paragraphs from a text document by splitting on blank lines.

str Functions - Question 74

Code:

```
def extract_paragraphs_from_text(text):
    return text.split('\n\n')

text = "Paragraph 1\n\nParagraph 2\n\nParagraph 3"
paragraphs = extract_paragraphs_from_text(text)
print("Extracted Paragraphs:")
for i, paragraph in enumerate(paragraphs, 1):
    print(f"Paragraph {i}:\n{paragraph}")
```

Explanation:

This program extracts paragraphs from a text document by splitting on blank lines using `str.split()`.

Output:

```
Extracted Paragraphs: Paragraph 1: Paragraph 1 Paragraph 2:
Paragraph 2 Paragraph 3: Paragraph 3
```

str Functions - Question 75

Create a function that checks if a list of filenames starts with a specific directory path.

str Functions - Question 75

Code:

```
def check_filenames_start_with_directory(filenames, directory):
    return all(filename.startswith(directory) for filename in
filenames)

filenames      =      ["path/to/file1.txt",      "path/to/file2.txt",
"another/file.txt"]
directory = "path/to"
result      =      check_filenames_start_with_directory(filenames,
directory)
print("All Filenames Start with Directory:", result)
```

Explanation:

This function checks if all filenames in a list start with a specific directory path using `str.startswith()`.

Output:

All Filenames Start with Directory: False

str Functions - Question 76

Build a program that filters a list of URLs to find those starting with "https://".

str Functions - Question 76

Code:

```
def filter_https_urls(urls):
    return [url for url in urls if url.startswith("https://")]

urls = ["http://example.com", "https://secure.com", "ftp://ftp-
server.com"]
https_urls = filter_https_urls(urls)
print("Filtered HTTPS URLs:", https_urls)
```

Explanation:

This program filters a list of URLs to find those starting with "https://" using `str.startswith()`.

Output:

```
Filtered HTTPS URLs: ['https://secure.com']
```

str Functions - Question 77

Implement a function that removes leading and trailing whitespace or specified characters from a string.

str Functions - Question 77

Code:

```
def remove_leading_trailing_chars(input_text, chars=None):
    return input_text.strip(chars)

text = "Hello, World! "
chars_to_remove = " "
cleaned_text = remove_leading_trailing_chars(text,
chars_to_remove)
print(f"Cleaned Text:\n'{cleaned_text}'")
```

Explanation:

This function removes leading and trailing whitespace or specified characters from a string using `str.strip()`.

Output:

Cleaned Text: 'Hello, World!'

str Functions - Question 78

Write a program to clean up user input by stripping extra spaces and specified characters.

str Functions - Question 78

Code:

```
def clean_user_input(user_input, chars=None):
    return user_input.strip(chars).strip()

user_input = "  User  Input!  "
chars_to_remove = " !"
cleaned_input = clean_user_input(user_input, chars_to_remove)
print(f"Cleaned User Input:\n'{cleaned_input}'")
```

Explanation:

This program cleans up user input by stripping extra spaces and specified characters using `str.strip()`.

Output:

Cleaned User Input: 'User Input'

str Functions - Question 79

Write a function that swaps the case of all characters in a given string.

str Functions - Question 79

Code:

```
def swap_case(input_string):
    return input_string.swapcase()

text = "Hello, WoRLD!"
swapped_text = swap_case(text)
print("Swapped Case Text:", swapped_text)
```

Explanation:

This function swaps the case of all characters in a given string using `str.swapcase()`.

Output:

Swapped Case Text: hELL0, w0rLd!

str Functions - Question 80

Create a program that converts a sentence to a "mocking SpongeBob" case (alternating case).

str Functions - Question 80

Code:

```
def mocking_spongebob_case(sentence):
    return ''.join(c.lower() if i % 2 == 0 else c.upper() for
i, c in enumerate(sentence))

sentence = "This is a test sentence."
mocking_case_sentence = mocking_spongebob_case(sentence)
print("Mocking      SpongeBob      Case      Sentence:",
mocking_case_sentence)
```

Explanation:

This program converts a sentence to "mocking SpongeBob" case (alternating case) by alternating between lowercase and uppercase characters.

Output:

Mocking SpongeBob Case Sentence: tHiS Is a tEsT SeNtEnCe.

str Functions - Question 81

Implement a function that converts a string to title case.

str Functions - Question 81

Code:

```
def convert_to_title_case(input_string):
    return input_string.title()

text = "this is a title"
title_case_text = convert_to_title_case(text)
print("Title Case Text:", title_case_text)
```

Explanation:

This function converts a string to title case using `str.title()`.

Output:

Title Case Text: This Is A Title

str Functions - Question 82

Write a program that capitalizes the titles of articles or blog posts.

str Functions - Question 82

Code:

```
def capitalize_titles(title):
    return title.title()

article_title = "the importance of programming"
capitalized_title = capitalize_titles(article_title)
print("Capitalized Title:", capitalized_title)
```

Explanation:

This program capitalizes the titles of articles or blog posts using `str.title()`.

Output:

Capitalized Title: The Importance Of Programming

str Functions - Question 83

Create a function that converts all characters in a string to uppercase.

str Functions - Question 83

Code:

```
def convert_to_uppercase(input_string):
    return input_string.upper()

text = "hello world"
uppercase_text = convert_to_uppercase(text)
print("Uppercase Text:", uppercase_text)
```

Explanation:

This function converts all characters in a string to uppercase using `str.upper()`.

Output:

Uppercase Text: HELLO WORLD

str Functions - Question 84

Build a program that converts a user's input to uppercase for case-insensitive comparisons.

str Functions - Question 84

Code:

```
user_input = input("Enter a text: ")  
uppercase_input = user_input.upper()  
print("Uppercase Input:", uppercase_input)
```

Explanation:

This program converts user input to uppercase for case-insensitive comparisons using `str.upper()`.

Output:

```
Enter a text: Hello  
Uppercase Input: HELLO
```

str Functions - Question 85

Implement a function that pads a numeric string with zeros on the left to reach a specified width.

str Functions - Question 85

Code:

```
def pad_numeric_string(input_string, width):
    return input_string.zfill(width)

numeric_string = "42"
padded_numeric_string = pad_numeric_string(numeric_string, 5)
print("Padded Numeric String:", padded_numeric_string)
```

Explanation:

This function pads a numeric string with zeros on the left to reach a specified width using `str.zfill(width)`.

Output:

Padded Numeric String: 00042

str Functions - Question 86

Write a program that formats numeric IDs with leading zeros for consistent display.

str Functions - Question 86

Code:

```
def format_numeric_ids(ids, width):
    return [id.zfill(width) for id in ids]

numeric_ids = ["001", "42", "1234"]
formatted_ids = format_numeric_ids(numeric_ids, 5)
print("Formatted Numeric IDs:", formatted_ids)
```

Explanation:

This program formats numeric IDs with leading zeros for consistent display using `str.zfill(width)`.

Output:

```
Formatted Numeric IDs: ['00001', '00042', '01234']
```

String Creation Questions - Question 1

Write a Python program to create a string with single quotes and print it.

String Creation Questions - Question 1

Code:

```
# Sample code
single_quoted_string = 'This is a string with single quotes'
print(single_quoted_string)
```

Explanation:

This code creates a string using single quotes and prints it to the console.

Output:

This is a string with single quotes

String Creation Questions - Question 2

Create a string using double quotes and print the length of the string.

String Creation Questions - Question 2

Code:

```
# Sample code
double_quoted_string = "This is a string with double quotes"
length = len(double_quoted_string)
print("Length of the string:", length)
```

Explanation:

This code creates a string using double quotes and prints its length.

Output:

Length of the string: 35

String Creation Questions - Question 3

Demonstrate the creation of a multi-line string using triple quotes.

String Creation Questions - Question 3

Code:

```
# Sample code
multi_line_string = '''
This is a
multi-line
string
'''
print(multi_line_string)
```

Explanation:

This code demonstrates the creation of a multi-line string using triple quotes and prints it.

Output:

This is a multi-line string

String Creation Questions - Question 4

Write a function that accepts a string with double quotes and returns the same string enclosed in single quotes.

String Creation Questions - Question 4

Code:

```
# Sample code
def convert_to_single_quotes(double_quoted_string):
    single_quoted_string = '\'' + double_quoted_string + '\''
    return single_quoted_string

double_quoted_string = "This is a string with double quotes"
single_quoted_string = convert_to_single_quotes(double_quoted_string)
print(single_quoted_string)
```

Explanation:

This code defines a function that converts a string with double quotes to a string enclosed in single quotes and then prints it.

Output:

'This is a string with double quotes'

String Creation Questions - Question 5

Create a multi-line string and count the number of lines in it.

String Creation Questions - Question 5

Code:

```
# Sample code
multi_line_string = '''
This is a
multi-line
string
'''
lines = len(multi_line_string.split('\n'))
print("Number of lines:", lines)
```

Explanation:

This code creates a multi-line string and counts the number of lines in it.

Output:

Number of lines: 5

String Creation Questions - Question 6

Write a program that concatenates three strings: one with single quotes, one with double quotes, and one with triple quotes.

String Creation Questions - Question 6

Code:

```
# Sample code
single_quoted_string = 'Single-quoted string'
double_quoted_string = "Double-quoted string"
triple_quoted_string = """
Triple-quoted string
with multiple lines
"""

result = single_quoted_string + " " + double_quoted_string + "
" + triple_quoted_string
print(result)
```

Explanation:

This code concatenates three strings: one with single quotes, one with double quotes, and one with triple quotes, and then prints the result.

Output:

Single-quoted string Double-quoted string Triple-quoted string
with multiple lines

String Creation Questions - Question 7

Create a string with single quotes and escape a single quote within the string.

String Creation Questions - Question 7

Code:

```
# Sample code
escaped_single_quote_string = 'This is a string with a single
quote: \'escaped\''
print(escaped_single_quote_string)
```

Explanation:

This code creates a string with single quotes and escapes a single quote within the string by using the backslash ('\\').

Output:

This is a string with a single quote: 'escaped'

String Creation Questions - Question 8

Write a script to read a multi-line string from the user input and print it.

sString Creation Questions - Question 8

Code:

```
# Sample code
print("Enter a multi-line string (type 'END' on a new line to
stop):")
user_input = []
while True:
    line = input()
    if line == "END":
        break
    user_input.append(line)
user_input = "\n".join(user_input)
print(user_input)
```

Explanation:

This code prompts the user to enter a multi-line string and then prints the entered string.

Output:

```
Enter a multi-line string (type 'END' on a new line to stop):
Line ONE Line TWO Line THREE END Line ONE Line TWO Line THREE
```

String Creation Questions - Question 9

Create a string using triple quotes and include both single and double quotes inside the string without escaping them.

String Creation Questions - Question 9

Code:

```
# Sample code
mixed_quotes_string = """
This string contains both 'single' and "double" quotes without
escaping them.
"""
print(mixed_quotes_string)
```

Explanation:

This code creates a multi-line string using triple quotes and includes both single and double quotes inside the string without escaping them.

Output:

This string contains both 'single' and "double" quotes without escaping them.

String Creation Questions - Question 10

Write a function that takes a list of words and returns a single string where each word is separated by a newline character, using triple quotes.

String Creation Questions - Question 10

Code:

```
# Sample code
def join_words_with_newline(words):
    result = '\n'.join(words)
    return result

word_list = ['This', 'is', 'a', 'list', 'of', 'words']
joined_string = join_words_with_newline(word_list)
print(joined_string)
```

Explanation:

This code defines a function that takes a list of words and returns a single string where each word is separated by a newline character using triple quotes. It then demonstrates the function with a sample list of words.

Output:

This is a list of words

Basic String Operations Questions - Question 1

Write a function that concatenates two given strings and returns the result.

Basic String Operations - Question 1

Code:

```
def concatenate_strings(str1, str2):
    concatenated_string = str1 + str2
    return concatenated_string

# Example usage:
string1 = "Hello"
string2 = "World"
result = concatenate_strings(string1, string2)
print(result)
```

Explanation:

This code defines a function `concatenate_strings` that takes two strings as input and returns their concatenation using the `+` operator.

Output:

HelloWorld

Basic String Operations - Question 2

Create a program that repeats a given string 5 times and prints the result.

Basic String Operations - Question 2

Code:

```
def repeat_string(input_string, times):
    repeated_string = input_string * times
    return repeated_string

# Example usage:
input_str = "Hello"
result = repeat_string(input_str, 5)
print(result)
```

Explanation:

This code defines a function `repeat_string` that takes a string and the number of times to repeat it. It uses the `*` operator for string repetition.

Output:

HelloHelloHelloHelloHello

Basic String Operations - Question 3

Write a function that takes a string and returns its first and last character concatenated.

Basic String Operations - Question 3

Code:

```
def first_and_last_char(input_string):
    if len(input_string) < 2:
        return input_string
    return input_string[0] + input_string[-1]

# Example usage:
input_str = "Python"
result = first_and_last_char(input_str)
print(result)
```

Explanation:

This code defines a function `first_and_last_char` that takes a string and returns its first and last characters concatenated.

Output:

Pn

Basic String Operations - Question 4

Create a script that slices a string from the 3rd to the 8th character and prints the result.

Basic String Operations - Question 4

Code:

```
input_str = "HelloWorld"
sliced_str = input_str[2:8]
print(sliced_str)
```

Explanation:

This code slices a string from the 3rd to the 8th character (inclusive) and prints the sliced substring.

Output:

lloWor

Basic String Operations - Question 5

Write a program to print the length of a given string.

Basic String Operations - Question 5

Code:

```
input_str = "Hello, World!"  
length = len(input_str)  
print(length)
```

Explanation:

This code calculates and prints the length of a given string using the `len()` function.

Output:

13

Basic String Operations - Question 6

Develop a function that reverses a string using slicing.

Basic String Operations - Question 6

Code:

```
def reverse_string(input_string):
    reversed_str = input_string[::-1]
    return reversed_str

# Example usage:
input_str = "Hello"
result = reverse_string(input_str)
print(result)
```

Explanation:

This code defines a function `reverse_string` that reverses a string using slicing with a step of -1.

Output:

olleH

Basic String Operations - Question 7

Create a script that takes a string and a number n and returns the nth character of the string.

Basic String Operations - Question 7

Code:

```
def nth_character(input_string, n):
    if n < len(input_string):
        return input_string[n]
    else:
        return "Index out of range"

# Example usage:
input_str = "Python"
index = 3
result = nth_character(input_str, index)
print(result)
```

Explanation:

This code defines a function `nth_character` that takes a string and an index `n` and returns the character at that index.

Output:

h

Basic String Operations - Question 8

Write a function that checks if the first and last characters of a string are the same. The function should return True or False.

Basic String Operations - Question 8

Code:

```
def first_and_last_same(input_string):
    if len(input_string) < 2:
        return True
    return input_string[0] == input_string[-1]

# Example usage:
input_str1 = "radar"
input_str2 = "hello"
result1 = first_and_last_same(input_str1)
result2 = first_and_last_same(input_str2)
print(result1) # True
print(result2) # False
```

Explanation:

This code defines a function `first_and_last_same` that checks if the first and last characters of a string are the same and returns True or False accordingly.

Output:

True
False

Basic String Operations - Question 9

Develop a program that swaps the first half and the second half of a string. Assume the string length is even.

Basic String Operations - Question 9

Code:

```
def swap_halves(input_string):
    length = len(input_string)
    half_length = length // 2
    first_half = input_string[:half_length]
    second_half = input_string[half_length:]
    swapped_string = second_half + first_half
    return swapped_string

# Example usage:
input_str = "abcdefgh"
result = swap_halves(input_str)
print(result)
```

Explanation:

This code swaps the first and second halves of a string of even length.

Output:

efghabcd

Basic String Operations - Question 10

Write a script that slices out the middle character(s) of a string. For even length strings, return the middle two characters.

Basic String Operations - Question 10

Code:

```
def slice_middle(input_string):
    length = len(input_string)
    if length % 2 == 0:
        middle_chars = input_string[length // 2 - 1:length // 2 +
1]
    else:
        middle_chars = input_string[length // 2]
    return middle_chars

# Example usage:
input_str1 = "abcde"
input_str2 = "xyz"
result1 = slice_middle(input_str1)
result2 = slice_middle(input_str2)
print(result1) # "c"
print(result2) # "y"
```

Explanation:

This code slices out the middle character(s) of a string, returning the middle two characters for even-length strings.

Output:

c y

String Methods - Question 1

Write a program that converts a given string to lowercase and then to uppercase.

String Methods - Question 1

Code:

```
input_str = "Hello World"
lowercase_str = input_str.lower()
uppercase_str = input_str.upper()
print("Lowercase:", lowercase_str)
print("Uppercase:", uppercase_str)
```

Explanation:

This code converts a given string to both lowercase and uppercase using the `lower()` and `upper()` methods, respectively.

Output:

Lowercase: hello world Uppercase: HELLO WORLD

String Methods - Question 2

Create a function that checks if a string is a valid identifier in Python.

String Methods - Question 2

Code:

```
import re

def is_valid_identifier(input_str):
    if re.match(r'^[a-zA-Z_][a-zA-Z0-9_]*$', input_str):
        return True
    else:
        return False

# Example usage:
identifier1 = "my_variable"
identifier2 = "123_variable"
result1 = is_valid_identifier(identifier1)
result2 = is_valid_identifier(identifier2)
print(result1) # True
print(result2) # False
```

Explanation:

This code defines a function `is_valid_identifier` that checks if a string is a valid Python identifier using a regular expression pattern.

Output:

True False

String Methods - Question 3

Write a script that finds the first occurrence of the substring "Python" in a given string and returns its index.

String Methods - Question 3

Code:

```
input_str = "Python is a popular programming language. Python  
is versatile."  
substring = "Python"  
index = input_str.find(substring)  
print("Index of", substring + ":", index)
```

Explanation:

This code uses the `find()` method to locate the first occurrence of the substring "Python" in the input string and returns its index.

Output:

Index of Python: 0

String Methods - Question 4

Develop a program that replaces all occurrences of the word "Java" with "Python" in a given string.

String Methods - Question 4

Code:

```
input_str = "Java is a popular programming language. Java is versatile."  
output_str = input_str.replace("Java", "Python")  
print(output_str)
```

Explanation:

This code uses the `replace()` method to replace all occurrences of the word "Java" with "Python" in the input string.

Output:

Python is a popular programming language. Python is versatile.

String Methods - Question 5

Create a function that takes a string and returns it with its first letter capitalized and the rest in lowercase.

String Methods - Question 5

Code:

```
def capitalize_first_letter(input_str):
    return input_str.capitalize()

# Example usage:
input_str = "hello world"
result = capitalize_first_letter(input_str)
print(result)
```

Explanation:

This code defines a function `capitalize_first_letter` that capitalizes the first letter of a string and converts the rest to lowercase using the `capitalize()` method.

Output:

Hello world

String Methods - Question 6

Write a script that checks if a given string is a palindrome, ignoring cases.

String Methods - Question 6

Code:

```
def is_palindrome(input_str):
    input_str = input_str.lower()
    return input_str == input_str[::-1]

# Example usage:
string1 = "racecar"
string2 = "Hello"
result1 = is_palindrome(string1)
result2 = is_palindrome(string2)
print(result1) # True
print(result2) # False
```

Explanation:

This code defines a function `is_palindrome` that checks if a string is a palindrome, ignoring cases, by comparing it to its reverse.

Output:

True False

String Methods - Question 7

Develop a function that splits a comma-separated string into a list of strings, then joins them back into a string separated by semicolons.

String Methods - Question 7

Code:

```
def comma_to_semicolon(input_str):
    str_list = input_str.split(',')
    semicolon_str = ';' .join(str_list)
    return semicolon_str

# Example usage:
input_str = "apple,banana,cherry"
result = comma_to_semicolon(input_str)
print(result)
```

Explanation:

This code defines a function `comma_to_semicolon` that first splits a comma-separated string into a list of strings using the `split()` method and then joins them back into a string separated by semicolons using the `join()` method.

Output:

apple;banana;cherry

String Methods - Question 8

Create a program that uses the `strip` method to remove both leading and trailing whitespace from a string.

String Methods - Question 8

Code:

```
input_str = "Hello, World! "
stripped_str = input_str.strip()
print(stripped_str)
```

Explanation:

This code uses the `strip()` method to remove both leading and trailing whitespace from the input string.

Output:

Hello, World!

String Methods - Question 9

Write a function that formats a given string into a predefined template using the `format()` method.

String Methods - Question 9

Code:

```
def format_string(input_str):
    template = "This is a formatted string: {}"
    formatted_str = template.format(input_str)
    return formatted_str

# Example usage:
input_str = "Hello, World!"
result = format_string(input_str)
print(result)
```

Explanation:

This code defines a function `format_string` that formats a given string into a predefined template using the `format()` method.

Output:

This is a formatted string: Hello, World!

String Methods - Question 10

Create a script that encodes a string to UTF-8 and then decodes it back to its original form.

String Methods - Question 10

Code:

```
input_str = "UTF-8 Encoding and Decoding"
encoded_bytes = input_str.encode("utf-8")
decoded_str = encoded_bytes.decode("utf-8")
print("Original String:", input_str)
print("Encoded Bytes:", encoded_bytes)
print("Decoded String:", decoded_str)
```

Explanation:

This code encodes a string to UTF-8 using the `encode()` method and then decodes it back to its original form using the `decode()` method.

Output:

```
Original String: UTF-8 Encoding and Decoding Encoded Bytes:
b'UTF-8 Encoding and Decoding' Decoded String: UTF-8 Encoding
and Decoding
```

String Formatting - Question 1

Write a program that formats a given integer into a string with leading zeros using the old-style `%' formatting.

String Formatting - Question 1

Code:

```
input_integer = 42
formatted_string = "%05d" % input_integer
print("Formatted:", formatted_string)
```

Explanation:

This code uses the old-style `%' formatting to format an integer with leading zeros, resulting in a string with a width of 5 characters.

Output:

Formatted: 00042

String Formatting - Question 2

Create a function that takes two arguments and formats them into a sentence using the `format()` method.

String Formatting - Question 2

Code:

```
def format_sentence(arg1, arg2):
    sentence = "This is an example: {} and {}"
    formatted_sentence = sentence.format(arg1, arg2)
    return formatted_sentence

# Example usage:
result = format_sentence("apple", "banana")
print(result)
```

Explanation:

This code defines a function `format_sentence` that takes two arguments and formats them into a sentence using the ` `.format()` method.

Output:

This is an example: apple and banana

String Formatting - Question 3

Use an f-string to embed a variable into a string and print the result.

String Formatting - Question 3

Code:

```
variable = "world"
greeting = f"Hello, {variable}!"
print(greeting)
```

Explanation:

This code uses an f-string to embed the value of the `variable` into a string and then prints the result.

Output:

Hello, world!

String Formatting - Question 4

Develop a script that formats a floating-point number to two decimal places using the old-style '%' formatting.

String Formatting - Question 4

Code:

```
floating_point_value = 3.14159265
formatted_float = "%.2f" % floating_point_value
print("Formatted:", formatted_float)
```

Explanation:

This code uses the old-style ` `%` formatting to format a floating-point number to two decimal places.

Output:

Formatted: 3.14

String Formatting - Question 5

Write a function that creates a formatted string displaying a date, using the format 'YYYY-MM-DD', with the `format()` method.

String Formatting - Question 5

Code:

```
def format_date(year, month, day):
    formatted_date = "{}-{:02d}-{:02d}".format(year, month,
day)
    return formatted_date

# Example usage:
result = format_date(2023, 12, 17)
print(result)
```

Explanation:

This code defines a function `format_date` that creates a formatted date string in the 'YYYY-MM-DD' format using the `format()` method.

Output:

2023-12-17

String Formatting - Question 6

Create a program that uses an f-string to include a calculated result (like $5 * 7$) within a string.

String Formatting - Question 6

Code:

```
result = 5 * 7
message = f"The result of 5 * 7 is {result}"
print(message)
```

Explanation:

This code uses an f-string to include the calculated result of $5 * 7$ within a string and then prints the message.

Output:

The result of $5 * 7$ is 35

String Formatting - Question 7

Use `%` formatting to align text to the right in a fixed-width string.

String Formatting - Question 7

Code:

```
text = "Right Aligned"
formatted_text = "{:>20}".format(text)
visible_spaces_text = formatted_text.replace(" ", "_")
print(visible_spaces_text)
```

Explanation:

This code uses `{:>20}` formatting to align the text "Right Aligned" to the right within a fixed-width string of 20 characters.

Output:

_____Right_Aligned

String Formatting - Question 8

Develop a function using `format()` that takes a list of numbers and returns a comma-separated string of these numbers.

String Formatting - Question 8

Code:

```
def format_numbers(numbers):
    formatted_numbers = ", ".join(map(str, numbers))
    return formatted_numbers

# Example usage:
numbers_list = [1, 2, 3, 4, 5]
result = format_numbers(numbers_list)
print(result)
```

Explanation:

This code defines a function `format_numbers` that takes a list of numbers, converts them to strings, and returns a comma-separated string using the `join()` method.

Output:

1, 2, 3, 4, 5

String Formatting - Question 9

Write a script using an f-string to format a dictionary's key-value pairs into a nicely aligned output.

String Formatting - Question 9

Code:

```
student_info = {  
    "Name": "John Doe",  
    "Age": 25,  
    "Grade": "A"  
}  
formatted_info = f"Student Information:\nName: {student_info['Name']}\\nAge: {student_info['Age']}\\nGrade: {student_info['Grade']}"  
print(formatted_info)
```

Explanation:

This code uses an f-string to format a dictionary's key-value pairs into a nicely aligned output, displaying student information.

Output:

Student Information: Name: John Doe Age: 25 Grade: A

String Formatting - Question 10

Create a script that encodes a string to UTF-8 and then decodes it back to its original form.

String Formatting - Question 10

Code:

```
input_str = "UTF-8 Encoding and Decoding"
encoded_bytes = input_str.encode("utf-8")
decoded_str = encoded_bytes.decode("utf-8")
print("Original String:", input_str)
print("Encoded Bytes:", encoded_bytes)
print("Decoded String:", decoded_str)
```

Explanation:

This code encodes a string to UTF-8 using the `encode()` method and then decodes it back to its original form using the `decode()` method.

Output:

```
Original String: UTF-8 Encoding and Decoding Encoded Bytes:
b'UTF-8 Encoding and Decoding' Decoded String: UTF-8 Encoding
and Decoding
```

Escape Sequences - Question 1

Write a program that prints a string with a newline character in the middle.

Escape Sequences - Question 1

Code:

```
print("Hello,\nWorld!")
```

Explanation:

This code prints a string with a newline character ("\n") in the middle, causing the text to be displayed on two separate lines.

Output:

```
Hello, World!
```

Escape Sequences - Question 2

Create a script that outputs a tab-separated list of numbers from 1 to 5.

Escape Sequences - Question 2

Code:

```
for i in range(1, 6):
print(i, end="\t")
print()
```

Explanation:

This script prints a tab-separated list of numbers from 1 to 5, using the `end` parameter to specify the tab character ("\t") at the end of each print statement.

Output:

```
1 2 3 4 5
```

Escape Sequences - Question 3

Develop a function that adds a backslash before every double quote in a given string.

Escape Sequences - Question 3

Code:

```
def add_backslash_to_quotes(input_str):
    modified_str = input_str.replace('"', '\\"')
    return modified_str

# Example usage:
input_str = 'This is a "quoted" string.'
result = add_backslash_to_quotes(input_str)
print(result)
```

Explanation:

This code defines a function `add_backslash_to_quotes` that adds a backslash before every double quote in a given string using the `replace()` method.

Output:

This is a \"quoted\" string.

Escape Sequences - Question 4

Write a program that prints a file path using escape sequences to handle backslashes.

Escape Sequences - Question 4

Code:

```
file_path = "C:\\\\Users\\\\Username\\\\Documents\\\\file.txt"
print(file_path)
```

Explanation:

This code prints a file path where double backslashes ("\\") are used as escape sequences to handle backslashes in the path.

Output:

C:\\\\Users\\\\Username\\\\Documents\\\\file.txt

Escape Sequences - Question 5

Create a multi-line string that uses escape sequences to start each new line with a tab.

Escape Sequences - Question 5

Code:

```
multi_line_string = "Line 1\n\tLine 2\n\t\tLine 3"
print(multi_line_string)
```

Explanation:

This code creates a multi-line string where escape sequences are used to start each new line with a tab character ("\t").

Output:

Line 1 Line 2 Line 3

Escape Sequences - Question 6

Write a script that prints a string where newlines are converted to the literal string "\n".

Escape Sequences - Question 6

Code:

```
text = "Line 1\nLine 2\nLine 3"
print(text.replace('\n', '\\n'))
```

Explanation:

This script prints a string where newline characters ("\\n") are replaced with the literal string "\\n" using the `replace()` method.

Output:

```
Line 1\\nLine 2\\nLine 3
```

Escape Sequences - Question 7

Develop a function that takes a string and prints it as raw text, ignoring any escape sequences it contains.

Escape Sequences - Question 7

Code:

```
def print_raw_text(input_str):
    print(repr(input_str))

    # Example usage:
input_str = 'This is a\\nraw\\ttext.'
print_raw_text(input_str)
```

Explanation:

This code defines a function `print_raw_text` that prints a string as raw text, preserving escape sequences by using `repr()`.

Output:

```
'This is a\\nraw\\ttext.'
```

Escape Sequences - Question 8

Create a program that uses escape sequences to print a string inside single quotes.

Escape Sequences - Question 8

Code:

```
text = 'This is a string with single quotes.'  
print(f'\'{text}\''')
```

Explanation:

This program uses escape sequences to print a string inside single quotes by including the single quotes in the f-string.

Output:

```
'This is a string with single quotes.'
```

Escape Sequences - Question 9

Write a function that converts all tab characters in a string to four spaces, without using the replace method.

Escape Sequences - Question 9

Code:

```
def convert_tabs_to_spaces(input_str):
    tab_size = 4
    output_str = ''
    for char in input_str:
        if char == '\t':
            output_str += ' ' * tab_size
        else:
            output_str += char
    return output_str

# Example usage:
input_str = 'Tab\tSeparated\tText'
result = convert_tabs_to_spaces(input_str)
print(result)
```

Explanation:

This code defines a function `convert_tabs_to_spaces` that converts all tab characters ("\t") in a string to four spaces without using the `replace()` method.

Output:

Tab Separated Text

Escape Sequences - Question 10

Develop a script that prints a block of text where each line is separated by a newline escape sequence.

Escape Sequences - Question 10

Code:

```
text_block = "Line 1\nLine 2\nLine 3"  
print(text_block)
```

Explanation:

This script prints a block of text where each line is separated by a newline escape sequence ("\n").

Output:

Line 1 Line 2 Line 3

Unicode and Byte Strings - Question 1

Write a program that converts a Unicode string to a byte string and then back to a Unicode string.

Unicode and Byte Strings - Question 1

Code:

```
# Unicode to byte string
unicode_str = "Hello, World!"
byte_str = unicode_str.encode('utf-8')

# Byte string to Unicode
decoded_str = byte_str.decode('utf-8')

print(decoded_str)
```

Explanation:

This program converts a Unicode string to a byte string using UTF-8 encoding and then back to a Unicode string by decoding it.

Output:

Hello, World!

Unicode and Byte Strings - Question 2

Create a Unicode string containing characters from multiple languages and print it.

Unicode and Byte Strings - Question 2

Code:

```
import html

unicode_str = "你好, Hello, Привет, مرحبا"
decoded_str = html.unescape(unicode_str)
print(decoded_str)
```

Explanation:

This code creates a Unicode string that contains characters from multiple languages and prints it.

Output:

你好, Hello, Привет, مرحبا

Unicode and Byte Strings - Question 3

Develop a function that takes a byte string, decodes it using UTF-8, and handles any decoding errors.

Unicode and Byte Strings - Question 3

Code:

```
def decode_utf8(byte_str):
    try:
        decoded_str = byte_str.decode('utf-8')
        return decoded_str
    except UnicodeDecodeError as e:
        return f"Decoding Error: {e}"

# Example usage:
byte_str = b'Hello, \xe4\xb8\x96\xe7\x95\x8c!'
result = decode_utf8(byte_str)
print(result)
```

Explanation:

This code defines a function `decode_utf8` that takes a byte string, decodes it using UTF-8, and handles any decoding errors using a try-except block.

Output:

Hello, 世界!

Unicode and Byte Strings - Question 4

Write a script that concatenates a Unicode string and a byte string after proper encoding.

Unicode and Byte Strings - Question 4

Code:

```
unicode_str = "Hello, "
byte_str = b"World!"
concatenated_str = unicode_str + byte_str.decode('utf-8')
print(concatenated_str)
```

Explanation:

This script concatenates a Unicode string and a byte string after properly encoding the byte string to Unicode using UTF-8.

Output:

Hello, World!

Unicode and Byte Strings - Question 5

Create a program that reads text from a file as a byte string and then prints it as a Unicode string.

Unicode and Byte Strings - Question 5

Code:

```
# Write a byte string to a file (for demonstration)
with open('sample.txt', 'wb') as file:
    byte_str = b'Hello, World!'
    file.write(byte_str)

# Read and print as Unicode string
with open('sample.txt', 'rb') as file:
    byte_str = file.read()
    unicode_str = byte_str.decode('utf-8')
print(unicode_str)
```

Explanation:

This program demonstrates reading text from a file as a byte string and then printing it as a Unicode string after decoding using UTF-8.

Output:

output sample.txt:Hello, World! screen output:Hello, World!

Unicode and Byte Strings - Question 6

Write a function that takes a list of Unicode strings and returns a single byte string after encoding each string with UTF-8.

Unicode and Byte Strings - Question 6

Code:

```
import html

def encode_utf8(strings):
    byte_str = b''
    for string in strings:
        decoded_string = html.unescape(string) # Decode HTML
entities
        byte_str += decoded_string.encode('utf-8') # Encode as
UTF-8
    return byte_str

# Example usage:
unicode_strings = ["Hello, ", "世界!", ";Hola, mundo!"]
result = encode_utf8(unicode_strings)
print(result)
```

Explanation:

This code defines a function `encode_utf8` that takes a list of Unicode strings and returns a single byte string after encoding each string with UTF-8.

Output:

```
b'Hello, \xe4\xb8\x96\xe7\x95\x8c!\xc2\xa1Hola, mundo!'
```

Unicode and Byte Strings - Question 7

Develop a script that prints the hexadecimal representation of a given byte string.

Unicode and Byte Strings - Question 7

Code:

```
byte_str = b"Hello, World!"  
hex_representation = ''.join([f'{byte:02X}' for byte in  
byte_str])  
print(hex_representation)
```

Explanation:

This script prints the hexadecimal representation of a given byte string by converting each byte to its hexadecimal form.

Output:

48656C6C6F2C20576F726C6421

Unicode and Byte Strings - Question 8

Create a Unicode string with emoji characters and encode it to a byte string.

Unicode and Byte Strings - Question 8

Code:

```
unicode_str = "\udc80\udc81\udc82"
byte_str = unicode_str.encode('utf-8')
print(byte_str)
```

Explanation:

This code creates a Unicode string with emoji characters and encodes it to a byte string using UTF-8 encoding.

Output:

```
b'\udc80\udc81\udc82'
```

Unicode and Byte Strings - Question 9

Write a program that demonstrates the difference in length between a Unicode string and its UTF-8 encoded byte string.

Unicode and Byte Strings - Question 9

Code:

```
unicode_str = "Hello, World!"  
byte_str = unicode_str.encode('utf-8')  
  
unicode_length = len(unicode_str)  
byte_length = len(byte_str)  
  
print(f"Unicode Length: {unicode_length} characters")  
print(f"Byte String Length: {byte_length} bytes")
```

Explanation:

This program demonstrates the difference in length between a Unicode string and its UTF-8 encoded byte string by calculating and printing their respective lengths.

Output:

Unicode Length: 13 characters Byte String Length: 13 bytes

Unicode and Byte Strings - Question 10

Develop a function that checks if a given byte string is valid UTF-8 without decoding it.

Unicode and Byte Strings - Question 10

Code:

```
def is_valid_utf8(byte_string):
    try:
        byte_string.decode('utf-8')
        return True
    except UnicodeDecodeError:
        return False

# Example usage:
byte_string = b'This is a valid UTF-8 byte string.'
result = is_valid_utf8(byte_string)
print(result)
```

Explanation:

This code defines a function `is_valid_utf8` that checks if a given byte string is valid UTF-8 without decoding it by attempting to decode it using 'utf-8' encoding and catching `UnicodeDecodeError` if it's not valid.

Output:

True

Regular Expressions (RegEx) - Question 1

Write a function that uses a regular expression to check if a string contains only letters and spaces.

Regular Expressions (RegEx) - Question 1

Code:

```
import re

def contains_only_letters_and_spaces(input_str):
    pattern = r'^[A-Za-z\s]*$'
    return bool(re.match(pattern, input_str))

# Example usage:
input_str = 'Hello World'
result = contains_only_letters_and_spaces(input_str)
print(result)
```

Explanation:

This code defines a function `contains_only_letters_and_spaces` that uses a regular expression to check if a string contains only letters and spaces.

Output:

True

Regular Expressions (RegEx) - Question 2

Create a script that finds all email addresses in a given string using regular expressions.

Regular Expressions (RegEx) - Question 2

Code:

```
import re

def find_email_addresses(input_str):
    pattern = r'\b[A-Za-z0-9._%+-]+\@[A-Za-z0-9.-]+\.\w{2,7}\b'
    return re.findall(pattern, input_str)

# Example usage:
input_str = 'Emails: john@example.com, alice@gmail.com'
result = find_email_addresses(input_str)
print(result)
```

Explanation:

This script finds all email addresses in a given string using a regular expression pattern for common email formats.

Output:

```
['john@example.com', 'alice@gmail.com']
```

Regular Expressions (RegEx)- Question 3

Develop a program that splits a string at every occurrence of a digit using a regular expression.

Regular Expressions (RegEx) - Question 3

Code:

```
import re

def split_string_at_digits(input_str):
    pattern = r'\d'
    return re.split(pattern, input_str)

# Example usage:
input_str = 'abc123def456'
result = split_string_at_digits(input_str)
print(result)
```

Explanation:

This program splits a string at every occurrence of a digit using a regular expression pattern for digits.

Output:

```
['abc', '', '', 'def', '', '', '']
```

Regular Expressions (RegEx) - Question 4

Write a regular expression to validate a phone number format (e.g., +1-555-123-4567).

Regular Expressions (RegEx) - Question 4

Code:

```
import re

def validate_phone_number(phone_number):
    pattern = r'^\+\d{1,3}-\d{3}-\d{3}-\d{4}$'
    return bool(re.match(pattern, phone_number))

# Example usage:
phone_number = '+1-555-123-4567'
result = validate_phone_number(phone_number)
print(result)
```

Explanation:

This code defines a regular expression pattern to validate phone number formats in the form of "+1-555-123-4567".

Output:

True

Regular Expressions (RegEx) - Question 5

Create a function that extracts the domain name from a URL in a given string.

Regular Expressions (RegEx) - Question 5

Code:

```
import re

def extract_domain_name(url):
    pattern = r'^(https?://)?(www\d?\.)?(?P[\w.-]+)'
    match = re.search(pattern, url)
    if match:
        return match.group('name')
    return None

# Example usage:
url = 'https://www.example.com/path/to/page'
result = extract_domain_name(url)
print(result)
```

Explanation:

This code defines a function `extract_domain_name` that extracts the domain name from a URL using a regular expression pattern.

Output:

example.com

Regular Expressions (RegEx) - Question 6

Write a script that replaces all occurrences of a specific word in a string with another word, using regular expressions.

Regular Expressions (RegEx) - Question 6

Code:

```
import re

def replace_word(input_str, old_word, new_word):
    pattern = r'\b' + re.escape(old_word) + r'\b'
    return re.sub(pattern, new_word, input_str)

# Example usage:
input_str = 'The quick brown fox jumps over the lazy dog.'
old_word = 'fox'
new_word = 'cat'
result = replace_word(input_str, old_word, new_word)
print(result)
```

Explanation:

This script replaces all occurrences of a specific word in a string with another word using a regular expression pattern.

Output:

The quick brown cat jumps over the lazy dog.

Regular Expressions (RegEx) - Question 7

Develop a regular expression to find dates in the format 'YYYY-MM-DD' in a string.

Regular Expressions (RegEx) - Question 7

Code:

```
import re

def find_dates(input_str):
    pattern = r'\b\d{4}-\d{2}-\d{2}\b'
    return re.findall(pattern, input_str)

# Example usage:
input_str = 'Dates: 2022-01-01 and 2023-12-31'
result = find_dates(input_str)
print(result)
```

Explanation:

This code defines a regular expression pattern to find dates in the format 'YYYY-MM-DD' in a string.

Output:

```
[ '2022-01-01', '2023-12-31' ]
```

Regular Expressions (RegEx) - Question 8

Create a function that checks if a string is a valid IPv4 address using regular expressions.

Regular Expressions (RegEx) - Question 8

Code:

```
import re

def is_valid_ipv4(ip_address):
    pattern = r'^\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b'
    return bool(re.match(pattern, ip_address))

# Example usage:
ip_address = '192.168.0.1'
result = is_valid_ipv4(ip_address)
print(result)
```

Explanation:

This code defines a function `is_valid_ipv4` that checks if a string is a valid IPv4 address using a regular expression pattern.

Output:

True

Regular Expressions (RegEx) - Question 9

Write a program that uses regular expressions to count the number of words in a string.

Regular Expressions (RegEx) - Question 9

Code:

```
import re

def count_words(input_string):
    words = re.findall(r'\b\w+\b', input_string)
    return len(words)

# Example usage:
input_string = 'This is a sample sentence with words.'
result = count_words(input_string)
print(result)
```

Explanation:

This code defines a function `count_words` that uses regular expressions to count the number of words in a string by searching for word boundaries and extracting words.

Output:

7

Regular Expressions (RegEx) - Question 10

Develop a script that finds all words that start with 'a' and end with 'e' in a given string.

Regular Expressions (RegEx) - Question 10

Code:

```
import re

def find_words_starting_with_a_ending_with_e(input_string):
    words = re.findall(r'\b[aA]\w*?e\b', input_string)
    return words

# Example usage:
input_string = 'Apple and orange are fruits, but bike is not.'
result = find_words_starting_with_a_ending_with_e(input_string)
print(result)
```

Explanation:

This script finds all words in a given string that start with 'a' (case-insensitive) and end with 'e' using regular expressions and the `\b` word boundary specifier.

Output:

```
['Apple', 'are']
```

String Constants - Question 1

Write a program that creates a string of all lowercase and uppercase alphabets using `string` constants.

String Constants - Question 1

Code:

```
import string

alphabet_string      =      string.ascii_lowercase      +
string.ascii_uppercase
print(alphabet_string)
```

Explanation:

This code uses the `string` module to create a string containing all lowercase and uppercase alphabets.

Output:

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
```

String Constants - Question 2

Create a function that generates a random 8-character password containing letters and digits.

String Constants - Question 2

Code:

```
import string
import random

def generate_random_password():
    characters = string.ascii_letters + string.digits
    password = ''.join(random.choice(characters) for _ in range(8))
    return password

# Example usage:
random_password = generate_random_password()
print(random_password)
```

Explanation:

This code defines a function `generate_random_password` that generates a random 8-character password containing letters and digits using the `string` module and `random` module.

Output:

ZToEPJiH

String Constants - Question 3

Develop a script that checks if a given string contains any punctuation characters.

String Constants - Question 3

Code:

```
import string

def contains_punctuation(input_string):
    punctuation_chars = set(string.punctuation)
        return any(char in punctuation_chars for char in
input_string)

# Example usage:
input_string = 'This is a string with punctuation: ?, !'
result = contains_punctuation(input_string)
print(result)
```

Explanation:

This script checks if a given string contains any punctuation characters by comparing each character with the set of punctuation characters from the `string` module.

Output:

True

String Constants - Question 4

Write a function that counts the number of digits in a given string.

String Constants - Question 4

Code:

```
import string

def count_digits(input_string):
    return sum(1 for char in input_string if char in
string.digits)

# Example usage:
input_string = 'There are 3 digits in this string.'
result = count_digits(input_string)
print(result)
```

Explanation:

This code defines a function `count_digits` that counts the number of digits in a given string by iterating through each character and checking if it's a digit using the `string` module.

Output:

1

String Constants - Question 5

Create a program that removes all whitespace characters from a string using `string` constants.

String Constants - Question 5

Code:

```
import string

def remove_whitespace(input_string):
    return ''.join(char for char in input_string if char not in
string.whitespace)

# Example usage:
input_string = 'This is a string with spaces and\ttabs.'
result = remove_whitespace(input_string)
print(result)
```

Explanation:

This program removes all whitespace characters from a given string using the `string` module's `whitespace` constant.

Output:

Thisisastrinwithspacesandtabs.

String Constants- Question 6

Write a script that generates a string of 10 random alphabetic characters.

String Constants - Question 6

Code:

```
import string
import random

def generate_random_alphabetic_string(length=10):
    characters = string.ascii_letters
        return ''.join(random.choice(characters) for _ in range(length))

# Example usage:
random_string = generate_random_alphabetic_string()
print(random_string)
```

Explanation:

This script generates a string of 10 random alphabetic characters using the `string` module's `ascii_letters` constant and the `random` module.

Output:

MaxWjpjldh

String Constants - Question 7

Develop a function that takes a string and returns a new string with only its alphabetic characters, preserving their case.

String Constants - Question 7

Code:

```
import string

def extract_alphabetic_characters(input_string):
    return ''.join(char for char in input_string if
char.isalpha())

# Example usage:
input_string = '123 This is a Test 456'
result = extract_alphabetic_characters(input_string)
print(result)
```

Explanation:

This code defines a function `extract_alphabetic_characters` that extracts and preserves the case of alphabetic characters from a given string using the `string` module's `isalpha()` method.

Output:

ThisisaTest

String Constants - Question 8

Create a program that checks if a string is composed only of ASCII characters.

String Constants - Question 8

Code:

```
def is_ascii(input_string):
    try:
        input_string.encode(encoding='utf-8').decode('ascii')
    except UnicodeDecodeError:
        return False
    return True

# Example usage:
input_string = 'This is a string with ASCII characters.'
result = is_ascii(input_string)
print(result)
```

Explanation:

This program checks if a given string is composed only of ASCII characters by attempting to encode and decode it using the 'utf-8' encoding and catching `UnicodeDecodeError` if it's not ASCII.

Output:

True

String Constants - Question 9

Write a function that replaces every alphabetic character in a string with a dash ('-').

String Constants - Question 9

Code:

```
import string

def replace_alphabetic_with_dash(input_string):
    return ''.join(['-' if char.isalpha() else char for char in
input_string])

# Example usage:
input_string = 'This is a test string with letters and digits:
123'
result = replace_alphabetic_with_dash(input_string)
print(result)
```

Explanation:

This code defines a function `replace_alphabetic_with_dash` that replaces every alphabetic character in a given string with a dash ('-') while preserving other characters.

Output:

```
-----: 123
```

String Constants - Question 10

Develop a script that creates a string of all hexadecimal digits in both uppercase and lowercase.

String Constants - Question 10

Code:

```
import string

hexadecimal_string = string.hexdigits
print(hexadecimal_string)
```

Explanation:

This script creates a string containing all hexadecimal digits in both uppercase and lowercase using the `string` module's `hexdigits` constant.

Output:

```
0123456789abcdefABCDEF
```

String Iteration and Comprehensions - Question 1

Write a program that iterates over a string and prints each character on a new line.

String Iteration and Comprehensions - Question 1

Code:

```
input_string = 'Hello, World!'
for char in input_string:
    print(char)
```

Explanation:

This program iterates over a string and prints each character on a new line using a `for` loop.

Output:

```
Hello, World!
```

String Iteration and Comprehensions - Question 2

Create a function that returns a list of the ASCII values of each character in a given string.

String Iteration and Comprehensions - Question 2

Code:

```
def ascii_values(input_string):
    return [ord(char) for char in input_string]

# Example usage:
input_string = 'Hello, World!'
result = ascii_values(input_string)
print(result)
```

Explanation:

This code defines a function `ascii_values` that returns a list of ASCII values of each character in a given string using a list comprehension and the `ord()` function.

Output:

```
[72, 101, 108, 108, 111, 44, 32, 87, 111, 114, 108, 100, 33]
```

String Iteration and Comprehensions - Question 3

Develop a script that uses a list comprehension to create a list of tuples, where each tuple is a character and its index in the string.

String Iteration and Comprehensions - Question 3

Code:

```
input_string = 'Hello, World!'
char_index_pairs = [(char, index) for index, char in
enumerate(input_string)]

# Example usage:
print(char_index_pairs)
```

Explanation:

This script uses a list comprehension to create a list of tuples, where each tuple contains a character and its index in the string using `enumerate()`.

Output:

```
[('H', 0), ('e', 1), ('l', 2), ('l', 3), ('o', 4), (',', 5), (' ', 6),
('W', 7), ('o', 8), ('r', 9), ('l', 10), ('d', 11), ('!', 12)]
```

String Iteration and Comprehensions - Question 4

Write a function that iterates over a string and counts the occurrence of a specified character.

String Iteration and Comprehensions - Question 4

Code:

```
def count_occurrence(input_string, target_char):
    count = 0
    for char in input_string:
        if char == target_char:
            count += 1
    return count

# Example usage:
input_string = 'Hello, World!'
target_char = 'o'
result = count_occurrence(input_string, target_char)
print(result)
```

Explanation:

This code defines a function `count_occurrence` that iterates over a string and counts the occurrence of a specified character.

Output:

2

String Iteration and Comprehensions - Question 5

Create a list comprehension that filters out all vowels from a given string.

String Iteration and Comprehensions - Question 5

Code:

```
def remove_vowels(input_string):
    vowels = 'AEIOUaeiou'
    return ''.join(char for char in input_string if char not in
vowels)

# Example usage:
input_string = 'Hello, World!'
result = remove_vowels(input_string)
print(result)
```

Explanation:

This code defines a function `remove_vowels` that uses a list comprehension to filter out all vowels from a given string.

Output:

Hll, Wrld!

String Iteration and Comprehensions - Question 6

Develop a program that iterates through a string and builds a new string with every second character.

String Iteration and Comprehensions - Question 6

Code:

```
input_string = 'Hello, World!'
new_string = input_string[1::2]

# Example usage:
print(new_string)
```

Explanation:

This program iterates through a string and builds a new string with every second character using slicing.

Output:

el,Wrd

String Iteration and Comprehensions - Question 7

Write a script that uses a list comprehension to convert a string into a list of its words in reverse order.

String Iteration and Comprehensions - Question 7

Code:

```
input_string = 'Hello, World!'
word_list = input_string.split()
reversed_word_list = word_list[::-1]

# Example usage:
print(reversed_word_list)
```

Explanation:

This script uses a list comprehension to convert a string into a list of its words in reverse order by splitting the string into words and reversing the list.

Output:

```
['World!', 'Hello,']
```

String Iteration and Comprehensions - Question 8

Create a function that takes a string and returns a list of its characters, excluding any whitespace.

String Iteration and Comprehensions - Question 8

Code:

```
def extract_characters(input_string):
    return [char for char in input_string if not
char.isspace()]

# Example usage:
input_string = 'Hello, World!'
result = extract_characters(input_string)
print(result)
```

Explanation:

This code defines a function `extract_characters` that returns a list of characters from a string, excluding any whitespace characters, using a list comprehension.

Output:

```
['H', 'e', 'l', 'l', 'o', ',', 'W', 'o', 'r', 'l', 'd', '!']
```

String Iteration and Comprehensions - Question 9

Write a program that iterates over a string and prints the index of the first occurrence of each character.

String Iteration and Comprehensions - Question 9

Code:

```
input_string = 'Hello, World!'
char_index_dict = {}

for index, char in enumerate(input_string):
    if char not in char_index_dict:
        char_index_dict[char] = index

# Example usage:
print(char_index_dict)
```

Explanation:

This program iterates over a string and prints the index of the first occurrence of each character in the form of a dictionary.

Output:

```
{'H': 0, 'e': 1, 'l': 2, 'o': 4, ',': 5, ' ': 6, 'W': 7, 'r': 9, 'd': 11, '!': 12}
```

String Iteration and Comprehensions - Question 10

Develop a script that uses a list comprehension to create a list of booleans, where each element is True if the corresponding character in a string is a digit.

String Iteration and Comprehensions - Question 10

Code:

```
input_string = 'Hello, 12345 World!'
is_digit_list = [char.isdigit() for char in input_string]

# Example usage:
print(is_digit_list)
```

Explanation:

This script uses a list comprehension to create a list of booleans, where each element is `True` if the corresponding character in a string is a digit, using the `isdigit()` method.

Output:

```
[False, False, False, False, False, False, False, True, True,
True, True, True, False, False, False, False, False, False]
```

String Conversion - Question 1

Write a function that converts a list of integers into a comma-separated string.

String Conversion - Question 1

Code:

```
def list_to_comma_separated_string(int_list):
    return ','.join(map(str, int_list))

# Example usage:
integers = [1, 2, 3, 4, 5]
result = list_to_comma_separated_string(integers)
print(result)
```

Explanation:

This code defines a function `list_to_comma_separated_string` that converts a list of integers into a comma-separated string using `join()` and `map()`.

Output:

1,2,3,4,5

String Conversion - Question 2

Create a program that converts a float to a string and prints it with two decimal places.

String Conversion - Question 2

Code:

```
float_value = 123.456789
formatted_float = "{:.2f}".format(float_value)

# Example usage:
print(formatted_float)
```

Explanation:

This program converts a float to a string and prints it with two decimal places using string formatting.

Output:

123.46

String Conversion - Question 3

Develop a script that takes a dictionary and converts it into a string representation.

String Conversion - Question 3

Code:

```
def dictionary_to_string(input_dict):
    return str(input_dict)

# Example usage:
my_dict = {'key1': 'value1', 'key2': 'value2'}
result = dictionary_to_string(my_dict)
print(result)
```

Explanation:

This script takes a dictionary and converts it into a string representation using the `str()` function.

Output:

```
{'key1': 'value1', 'key2': 'value2'}
```

String Conversion - Question 4

Write a function that converts a boolean value (`True` or `False`) into a string.

String Conversion - Question 4

Code:

```
def boolean_to_string(boolean_value):
    return str(boolean_value)

# Example usage:
my_boolean = True
result = boolean_to_string(my_boolean)
print(result)
```

Explanation:

This code defines a function `boolean_to_string` that converts a boolean value into a string using the `str()` function.

Output:

True

String Conversion - Question 5

Create a program that takes an integer and prints it as a binary string.

String Conversion - Question 5

Code:

```
integer_value = 42
binary_string = bin(integer_value)[2:]

# Example usage:
print(binary_string)
```

Explanation:

This program takes an integer and prints it as a binary string using the `bin()` function and slicing to remove the '0b' prefix.

Output:

101010

String Conversion - Question 6

Write a script that converts a tuple into a string with each element separated by a space.

String Conversion - Question 6

Code:

```
my_tuple = (1, 2, 3, 4, 5)
string_representation = ' '.join(map(str, my_tuple))

# Example usage:
print(string_representation)
```

Explanation:

This script converts a tuple into a string with each element separated by a space using `join()` and `map()`.

Output:

```
1 2 3 4 5
```

String Conversion - Question 7

Develop a function that takes a set of numbers and returns a string with each number rounded and separated by commas.

String Conversion - Question 7

Code:

```
def round_numbers_to_string(number_set):
    rounded_numbers = [str(round(num)) for num in number_set]
    return ', '.join(rounded_numbers)

# Example usage:
numbers = {3.5, 4.8, 6.2, 7.9}
result = round_numbers_to_string(numbers)
print(result)
```

Explanation:

This code defines a function `round_numbers_to_string` that takes a set of numbers, rounds each number, and returns a string with rounded numbers separated by commas.

Output:

4, 5, 6, 8

String Conversion - Question 8

Create a program that converts a list of dates (as datetime objects) into a string where dates are separated by a newline character.

String Conversion - Question 8

Code:

```
import datetime

date_list = [datetime.date(2023, 1, 1), datetime.date(2023, 2,
1), datetime.date(2023, 3, 1)]
date_string = '\n'.join(str(date) for date in date_list)

# Example usage:
print(date_string)
```

Explanation:

This program converts a list of dates (as datetime objects) into a string where dates are separated by a newline character using `join()`.

Output:

2023-01-01 2023-02-01 2023-03-01

String Conversion - Question 9

Write a function that takes an array of floating-point numbers and returns a string of these numbers formatted with one decimal place.

String Conversion - Question 9

Code:

```
def format_floats(floating_point_array):
    formatted_numbers = [f"{num:.1f}" for num in
floating_point_array]
    return ', '.join(formatted_numbers)

# Example usage:
floats = [1.234, 2.567, 3.789]
result = format_floats(floats)
print(result)
```

Explanation:

This code defines a function `format_floats` that takes an array of floating-point numbers and returns a string with these numbers formatted with one decimal place.

Output:

1.2, 2.6, 3.8

String Conversion - Question 10

Develop a script that takes an object of a custom class and converts it into a string representation.

String Conversion - Question 10

Code:

```
class MyClass:

    def __init__(self, value):
        self.value = value

    def __str__():
        return f"MyClass({self.value})"

# Example usage:
obj = MyClass(42)
result = str(obj)
print(result)
```

Explanation:

This script defines a custom class `MyClass` with a `__str__` method that returns a string representation of the object.

Output:

MyClass(42)

Immutability of Strings - Question 1

Write a function that tries to modify a character in a given string and handles the resulting error.

Immutability of Strings - Question 1

Code:

```
def modify_string_character(input_string, index, new_char):
    try:
        modified_string = input_string[:index] + new_char +
input_string[index+1:]
        return modified_string
    except IndexError as e:
        return f"Error: {e}"

# Example usage:
input_str = "Hello, World!"
result = modify_string_character(input_str, 7, 'X')
print(result)
```

Explanation:

This function attempts to modify a character in a given string at a specified index. It handles the `IndexError` that may occur if the index is out of range.

Output:

Hello, Xorld!

Immutability of Strings - Question 2

Create a script that demonstrates concatenation of strings and shows that the original strings are not altered.

Immutability of Strings - Question 2

Code:

```
original_str1 = "Hello"
original_str2 = " World"
concatenated_str = original_str1 + original_str2

# Example usage:
print("Concatenated String:", concatenated_str)
print("Original String 1:", original_str1)
print("Original String 2:", original_str2)
```

Explanation:

This script demonstrates concatenation of two strings and shows that the original strings `original_str1` and `original_str2` remain unaltered.

Output:

```
Concatenated String: Hello World Original String 1: Hello
Original String 2: World
```

Immutability of Strings - Question 3

Develop a program that iterates through a string and tries to modify each character, then explain the outcome.

Immutability of Strings - Question 3

Code:

```
input_string = "Immutable"
modified_string = ""

for char in input_string:
    modified_string += 'X'

# Example usage:
print("Modified String:", modified_string)
print("Original String:", input_string)
```

Explanation:

This program iterates through the characters of a string (`input_string`) and attempts to modify each character by appending 'X'. However, due to string immutability, it creates a new string (`modified_string`) with the desired changes, and the original string remains unaltered.

Output:

Modified String: XXXXXXXXX Original String: Immutable

Immutability of Strings - Question 4

Write a function that accepts a string and returns a new string with a character changed, illustrating immutability.

Immutability of Strings - Question 4

Code:

```
def change_character(input_string, index, new_char):
    if 0 <= index < len(input_string):
        return input_string[:index] + new_char + input_string[index+1:]
    return input_string

# Example usage:
input_str = "Immutable"
result = change_character(input_str, 3, 'X')
print(result)
```

Explanation:

This function accepts a string, an index, and a new character. It returns a new string with the specified character changed at the given index while illustrating the immutability of strings.

Output:

ImmXtable

Immutability of Strings - Question 5

Create a program that appends a suffix to a string without altering the original string.

Immutability of Strings - Question 5

Code:

```
original_str = "Hello"
suffix = ", World"
result_str = original_str + suffix

# Example usage:
print("Result String:", result_str)
print("Original String:", original_str)
```

Explanation:

This program appends a suffix to a string (`original_str`) without altering the original string, demonstrating string immutability.

Output:

```
Result String: Hello, World Original String: Hello
```

Immutability of Strings - Question 6

Write a script that merges two strings into a third string and shows that the first two strings remain unchanged.

Immutability of Strings - Question 6

Code:

```
str1 = "Hello"  
str2 = " World"  
merged_str = str1 + str2  
  
# Example usage:  
print("Merged String:", merged_str)  
print("Original String 1:", str1)  
print("Original String 2:", str2)
```

Explanation:

This script merges two strings (`str1` and `str2`) into a third string (`merged_str`) and demonstrates that the original strings remain unchanged, illustrating string immutability.

Output:

```
Merged String: Hello World Original String 1: Hello Original  
String 2: World
```

Immutability of Strings - Question 7

Develop a function that explains the concept of string immutability with a practical example.

Immutability of Strings - Question 7

Code:

```
def explain_string_immutability():
    original_str = "Immutable"
    modified_str = original_str[:3] + "X" + original_str[4:]

    explanation = (
        "String immutability means that once a string is created,  

        it cannot be changed in place."
        "Instead, any modification creates a new string. For  

        example, in the word 'Immutable', "
        "changing the fourth character 'u' to 'X' creates a new  

        string, resulting in 'ImmutaXble'. "
        "The original string 'Immutable' remains unaltered."
    )

    return modified_str, explanation

# Example usage:
result, explanation = explain_string_immutability()
print("Modified String:", result)
print("Explanation:", explanation)
```

Explanation:

This function explains the concept of string immutability with a practical example. It illustrates how modifying a character in a string creates a new string while the original remains unaltered.

Output:

```
Modified String: ImmXtable Explanation: String immutability  

means that once a string is created, it cannot be changed in  

place. Instead, any modification creates a new string. For  

example, in the word 'Immutable', changing the fourth character  

'u' to 'X' creates a new string, resulting in 'ImmutaXble'. The  

original string 'Immutable' remains unaltered.
```

Immutability of Strings - Question 8

Create a program that takes a string and a list of indices and attempts to change the characters at these indices.

Immutability of Strings - Question 8

Code:

```
def modify_string_at_indices(input_string, indices, new_chars):
    modified_string = input_string

    for i, index in enumerate(indices):
        if 0 <= index < len(modified_string) and i <
len(new_chars):
            modified_string = modified_string[:index] +
new_chars[i] + modified_string[index+1:]

    return modified_string

# Example usage:
input_str = "Immutable"
change_indices = [3, 6]
new_characters = ['X', 'Y']
result = modify_string_at_indices(input_str, change_indices,
new_characters)
print(result)
```

Explanation:

This program takes a string, a list of indices, and a list of new characters. It attempts to change the characters at the specified indices and returns the modified string. It demonstrates that modifying a string creates a new string while maintaining immutability.

Output:

ImmXtaYle

Immutability of Strings - Question 9

Write a function that demonstrates how to effectively create a modified version of an immutable string.

Immutability of Strings - Question 9

Code:

```
def create_modified_string(input_string, index, new_char):
    modified_string = list(input_string)

    if 0 <= index < len(modified_string):
        modified_string[index] = new_char

    return ''.join(modified_string)

# Example usage:
input_str = "Immutable"
result = create_modified_string(input_str, 3, 'X')
print(result)
```

Explanation:

This function demonstrates how to effectively create a modified version of an immutable string by converting it to a list of characters, modifying the list, and then joining it back into a string.

Output:

ImmXtable

Immutability of Strings - Question 10

Develop a script that explains the difference in behavior between mutable and immutable data types using strings as an example.

Immutability of Strings- Question 10

Code:

```
# Mutable List
mutable_list = [1, 2, 3]
mutable_list.append(4)
print("Mutable List:", mutable_list)

# Immutable String
immutable_str = "Hello"
new_str = immutable_str + ", World"
print("Immutable String:", immutable_str)
print("New String:", new_str)
```

Explanation:

This script demonstrates the difference in behavior between mutable and immutable data types using a mutable list and an immutable string as examples. The mutable list can be modified in place, while the immutable string requires creating a new string for modifications.

Output:

```
Mutable List: [1, 2, 3, 4] Immutable String: Hello New String:
Hello, World
```

String Slicing and Extended Slicing - Question 1

Write a function that slices a string to extract characters from the 2nd to the 5th index.

String Slicing and Extended Slicing - Question 1

Code:

```
def slice_string(input_string):
    sliced_string = input_string[1:5]
    return sliced_string

# Example usage:
input_str = "Hello, World!"
result = slice_string(input_str)
print(result)
```

Explanation:

This function slices a string to extract characters from the 2nd to the 5th index (0-based indexing) and returns the sliced string.

Output:

ello

String Slicing and Extended Slicing - Question 2

Create a script that reverses a string using slicing.

String Slicing and Extended Slicing - Question 2

Code:

```
def reverse_string(input_string):
    reversed_string = input_string[::-1]
    return reversed_string

# Example usage:
input_str = "Hello, World!"
result = reverse_string(input_str)
print(result)
```

Explanation:

This script reverses a string using slicing by specifying a step of -1, effectively reversing the characters in the string.

Output:

```
!dlroW ,olleH
```

String Slicing and Extended Slicing - Question 3

Develop a program that extracts every third character from a string starting from the end.

String Slicing and Extended Slicing - Question 3

Code:

```
def extract_every_third_character(input_string):
    extracted_string = input_string[-1::-3]
    return extracted_string

# Example usage:
input_str = "Hello, World!"
result = extract_every_third_character(input_str)
print(result)
```

Explanation:

This program extracts every third character from a string starting from the end by specifying a step of -3 in slicing.

Output:

!r lH

String Slicing and Extended Slicing - Question 4

Write a function that slices a string to include only characters at even indices.

String Slicing and Extended Slicing - Question 4

Code:

```
def slice_even_indices(input_string):
    sliced_string = input_string[::2]
    return sliced_string

# Example usage:
input_str = "Hello, World!"
result = slice_even_indices(input_str)
print(result)
```

Explanation:

This function slices a string to include only characters at even indices (0-based indexing) and returns the sliced string.

Output:

Hlo ol!

String Slicing and Extended Slicing - Question 5

Create a program that concatenates the first half and the reversed second half of a string.

String Slicing and Extended Slicing - Question 5

Code:

```
def concatenate_halves(input_string):
    middle_index = len(input_string) // 2
    first_half = input_string[:middle_index]
    second_half = input_string[middle_index:]
    reversed_second_half = second_half[::-1]
    concatenated_string = first_half + reversed_second_half
    return concatenated_string

# Example usage:
input_str = "Hello, World!"
result = concatenate_halves(input_str)
print(result)
```

Explanation:

This program concatenates the first half and the reversed second half of a string. It calculates the middle index, slices the string accordingly, reverses the second half, and concatenates the two halves.

Output:

Hello,!dlrow

String Slicing and Extended Slicing - Question 6

Write a script that uses slicing to check if a string is a palindrome.

String Slicing and Extended Slicing - Question 6

Code:

```
def is_palindrome(input_string):
    reversed_string = input_string[::-1]
    return input_string == reversed_string

# Example usage:
input_str = "racecar"
result = is_palindrome(input_str)
print(result)
```

Explanation:

This script checks if a string is a palindrome using slicing. It reverses the string and compares it to the original string to determine if it's a palindrome.

Output:

True

String Slicing and Extended Slicing - Question 7

Develop a function that slices out and returns the middle third of a string.

String Slicing and Extended Slicing - Question 7

Code:

```
def middle_third(input_string):
    length = len(input_string)
    third = length // 3
    middle_third = input_string[third:2*third]
    return middle_third

# Example usage:
input_str = "abcdefghijklm"
result = middle_third(input_str)
print(result)
```

Explanation:

This function slices out and returns the middle third of a string. It calculates the length of the string, divides it by 3, and slices the middle third.

Output:

cd

String Slicing and Extended Slicing - Question 8

Create a program that takes a string and returns two halves of the string, using slicing.

String Slicing and Extended Slicing - Question 8

Code:

```
def split_string(input_string):
    length = len(input_string)
    middle = length // 2
    first_half = input_string[:middle]
    second_half = input_string[middle:]
    return first_half, second_half

# Example usage:
input_str = "Hello, World!"
result1, result2 = split_string(input_str)
print("First Half:", result1)
print("Second Half:", result2)
```

Explanation:

This program takes a string and returns two halves of the string using slicing. It calculates the middle index and slices the string accordingly.

Output:

First Half: Hello, Second Half: World!

String Slicing and Extended Slicing - Question 9

Write a function that alternates characters in a string (first, third, fifth, etc.), then the remaining characters (second, fourth, sixth, etc.).

String Slicing and Extended Slicing - Question 9

Code:

```
def alternate_characters(input_string):
    odd_chars = input_string[::2]
    even_chars = input_string[1::2]
    alternated_string = ''.join([a + b for a, b in
zip(odd_chars, even_chars)])
    return alternated_string

# Example usage:
input_str = "abcdefgh"
result = alternate_characters(input_str)
print(result)
```

Explanation:

This function alternates characters in a string, first taking the characters at odd indices (1-based indexing) and then taking the characters at even indices. It combines the two sets of characters to create the alternated string.

Output:

abcdefgh

String Slicing and Extended Slicing - Question 10

Develop a script that uses extended slicing to create a new string with every second character from the first half and every third character from the second half.

String Slicing and Extended Slicing - Question 10

Code:

```
def extended_slice_string(input_string):
    length = len(input_string)
    middle = length // 2
        new_string      =      input_string[:middle:2]      +
input_string[middle::3]
    return new_string

# Example usage:
input_str = "abcdefghijklm"
result = extended_slice_string(input_str)
print(result)
```

Explanation:

This script uses extended slicing to create a new string with every second character from the first half and every third character from the second half. It calculates the middle index and slices the string accordingly.

Output:

aceh

String and Memory Interning - Question 1

Write a program that demonstrates string interning by comparing the memory addresses of two identical strings.

String and Memory Interning - Question 1

Code:

```
string1 = "Hello, World!"  
string2 = "Hello, World!"  
  
# Check if the strings share the same memory address  
are_equal = string1 is string2  
print("Strings share memory address:", are_equal)
```

Explanation:

This program demonstrates string interning by comparing the memory addresses of two identical strings. When strings are identical, Python optimizes memory usage by reusing the same memory address.

Output:

Strings share memory address: True

String and Memory Interning - Question 2

Create a function that checks if interning occurs for two strings created separately but with the same content.

String and Memory Interning - Question 2

Code:

```
def check_interning(string1, string2):
    are_equal = string1 is string2
    return are_equal

# Example usage:
str1 = "Hello, World!"
str2 = "Hello, World!"
result = check_interning(str1, str2)
print("Interning Occurs:", result)
```

Explanation:

This function checks if interning occurs for two strings created separately but with the same content. If the strings are interned, it returns True; otherwise, it returns False.

Output:

Interning Occurs: True

String and Memory Interning - Question 3

Develop a script that compares the performance difference between using interned and non-interned strings in a long loop.

String and Memory Interning - Question 3

Code:

```
import time

# Interned strings
interned_strings = ["Hello"] * 1000000

# Non-interned strings
non_interned_strings = ["Hello" + str(i) for i in range(1000000)]

# Measure the time taken to iterate over interned strings
start_time = time.time()
for string in interned_strings:
    pass
end_time = time.time()
interned_time = end_time - start_time

# Measure the time taken to iterate over non-interned strings
start_time = time.time()
for string in non_interned_strings:
    pass
end_time = time.time()
non_interned_time = end_time - start_time

print("Time taken for interned strings:", interned_time)
print("Time taken for non-interned strings:", non_interned_time)
```

Explanation:

This script compares the performance difference between using interned and non-interned strings in a long loop. Interned strings show better performance because they share memory addresses.

Output:

```
Time taken for interned strings: 0.017939090728759766 Time
taken for non-interned strings: 0.018505096435546875
```

String and Memory Interning - Question 4

Write a program to explore the effect of string interning on memory usage with a large number of identical strings.

String and Memory Interning - Question 4

Code:

```
import sys

# Create a large number of identical strings
identical_strings = ["Hello, World!"] * 1000000

# Measure memory usage
memory_usage = sys.getsizeof(identical_strings)

print("Memory Usage (identical strings):", memory_usage,
"bytes")
```

Explanation:

This program explores the effect of string interning on memory usage by creating a large number of identical strings. Interned strings save memory by sharing the same memory address.

Output:

Memory Usage (identical strings): 8000056 bytes

String and Memory Interning - Question 5

Create a function that forcibly interns a string using the `sys.intern()` method and demonstrates its effect.

String and Memory Interning - Question 5

Code:

```
import sys

def force_intern_string(input_string):
    interned_string = sys.intern(input_string)
    return interned_string

# Example usage:
input_str = "Hello, World!"
interned_str = force_intern_string(input_str)
are_equal = input_str is interned_str
print("String Interned:", are_equal)
```

Explanation:

This function forcibly interns a string using the `sys.intern()` method and demonstrates its effect by comparing the memory addresses of the original and interned strings.

Output:

String Interned: True

String and Memory Interning - Question 6

Write a script that creates a large number of similar strings and checks how many of them are interned.

String and Memory Interning - Question 6

Code:

```
import sys

# Create a large number of similar strings
similar_strings = [f"Hello_{i}" for i in range(1000000)]

# Check how many of them are interned
interned_count = sum(sys.intern(string) is string for string in
similar_strings)

print("Number of interned strings:", interned_count)
```

Explanation:

This script creates a large number of similar strings and checks how many of them are interned using the `sys.intern()` method. All of the strings are interned, as they have the same content.

Output:

Number of interned strings: 1000000

String and Memory Interning - Question 7

Develop a program that shows the difference in behavior between mutable and immutable types with respect to interning.

String and Memory Interning - Question 7

Code:

```
import sys

# Immutable string
immutable_string = "Hello, World!"

# Mutable list
mutable_list = ["Hello, World!"]

# Check if the strings are interned
immutable_interned      =      sys.intern(immutable_string)      is
immutable_string
mutable_interned         =      sys.intern(mutable_list[0])      is
mutable_list[0]

print("Immutable String Interned:", immutable_interned)
print("Mutable List Interned:", mutable_interned)
```

Explanation:

This program demonstrates the difference in behavior between mutable and immutable types with respect to interning. Immutable strings can be interned, while mutable lists cannot.

Output:

```
Immutable String Interned: True Mutable List Interned: True
```

String and Memory Interning - Question 8

Create a function that explains when Python decides to automatically intern a string and when it does not.

String and Memory Interning - Question 8

Code:

```
def explain_string_interning():
    explanation = """
        Python automatically interns strings in the following
scenarios:
        1. String literals in the source code are interned by
default.
        2. Strings that are used as attribute names in classes and
objects are interned.
        3. Identifiers (variable names, function names, etc.) used
in the same scope can be interned for optimization.

        Python may not intern strings in the following scenarios:
        1. Strings created dynamically at runtime may not be
interned.
        2. Strings with certain characters or special characters
may not be interned.
    """
    return explanation

# Example usage:
explanation_text = explain_string_interning()
print(explanation_text)
```

Explanation:

This function explains when Python decides to automatically intern a string and when it does not. It covers scenarios where strings are interned by default and scenarios where they may not be interned.

Output:

Python automatically interns strings in the following scenarios: 1. String literals in the source code are interned by default. 2. Strings that are used as attribute names in classes and objects are interned. 3. Identifiers (variable names, function names, etc.) used in the same scope can be interned for optimization. Python may not intern strings in the

following scenarios: 1. Strings created dynamically at runtime may not be interned. 2. Strings with certain characters or special characters may not be interned.

String and Memory Interning - Question 9

Write a script that demonstrates the use of string interning in a real-world scenario, such as processing large text files.

String and Memory Interning - Question 9

Code:

```
import sys

# Read a large text file
with open("large_text_file.txt", "r") as file:
    text = file.read()

# Intern the text
interned_text = sys.intern(text)

# Check if the original and interned text share the same memory
# address
are_equal = text is interned_text

print("Text Interned:", are_equal)
```

Explanation:

This script demonstrates the use of string interning in a real-world scenario by processing a large text file. The text from the file is interned, optimizing memory usage.

Output:

Text Interned: True

String and Memory Interning - Question 10

Develop a program that compares the identity of strings created through concatenation versus direct assignment.

String and Memory Interning - Question 10

Code:

```
string1 = "Hello"
string2 = "World!"
concatenated_string = string1 + ", " + string2
direct_assigned_string = "Hello, World!"

# Check if the concatenated and directly assigned strings share
# memory addresses
concatenated_equals_direct      =      concatenated_string      is
direct_assigned_string

print("Concatenated String Equals Directly Assigned String:", concatenated_equals_direct)
```

Explanation:

This program compares the identity of strings created through concatenation and direct assignment. While identical strings created directly are usually interned, concatenated strings may not be interned, leading to different memory addresses.

Output:

Concatenated String Equals Directly Assigned String: False

Raw Strings - Question 1

Write a program that creates a raw string containing file path information and prints it.

Raw Strings - Question 1

Code:

```
# Create a raw string with file path
raw_path = r'C:\Users\username\Documents\file.txt'

# Print the raw string
print(raw_path)
```

Explanation:

This program creates a raw string containing file path information and prints it. The raw string preserves backslashes as-is.

Output:

C:\Users\username\Documents\file.txt

Raw Strings - Question 2

Create a script using a raw string to define a regular expression pattern that includes backslashes.

Raw Strings - Question 2

Code:

```
import re

# Define a regular expression pattern with a raw string
raw_pattern = r'\d+\.\d+\\'

# Use the raw pattern to search for matches
text = 'Match 123.45\\ and 678.90\\'
matches = re.findall(raw_pattern, text)

print(matches)
```

Explanation:

This script defines a regular expression pattern using a raw string that includes backslashes. It then searches for matches in a given text and prints them.

Output:

```
['123.45\\', '678.90\\']
```

Raw Strings - Question 3

Develop a function that takes a regular expression pattern as a raw string and searches for this pattern in a given text.

Raw Strings - Question 3

Code:

```
import re

def search_pattern(raw_pattern, text):
    matches = re.findall(raw_pattern, text)
    return matches

# Example usage:
pattern = r'\d+\.\d+\\'
input_text = 'Match 123.45\\ and 678.90\\'
result = search_pattern(pattern, input_text)
print(result)
```

Explanation:

This function takes a regular expression pattern as a raw string and searches for this pattern in a given text. It returns a list of matches.

Output:

```
['123.45\\', '678.90\\']
```

Raw Strings - Question 4

Write a raw string that includes newline characters literally and print it to show the effect.

Raw Strings - Question 4

Code:

```
# Create a raw string with newline characters
raw_text = r'This is a raw string.\nIt contains newline
characters.'

# Print the raw string
print(raw_text)
```

Explanation:

This code creates a raw string that includes newline characters literally (as '\n') and prints it. The newline characters are not interpreted as line breaks.

Output:

This is a raw string.\nIt contains newline characters.

Raw Strings - Question 5

Create a program where you define a Windows file path using both a regular string and a raw string, and illustrate the difference.

Raw Strings - Question 5

Code:

```
# Define a Windows file path using a regular string
regular_path = 'C:\\\\Users\\\\username\\\\Documents\\\\file.txt'

# Define the same path using a raw string
raw_path = r'C:\\Users\\username\\Documents\\file.txt'

# Check and illustrate the difference
print("Regular String Path:", regular_path)
print("Raw String Path:", raw_path)
```

Explanation:

This program defines a Windows file path using both a regular string and a raw string. It illustrates that both representations result in the same path.

Output:

```
Regular String Path: C:\\Users\\username\\Documents\\file.txt Raw
String Path: C:\\Users\\username\\Documents\\file.txt
```

Raw Strings - Question 6

Write a script that uses a raw string to avoid escape sequence processing in a string that represents a JSON object.

Raw Strings - Question 6

Code:

```
# Create a JSON object as a string with escape sequences
json_with_escapes = '{"name": "John", "age": 30, "city": "New\nYork"}'

# Create the same JSON object as a raw string to avoid
# interpreting escape sequences
json_raw = r'{"name": "John", "age": 30, "city": "New\nYork"}'

# Check and illustrate the difference
print("JSON with Escapes:", json_with_escapes)
print("JSON as Raw String:", json_raw)
```

Explanation:

This script creates a JSON object as a string with escape sequences and the same JSON object as a raw string to avoid escape sequences. It illustrates the difference in representation.

Output:

```
JSON with Escapes: {"name": "John", "age": 30, "city": "New York"} JSON as Raw String: {"name": "John", "age": 30, "city": "New\nYork"}
```

Raw Strings - Question 7

Develop a function that demonstrates the difference between printing a string and a raw string with tab characters.

Raw Strings - Question 7

Code:

```
def print_strings():
    # Define a string with tab characters
    tab_string = 'This\tis\ta\ttab\tseparated\tstring'

    # Define the same string as a raw string
    tab_raw = r'This\tis\ta\ttab\tseparated\tstring'

    # Print both the string and raw string
    print("String with Tabs:", tab_string)
    print("Raw String with Tabs:", tab_raw)

    # Example usage:
print_strings()
```

Explanation:

This function demonstrates the difference between printing a string and a raw string with tab characters. The string with tabs is formatted with spaces, while the raw string preserves the escape sequence.

Output:

```
String with Tabs: This is a tab separated string Raw String
with Tabs: This\tis\ta\ttab\tseparated\tstring
```

Raw Strings - Question 8

Create a raw string containing special characters and use it in a regular expression search.

Raw Strings - Question 8

Code:

```
import re

# Create a raw string with special characters
raw_pattern = r'[a-z]+[0-9]?\.'

# Search for the raw pattern in a text
text = 'Match abc1. and def2. in the text.'
matches = re.findall(raw_pattern, text)

print(matches)
```

Explanation:

This code creates a raw string containing special characters and uses it in a regular expression search. It matches and returns substrings that fit the pattern.

Output:

```
['abc1.', 'def2.', 'text.']}
```

Raw Strings - Question 9

Write a program that uses raw strings to handle a multi-line string without interpreting escape characters.

Raw Strings - Question 9

Code:

```
# Create a multi-line string with escape sequences
multi_line_string = "Line 1\nLine 2\nLine 3"

# Create the same multi-line string as a raw string to avoid
# escape sequences
raw_multi_line = r"Line 1\nLine 2\nLine 3"

# Check and illustrate the difference
print("Multi-Line String with Escapes:")
print(multi_line_string)
print("Multi-Line String as Raw String:")
print(raw_multi_line)
```

Explanation:

This program creates a multi-line string with escape sequences and the same multi-line string as a raw string to avoid escape sequences. It illustrates the difference in representation.

Output:

```
Multi-Line String with Escapes: Line 1 Line 2 Line 3
Multi-Line String as Raw String: Line 1\nLine 2\nLine 3
```

Raw Strings - Question 10

Develop a script that compares the behavior of a string and a raw string when used in file handling operations.

Raw Strings - Question 10

Code:

```
# Define a string and a raw string for file paths
string_path = 'C:\\\\Users\\\\username\\\\Documents\\\\file.txt'
raw_path = r'C:\\Users\\username\\Documents\\file.txt'

# Create and open a file using both paths
with open(string_path, 'w') as string_file:
    string_file.write('This is a file written using a regular
string path.')

with open(raw_path, 'w') as raw_file:
    raw_file.write('This is a file written using a raw string
path.')

# Illustrate the successful creation of both files
print("File created using Regular String Path:")
with open(string_path, 'r') as string_file:
    print(string_file.read())

print("File created using Raw String Path:")
with open(raw_path, 'r') as raw_file:
    print(raw_file.read())
```

Explanation:

This script compares the behavior of a string and a raw string when used in file handling operations. It successfully creates and reads files using both string representations for file paths.

Output:

```
file.txt: This is a file written using a raw string path.
```

String Exceptions - Question 1

Write a program that catches and handles the exception when converting an invalid string to a number using `int()`.

String Exceptions - Question 1

Code:

```
# Attempt to convert an invalid string to a number
invalid_string = "abc"
try:
    number = int(invalid_string)
except ValueError as e:
    print("ValueError:", e)
    number = None

# Print the result
if number is not None:
    print("Converted Number:", number)
else:
    print("Conversion failed.")
```

Explanation:

This program attempts to convert an invalid string to a number using `int()` and handles the resulting `ValueError` exception.

Output:

```
ERROR! ValueError: invalid literal for int() with base 10:
'abc' Conversion failed.
```

String Exceptions - Question 2

Create a script that tries to access a character out of the string index range and handles the resulting `IndexError`.

String Exceptions - Question 2

Code:

```
# Define a string
text = "Hello, World!"

try:
# Try to access a character out of range
    character = text[15]
except IndexError as e:
    print("IndexError:", e)
    character = None

# Print the result
if character is not None:
    print("Character:", character)
else:
    print("Accessing out of range.")
```

Explanation:

This script tries to access a character out of the string index range and handles the resulting `IndexError` exception.

Output:

```
ERROR! IndexError: string index out of range Accessing out of
range.
```

String Exceptions - Question 3

Develop a function that handles exceptions when using an invalid key with `str.format()`.

String Exceptions - Question 3

Code:

```
def format_string_with_exception():
    try:
        # Try to format a string with an invalid key
        formatted_string = "Hello, {name}".format(age=30)
    except KeyError as e:
        print("KeyError:", e)
        formatted_string = None

    # Print the result
    if formatted_string is not None:
        print("Formatted String:", formatted_string)
    else:
        print("Formatting failed.")

# Example usage:
format_string_with_exception()
```

Explanation:

This function demonstrates handling exceptions when using an invalid key with `str.format()` and catching the resulting `KeyError`.

Output:

```
ERROR!  File "", line 4  formatted_string  =  "Hello,
{name}".format(age=30)  IndentationError: expected an indented
block after 'try' statement on line 2
```

String Exceptions - Question 4

Write a program that demonstrates exception handling for encoding errors when converting a string to bytes.

String Exceptions - Question 4

Code:

```
# Define a string with a character not representable in ASCII
invalid_string = "©"

try:
    # Try to encode the string to bytes using ASCII encoding
    encoded_bytes = invalid_string.encode("ascii")
except UnicodeEncodeError as e:
    print("UnicodeEncodeError:", e)
    encoded_bytes = None

# Print the result
if encoded_bytes is not None:
    print("Encoded Bytes:", encoded_bytes)
else:
    print("Encoding failed.")
```

Explanation:

This program demonstrates exception handling for encoding errors when trying to convert a string with a character not representable in ASCII to bytes.

Output:

Encoded Bytes: b'©'

String Exceptions - Question 5

Create a function that catches and handles the `TypeError` when trying to concatenate a string with a non-string type.

String Exceptions - Question 5

Code:

```
def concatenate_strings_with_exception():
    try:
        # Try to concatenate a string with a non-string type
        result = "Hello, " + 42
    except TypeError as e:
        print("TypeError:", e)
    result = None

    # Print the result
    if result is not None:
        print("Concatenated String:", result)
    else:
        print("Concatenation failed.")

# Example usage:
concatenate_strings_with_exception()
```

Explanation:

This function catches and handles the `TypeError` that occurs when trying to concatenate a string with a non-string type.

Output:

```
ERROR! TypeError: can only concatenate str (not "int") to str
Concatenation failed.
```

String Exceptions - Question 6

Write a script that handles exceptions for invalid operations, like dividing a string by a number.

String Exceptions - Question 6

Code:

```
def divide_strings_with_exception():
    try:
        # Try to divide a string by a number
        result = "Hello" / 2
    except TypeError as e:
        print("TypeError:", e)
        result = None

    # Print the result
    if result is not None:
        print("Result of Division:", result)
    else:
        print("Division failed.")

# Example usage:
divide_strings_with_exception()
```

Explanation:

This script handles exceptions for invalid operations, such as dividing a string by a number, and catches the resulting `TypeError`.

Output:

```
ERROR! TypeError: unsupported operand type(s) for /: 'str' and
'int' Division failed.
```

String Exceptions - Question 7

Develop a program that tries to modify a string (an immutable type) and handles the resulting `TypeError`.

String Exceptions - Question 7

Code:

```
def modify_immutable_string_with_exception():
    try:
        # Try to modify an immutable string
        text = "Hello, World!"
        text[0] = "B"
    except TypeError as e:
        print("TypeError:", e)

    # Print the result
    print("String after Modification:", text)

# Example usage:
modify_immutable_string_with_exception()
```

Explanation:

This program demonstrates trying to modify an immutable string and handling the resulting `TypeError` exception.

Output:

```
ERROR! TypeError: 'str' object does not support item assignment
String after Modification: Hello, World!
```

String Exceptions - Question 8

Create a function that uses try-except blocks to handle errors that occur when using the `replace()` method with incorrect arguments.

String Exceptions - Question 8

Code:

```
def replace_with_exception(text, old, new):
    try:
        # Try to replace a substring with incorrect arguments
        result = text.replace(old, new, maxreplace="invalid")
    except TypeError as e:
        print("TypeError:", e)
        result = None

    # Print the result
    if result is not None:
        print("Modified Text:", result)
    else:
        print("Replacement failed.")

# Example usage:
replace_with_exception("Hello, Hello, Hello!", "Hello", "Hi")
```

Explanation:

This function uses try-except blocks to handle errors that occur when using the `replace()` method with incorrect arguments, catching the resulting `TypeError`.

Output:

```
ERROR! TypeError: str.replace() takes no keyword arguments
Replacement failed.
```

String Exceptions - Question 9

Write a script that gracefully handles exceptions when using incorrect regular expression patterns.

String Exceptions - Question 9

Code:

```
import re

def search_with_exception(text, pattern):
    try:
        # Try to search for a pattern with incorrect syntax
        match = re.search(pattern, text)
    except re.error as e:
        print("Regex Error:", e)
        match = None

    # Print the result
    if match is not None:
        print("Match Found:", match.group())
    else:
        print("Search failed.")

# Example usage:
search_with_exception("Hello, World!", ".*[")
```

Explanation:

This script gracefully handles exceptions when trying to use incorrect regular expression patterns and catches the resulting `re.error`.

Output:

```
ERROR! Regex Error: unterminated character set at position 2
Search failed.
```

String Exceptions - Question 10

Develop a program that demonstrates handling exceptions when parsing a date string with an incorrect format.

String Exceptions - Question 10

Code:

```
from datetime import datetime

def parse_date_with_exception(date_string, format_string):
    try:
        # Try to parse a date string with an incorrect format
        parsed_date = datetime.strptime(date_string,
format_string)
    except ValueError as e:
        print("ValueError:", e)
        parsed_date = None

    # Print the result
    if parsed_date is not None:
        print("Parsed Date:", parsed_date)
    else:
        print("Parsing failed.")

# Example usage:
parse_date_with_exception("2023-12-17", "%Y/%m/%d")
```

Explanation:

This program demonstrates handling exceptions when trying to parse a date string with an incorrect format using the `datetime.strptime()` method and catching the resulting `ValueError`.

Output:

```
ERROR! ValueError: time data '2023-12-17' does not match format
'%Y/%m/%d' Parsing failed.
```

String Interning - Question 1

Write a program that demonstrates the concept of string interning by comparing the identity of two identical strings.

String Interning - Question 1

Code:

```
# Program to demonstrate string interning
str1 = "Hello, World!"
str2 = "Hello, World!"

# Check if str1 and str2 are identical objects
are_identical = str1 is str2

# Print the result
print("Are str1 and str2 identical objects?", are_identical)
```

Explanation:

This program demonstrates the concept of string interning by comparing the identity of two identical strings. In Python, identical string literals are typically interned, meaning they refer to the same memory object.

Output:

Are str1 and str2 identical objects? True

String Interning - Question 2

Create a function that uses `sys.intern()` to intern a string and shows the memory savings.

String Interning - Question 2

Code:

```
import sys

def intern_string_and_show_memory_savings(input_str):
    # Intern the input string
    interned_str = sys.intern(input_str)

    # Check memory addresses before and after interning
    original_address = id(input_str)
    interned_address = id(interned_str)

    # Calculate memory savings
    memory_savings = original_address - interned_address

    return interned_str, memory_savings

# Example usage:
input_str = "Hello, World!"
interned_str, memory_savings = intern_string_and_show_memory_savings(input_str)
print("Original String:", input_str)
print("Interned String:", interned_str)
print("Memory Savings (bytes):", memory_savings)
```

Explanation:

This function uses `sys.intern()` to intern a string and demonstrates the memory savings achieved by interning. It calculates the memory difference before and after interning. Demonstrating memory savings from string interning in a Python script is challenging because Python's memory management, including the allocation and representation of objects, is quite complex and often abstracted away from the developer. However, I can provide a theoretical example where string interning might lead to memory savings. Consider a scenario where you have a large number of identical strings that you're using frequently in your program. Without interning, each of these strings could be a separate object in memory, even if they have the same value. By interning these strings, Python

will store only one instance of the string in memory, which all references will point to.

Output:

Original String: Hello, World! Interned String: Hello, World!
Memory Savings (bytes): 0

String Interning - Question 3

Develop a script that compares the performance of string operations with and without string interning.

String Interning - Question 3

Code:

```
import sys
import timeit

# Define a function to perform string operations without interning
def perform_string_operations_without_interning():
    str1 = "Hello, " + "World!"
    str2 = "Python" + " is great!"

        # Define a function to perform string operations with interning
def perform_string_operations_with_interning():
    str1 = sys.intern("Hello, ") + sys.intern("World!")
    str2 = sys.intern("Python") + sys.intern(" is great!")

    # Measure the time taken for string operations without interning
time_without_interning = timeit.timeit(perform_string_operations_without_interning,
                                         number=1000000)

# Measure the time taken for string operations with interning
time_with_interning = timeit.timeit(perform_string_operations_with_interning,
                                         number=1000000)

# Print the results
print("Time without interning (seconds):",
      time_without_interning)
print("Time with interning (seconds):", time_with_interning)
```

Explanation:

This script compares the performance of string operations with and without string interning. It measures the time taken for each operation and demonstrates the advantage of interning in reducing memory and improving performance.

Output:

```
Time without interning (seconds): 0.062330199987627566 Time  
with interning (seconds): 0.2533699000487104
```

String Interning - Question 4

Write a program to explore the conditions under which Python automatically interns strings.

String Interning - Question 4

Code:

```
# Program to explore string interning in Python

def explore_string_interning():
    # String literals with the same content are typically interned
    str1 = "Hello, World!"
    str2 = "Hello, World!"

    # String variables with the same content may not be interned
    str3 = "Python"
    str4 = "Python"

    # Concatenated strings are not interned
    str5 = "Hello, " + "World!"
    str6 = "Python" + " is great!"

    # Check if strings are identical
    are_identical_1 = str1 is str2
    are_identical_2 = str3 is str4
    are_identical_3 = str5 is str6

    return are_identical_1, are_identical_2, are_identical_3

# Example usage:
are_identical_1, are_identical_2, are_identical_3 = explore_string_interning()
print("Are str1 and str2 identical objects?", are_identical_1)
print("Are str3 and str4 identical objects?", are_identical_2)
print("Are str5 and str6 identical objects?", are_identical_3)
```

Explanation:

This program explores the conditions under which Python automatically interns strings. It demonstrates that string literals with the same content are typically interned, but string variables with the same content and concatenated strings are not interned.

Output:

```
Are str1 and str2 identical objects? True Are str3 and str4  
identical objects? True Are str5 and str6 identical objects?  
False
```

String Interning - Question 5

Create a function that checks if two strings that are exactly the same are also identical objects in memory.

String Interning - Question 5

Code:

```
def are_identical_strings(str1, str2):
    # Check if str1 and str2 are identical objects
    are_identical = str1 is str2
    return are_identical

# Example usage:
str1 = "Hello, World!"
str2 = "Hello, World!"
are_identical = are_identical_strings(str1, str2)
print("Are str1 and str2 identical objects?", are_identical)
```

Explanation:

This function checks if two strings with the same content are also identical objects in memory by using the `is` operator for identity comparison.

Output:

Are str1 and str2 identical objects? True

String Interning - Question 6

Write a script that compares the memory addresses of short and long strings with identical content to see if they are interned.

String Interning - Question 6

Code:

```
# Script to compare memory addresses of strings with identical
content

# Short string
short_str = "Hello, World!"

# Long string with identical content
long_str = "Hello, World!" * 1000

# Check memory addresses
short_str_address = id(short_str)
long_str_address = id(long_str)

# Determine if they are interned
are_interned = short_str_address == long_str_address

# Print the result
print("Are short_str and long_str interned?", are_interned)
```

Explanation:

This script compares the memory addresses of short and long strings with identical content to determine if they are interned. In this case, they are not interned because the lengths of the strings differ.

Output:

Are short_str and long_str interned? False

String Interning - Question 7

Develop a program that demonstrates the use of string interning in a loop with a large number of string operations.

String Interning - Question 7

Code:

```
import sys

# Function to perform string operations with interning
def perform_string_operations_with_interning():
    result = ""
    for _ in range(100000):
        result += sys.intern("Hello, World!")
    return result

# Measure the time taken for string operations with interning
import timeit
time_with_interning = timeit.timeit(perform_string_operations_with_interning,
                                    number=100)

# Print the time taken
print("Time with interning (seconds):", time_with_interning)
```

Explanation:

This program demonstrates the use of string interning in a loop with a large number of string operations. It measures the time taken for interning-based string operations.

Output:

```
Time with interning (seconds): 4.18507020000834
```

String Interning - Question 8

Create a function that illustrates the difference between `==` and `is` operators with interned and non-interned strings.

String Interning - Question 8

Code:

```
def illustrate_difference_between_operators():
    # Interned strings
    str1 = "Hello, World!"
    str2 = "Hello, World!"

    # Non-interned strings
    str3 = "Python"
    str4 = " is great!"

    # Using '==' operator
    equal_interned = str1 == str2
    equal_non_interned = str3 == str4

    # Using 'is' operator
    identical_interned = str1 is str2
    identical_non_interned = str3 is str4

    return equal_interned, equal_non_interned,
           identical_interned, identical_non_interned

# Example usage:
equal_interned, equal_non_interned, identical_interned,
identical_non_interned = illustrate_difference_between_operators()
print("Using '==' with interned strings:", equal_interned)
print("Using '==' with non-interned strings:", equal_non_interned)
print("Using 'is' with interned strings:", identical_interned)
print("Using 'is' with non-interned strings:", identical_non_interned)
```

Explanation:

This function illustrates the difference between the `==` and `is` operators when used with interned and non-interned strings. The `==` operator checks content equality, while the `is` operator checks identity.

Output:

```
Using '==' with interned strings: True Using '==' with non-interned strings: False Using 'is' with interned strings: True  
Using 'is' with non-interned strings: False
```

String Interning - Question 9

Write a program that creates multiple strings with the same content in different ways and checks if they are interned.

String Interning - Question 9

Code:

```
# Program to create and check interned strings

def create_and_check_interned_strings():
    # Using literals
    str1 = "Hello, World!"
    str2 = "Hello, World!"

    # Using concatenation
    str3 = "Hello, " + "World!"
    str4 = "Hello" + ", World!"

    # Check if strings are interned
    interned_1_2 = str1 is str2
    interned_1_3 = str1 is str3
    interned_1_4 = str1 is str4

    return interned_1_2, interned_1_3, interned_1_4

# Example usage:
interned_1_2, interned_1_3, interned_1_4 = create_and_check_interned_strings()
print("str1 and str2 are interned:", interned_1_2)
print("str1 and str3 are interned:", interned_1_3)
print("str1 and str4 are interned:", interned_1_4)
```

Explanation:

This program creates multiple strings with the same content in different ways and checks if they are interned. String literals are typically interned, while concatenated strings may not be.

Output:

```
str1 and str2 are interned: True str1 and str3 are interned:
True str1 and str4 are interned: True
```

String Interning - Question 10

Develop a script that explains the impact of string interning on memory usage in large-scale applications.

String Interning - Question 10

Code:

```
# Script to explain the impact of string interning on memory usage

import sys

# Create a large number of identical strings
strings = ["Hello, World!"] * 1000000

# Measure memory usage without interning
memory_usage_without_interning = sys.getsizeof(strings[0]) * len(strings)

# Intern the strings
interned_strings = [sys.intern(s) for s in strings]

# Measure memory usage with interning
memory_usage_with_interning = sys.getsizeof(interned_strings[0]) * len(interned_strings)

# Print the results
print("Memory usage without interning:", memory_usage_without_interning, "bytes")
print("Memory usage with interning:", memory_usage_with_interning, "bytes")
```

Explanation:

This script explains the impact of string interning on memory usage in large-scale applications. It demonstrates that interning can reduce memory consumption by avoiding duplicate string storage.

Output:

```
Memory usage without interning: 62000000 bytes Memory usage with interning: 62000000 bytes
```

String Views - Question 1

Write a program that creates a `memoryview` from a byte string and prints its contents.

String Views - Question 1

Code:

```
# Program to create a memoryview from a byte string and print its contents

# Create a byte string
byte_string = b"Hello, World!"

# Create a memoryview from the byte string
memory_view = memoryview(byte_string)

# Print the contents of the memoryview
print(memory_view.tolist())
```

Explanation:

This program creates a `memoryview` from a byte string and prints its contents as a list of integer values representing the ASCII codes of each character.

Output:

```
[72, 101, 108, 108, 111, 44, 32, 87, 111, 114, 108, 100, 33]
```

String Views - Question 2

Create a function that uses a `memoryview` to modify a byte string without creating a copy.

String Views - Question 2

Code:

```
def modify_byte_string(byte_string, start, end, replacement):
    # Ensure that the byte_string is a bytearray (mutable)
    if not isinstance(byte_string, bytearray):
        raise TypeError("byte_string must be a bytearray")

    # Validate the length of the replacement
    if end - start != len(replacement):
        raise ValueError("Replacement length must be equal to
the slice length")

    # Create a memoryview from the byte string
    memory_view = memoryview(byte_string)

    # Modify the memoryview directly
    memory_view[start:end] = replacement

# Example usage:
byte_string = bytearray(b"Hello, World!")
modify_byte_string(byte_string, 0, 5, b"Hi---")
print(byte_string)
```

Explanation:

This function uses a `memoryview` to modify a byte string without creating a copy. It allows for in-place modification of a byte string's portion specified by the `start` and `end` indices.

Output:

```
bytearray(b'Hi---, World!')
```

String Views - Question 3

Develop a script that slices a byte string using a `memoryview` and compares the performance with regular slicing.

String Views - Question 3

Code:

```
import timeit

# Create a large byte string
byte_string = bytearray(b"Hello, World!" * 1000)

# Measure time taken for slicing with memoryview
def slice_with_memoryview():
    memory_view = memoryview(byte_string)
    result = memory_view[7:13] # Slicing using memoryview
    return result

time_memoryview      =      timeit.timeit(slice_with_memoryview,
number=10000)

# Measure time taken for regular slicing
def slice_without_memoryview():
    result = byte_string[7:13] # Regular slicing
    return result

time_regular      =      timeit.timeit(slice_without_memoryview,
number=10000)

print("Time with memoryview (seconds):", time_memoryview)
print("Time without memoryview (seconds):", time_regular)
```

Explanation:

This script compares the performance of slicing a byte string using a `memoryview` with regular slicing. It demonstrates that slicing with a `memoryview` is faster than regular slicing.

Output:

```
Time with memoryview (seconds): 0.0027788999723270535 Time
without memoryview (seconds): 0.001216100063174963
```

String Views - Question 4

Write a program that demonstrates the creation of multiple slices from the same `memoryview`.

String Views - Question 4

Code:

```
# Program to create multiple slices from the same memoryview

# Create a byte string
byte_string = bytearray(b"Hello, World!")

# Create a memoryview from the byte string
memory_view = memoryview(byte_string)

# Create multiple slices from the same memoryview
slice1 = memory_view[0:5]
slice2 = memory_view[6:11]

# Modify one of the slices
slice1[0] = ord('H')

# Print the original byte string
print(byte_string)

# Print the slices
print(slice1.tobytes()) # Convert slice1 to bytes
print(slice2.tobytes()) # Convert slice2 to bytes
```

Explanation:

This program demonstrates the creation of multiple slices from the same `memoryview`. It shows that modifications to one slice can affect the underlying byte string and other slices.

Output:

```
bytearray(b'Hello, World!') b'Hello' b' Worl'
```

String Views - Question 5

Create a function that converts a `memoryview` back into a byte string after modifying its contents.

String Views - Question 5

Code:

```
# Function to convert a memoryview back into a byte string
def memoryview_to_byte_string(memory_view):
    return memory_view.tobytes()

# Example usage:
byte_string = bytearray(b"Hello, World!")
memory_view = memoryview(byte_string)
memory_view[0] = ord('h')  # Modify the memoryview
result_byte_string = memoryview_to_byte_string(memory_view)
print(result_byte_string)
```

Explanation:

This function converts a `memoryview` back into a byte string after modifying its contents. It uses the `tobytes()` method to create a new byte string.

Output:

b'hello, World!'

String Views - Question 6

Write a script that uses a `memoryview` to access and print individual bytes of a byte string.

String Views - Question 6

Code:

```
# Script to access and print individual bytes of a byte string
# using memoryview

# Create a byte string
byte_string = bytearray(b"Hello, World!")

# Create a memoryview from the byte string
memory_view = memoryview(byte_string)

# Access and print individual bytes using memoryview
for byte in memory_view:
    print(byte)
```

Explanation:

This script uses a `memoryview` to access and print individual bytes of a byte string. It iterates through the memory view, allowing direct access to each byte's value.

Output:

```
72 101 108 108 111 44 32 87 111 114 108 100 33
```

String Views - Question 7

Develop a program that illustrates the difference in memory usage between a `memoryview` slice and a regular string slice.

String Views - Question 7

Code:

```
# Program to illustrate memory usage difference between
memoryview slice and string slice

import sys

# Create a large byte string
byte_string = bytearray(b"Hello, World!" * 1000)

# Create a memoryview slice
memory_view = memoryview(byte_string)[7:13] # Memoryview slice

# Create a regular string slice
string_slice = byte_string[7:13] # Regular string slice

# Measure memory usage of the memoryview slice
memory_usage_memoryview = sys.getsizeof(memory_view)

# Measure memory usage of the regular string slice
memory_usage_string = sys.getsizeof(string_slice)

print("Memory usage of memoryview slice:", memory_usage_memoryview, "bytes")
print("Memory usage of string slice:", memory_usage_string, "bytes")
```

Explanation:

This program illustrates the difference in memory usage between a `memoryview` slice and a regular string slice. It shows that the memoryview slice consumes slightly more memory due to its internal structure.

Output:

```
Memory usage of memoryview slice: 184 bytes Memory usage of
string slice: 63 bytes
```

String Views - Question 8

Create a function that takes a byte string, creates a `memoryview` of it, and then uses the `memoryview` to reverse the byte string.

String Views - Question 8

Code:

```
def reverse_byte_string(byte_string):
    # Create a memoryview from the byte string
    memory_view = memoryview(byte_string)

        # Reverse the memoryview and convert it to bytearray for
    assignment
    memory_view[:] = memory_view[::-1].tobytes()

# Example usage:
byte_string = bytearray(b"Hello, World!")
reverse_byte_string(byte_string)
print(byte_string)
```

Explanation:

This function takes a byte string, creates a `memoryview` of it, and then uses the `memoryview` to reverse the byte string in-place.

Output:

```
bytearray(b' !dlrow ,olleH')
```

String Views - Question 9

Write a script that demonstrates the immutability of a `memoryview` when created from a regular string.

String Views - Question 9

Code:

```
# Script to demonstrate the immutability of a memoryview from a
# regular string

# Create a regular string
regular_string = "Hello, World!"

# Create a memoryview from the regular string
memory_view = memoryview(regular_string.encode())

try:
    # Attempt to modify the memoryview
    memory_view[0] = ord('h')
except TypeError as e:
    print("Error:", e)
```

Explanation:

This script demonstrates the immutability of a `memoryview` when created from a regular string. It attempts to modify the memoryview, resulting in a `TypeError` because memoryviews from regular strings are read-only.

Output:

None

String Views - Question 10

Develop a program that combines multiple `memoryview` slices of a byte string into a new byte string.

String Views - Question 10

Code:

```
# Program to combine multiple memoryview slices into a new byte
string

# Create a byte string
byte_string = bytearray(b"Hello, World!")

# Create memoryview slices
memory_view1 = memoryview(byte_string)[:5]
memory_view2 = memoryview(byte_string)[7:]

# Combine memoryview slices into a new byte string
combined_byte_string      =      memory_view1.tobytes()      +
memory_view2.tobytes()

print(combined_byte_string)
```

Explanation:

This program combines multiple `memoryview` slices of a byte string into a new byte string by converting each memory view slice into bytes and concatenating them.

Output:

b'HelloWorld!'

String Translation - Question 1

Write a program that replaces all vowels in a string with asterisks using string translation.

String Translation - Question 1

Code:

```
# Program to replace vowels with asterisks using string
translation

# Input string
input_string = "Hello, World!"

# Define translation table to replace vowels with asterisks
translation_table = str.maketrans("AEIOUaeiou", "*****")

# Replace vowels with asterisks
result_string = input_string.translate(translation_table)

print(result_string)
```

Explanation:

This program replaces all vowels in a given string with asterisks using string translation. It defines a translation table that maps vowels to asterisks and uses the `str.translate()` method to perform the replacement.

Output:

None

String Translation - Question 2

Create a function that translates all digits in a string into their word equivalents (e.g., '1' to 'one').

String Translation - Question 2

Code:

```
def translate_digits_to_words(input_string):
    # Define translation table for digit-to-word mapping
    translation_table = {
        ord('0'): 'zero',
        ord('1'): 'one',
        ord('2'): 'two',
        ord('3'): 'three',
        ord('4'): 'four',
        ord('5'): 'five',
        ord('6'): 'six',
        ord('7'): 'seven',
        ord('8'): 'eight',
        ord('9'): 'nine'
    }

    # Translate digits to word equivalents
    result_string = input_string.translate(translation_table)

    return result_string.strip()

# Example usage:
input_string = "I have 3 apples and 2 bananas."
translated_string = translate_digits_to_words(input_string)
print(translated_string)
```

Explanation:

This function translates all digits in a given string into their word equivalents using string translation. It defines a translation table that maps digits to their word counterparts and applies it to the input string.

Output:

I have three apples and two bananas.

String Translation - Question 3

Develop a script that uses string translation to remove punctuation from a given string.

String Translation - Question 3

Code:

```
# Script to remove punctuation from a string using string
translation

import string

# Input string
input_string = "Hello, World! This is a test."

# Define translation table to remove punctuation
translation_table = str.maketrans("", "", string.punctuation)

# Remove punctuation from the string
result_string = input_string.translate(translation_table)

print(result_string)
```

Explanation:

This script uses string translation to remove punctuation from a given string. It defines a translation table that maps punctuation characters to empty strings and applies it to the input string.

Output:

Hello World This is a test

String Translation - Question 4

Write a function that creates a translation table to swap lowercase and uppercase letters, then uses it on a string.

String Translation - Question 4

Code:

```
# Function to create a translation table for swapping lowercase
and uppercase letters using string translation
def swap_lower_and_upper(input_string):
    # Define translation table for swapping lowercase and
    # uppercase letters
    translation_table =
        str.maketrans("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ",
                      "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz")

    # Apply the translation table to the input string
    result_string = input_string.translate(translation_table)

    return result_string

# Example usage:
input_string = "Hello, World!"
swapped_string = swap_lower_and_upper(input_string)
print(swapped_string)
```

Explanation:

This function creates a translation table for swapping lowercase and uppercase letters and applies it to a given string using string translation. It effectively swaps the case of the letters in the string.

Output:

hELLO, w0RLD!

String Translation - Question 5

Create a program that uses `str.maketrans()` and `str.translate()` to encode and decode a string with a simple Caesar cipher.

String Translation - Question 5

Code:

```
# Program to encode and decode a string with a simple Caesar
cipher using string translation

def caesar_cipher(text, shift):
    # Define translation table for Caesar cipher encoding and
    decoding
    def create_translation_table(shift):
        alphabet = "abcdefghijklmnopqrstuvwxyz"
        shifted_alphabet = alphabet[shift:] + alphabet[:shift]
        return str.maketrans(alphabet + alphabet.upper(),
shifted_alphabet + shifted_alphabet.upper())

    # Encode the text using the Caesar cipher
    encoded_text = text.translate(create_translation_table(shift))

    # Decode the encoded text using the Caesar cipher
    decoded_text = encoded_text.translate(create_translation_table(-shift))

    return encoded_text, decoded_text

# Example usage:
input_text = "Hello, World!"
shift_value = 3
encoded_text, decoded_text = caesar_cipher(input_text,
shift_value)
print(f"Original Text: {input_text}")
print(f"Encoded Text: {encoded_text}")
print(f"Decoded Text: {decoded_text}")
```

Explanation:

This program demonstrates encoding and decoding a string using a simple Caesar cipher. It defines a function that creates a translation table for the Caesar cipher, applies it to encode and decode the input text, and shows the original, encoded, and decoded texts.

Output:

Original Text: Hello, World! Encoded Text: Khoor, Zruog!
Decoded Text: Hello, World!

String Translation - Question 6

Write a script that translates spaces into underscores and vice versa in a string.

String Translation - Question 6

Code:

```
# Script to translate spaces into underscores and vice versa in
# a string

# Input string
input_string = "This is a test string with spaces."

# Define translation table to swap spaces and underscores
translation_table = str.maketrans(" _", "__")

# Translate spaces into underscores and vice versa
result_string = input_string.translate(translation_table)

print(result_string)
```

Explanation:

This script translates spaces into underscores and underscores into spaces in a given string using string translation. It defines a translation table that maps spaces to underscores and underscores to spaces and applies it to the input string.

Output:

This_is_a_test_string_with_spaces.

String Translation - Question 7

Develop a function that replaces specific characters in a string based on a translation table provided as an argument.

String Translation - Question 7

Code:

```
# Function to replace specific characters in a string using a
provided translation table
def replace_characters(input_string, translation_table):
    # Replace characters using the provided translation table
    result_string = input_string.translate(translation_table)
    return result_string

# Example usage:
input_string = "Hello, World!"
custom_translation_table = str.maketrans("Ho", "Xi")  # Replace
'H' with 'X' and 'o' with 'i'
replaced_string      =      replace_characters(input_string,
custom_translation_table)
print(replaced_string)
```

Explanation:

This function replaces specific characters in a given string using a provided translation table. It utilizes the `str.translate()` method with the custom translation table to perform the replacements.

Output:

Xelli, Wirld!

String Translation - Question 8

Create a program that removes all numeric characters from a string using string translation.

String Translation - Question 8

Code:

```
# Program to remove all numeric characters from a string using
# string translation

# Input string
input_string = "Hello, 123 World! 456"

# Define translation table to remove numeric characters
translation_table = str.maketrans("", "", "0123456789")

# Remove numeric characters from the string
result_string = input_string.translate(translation_table)

print(result_string)
```

Explanation:

This program removes all numeric characters from a given string using string translation. It defines a translation table that maps numeric characters to empty strings and applies it to the input string.

Output:

Hello, World!

String Translation - Question 9

Write a function that uses a translation table to replace multiple specific characters in a string with a single character.

String Translation - Question 9

Code:

```
# Function to replace multiple specific characters in a string
# with a single character using a translation table
def replace_multiple_characters(input_string,
                                translation_table):
    # Replace multiple characters using the provided
    # translation table
    result_string = input_string.translate(translation_table)
    return result_string

# Example usage:
input_string = "apple orange banana"
custom_translation_table = str.maketrans("aeiou", "XXXXX")  # Replace each vowel with 'X'
replaced_string = replace_multiple_characters(input_string,
                                              custom_translation_table)
print(replaced_string)
```

Explanation:

This function replaces multiple specific characters in a given string with a single character using a provided translation table. It demonstrates replacing vowels ('aeiou') and spaces with 'X' in the input string.

Output:

XpplX XrXngX bXnXnX

String Translation - Question 10

Develop a script that simulates a ROT13 cipher on a string using `str.translate()`.

String Translation - Question 10

Code:

```
# Script to simulate a ROT13 cipher on a string using
str.translate()

def rot13(text):
    # Define the ROT13 translation table
    rot13_table = str.maketrans(
        "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz",
        "NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm"
    )

    # Apply the ROT13 translation to the text
    result_text = text.translate(rot13_table)
    return result_text

# Example usage:
input_text = "Hello, World!"
encoded_text = rot13(input_text)
decoded_text = rot13(encoded_text)
print(f"Original Text: {input_text}")
print(f"Encoded Text: {encoded_text}")
print(f"Decoded Text: {decoded_text}")
```

Explanation:

This script simulates a ROT13 cipher on a given string using `str.translate()`. It defines a ROT13 translation table and applies it to encode and decode the input text.

Output:

```
Original Text: Hello, World! Encoded Text: Uryyb, Jbeyq!
Decoded Text: Hello, World!
```

String and Time Formatting - Question 1

Write a program that formats the current date and time as a string in the format "YYYY-MM-DD HH:MM:SS".

String and Time Formatting - Question 1

Code:

```
# Program to format the current date and time

import datetime

# Get the current date and time
current_datetime = datetime.datetime.now()

# Format the datetime as a string
formatted_datetime = current_datetime.strftime("%Y-%m-%d
%H:%M:%S")

print(formatted_datetime)
```

Explanation:

This program gets the current date and time and formats it as a string in the "YYYY-MM-DD HH:MM:SS" format using the `strftime` method.

Output:

2023-12-20 04:16:24

String and Time Formatting - Question 2

Create a function that takes a string representing a date and time, and converts it to a datetime object.

String and Time Formatting - Question 2

Code:

```
# Function to convert a string to a datetime object
import datetime

def string_to_datetime(date_string):
    try:
        # Parse the date string into a datetime object
        datetime_obj = datetime.datetime.strptime(date_string,
        "%Y-%m-%d %H:%M:%S")
        return datetime_obj
    except ValueError:
        return None

# Example usage:
date_string = "2023-01-17 14:30:45"
datetime_obj = string_to_datetime(date_string)
print(datetime_obj)
```

Explanation:

This function takes a string representing a date and time in the "YYYY-MM-DD HH:MM:SS" format and converts it into a datetime object using the `strptime` method.

Output:

2023-01-17 14:30:45

String and Time Formatting - Question 3

Develop a script that formats a given datetime object into a more readable string format, like "1st January 2023, 10:00 AM".

String and Time Formatting - Question 3

Code:

```
# Script to format a datetime object into a readable string
format

import datetime

# Create a datetime object
date_obj = datetime.datetime(2023, 1, 1, 10, 0)

# Format the datetime object into a readable string
formatted_date = date_obj.strftime("%dS %B %Y, %I:%M %p")

print(formatted_date)
```

Explanation:

This script takes a datetime object and formats it into a more readable string format like "1st January 2023, 10:00 AM" using the `strftime` method.

Output:

01S January 2023, 10:00 AM

String and Time Formatting - Question 4

Write a function that calculates the difference in days between two dates represented as strings.

String and Time Formatting - Question 4

Code:

```
# Function to calculate the difference in days between two
dates represented as strings
import datetime

def days_difference(date_string1, date_string2):
    try:
        # Parse the date strings into datetime objects
        date1 = datetime.datetime.strptime(date_string1, "%Y-
%m-%d")
        date2 = datetime.datetime.strptime(date_string2, "%Y-
%m-%d")

        # Calculate the difference in days
        delta = abs(date1 - date2)
        return delta.days
    except ValueError:
        return None

# Example usage:
date_string1 = "2023-01-01"
date_string2 = "2023-01-10"
difference = days_difference(date_string1, date_string2)
print(difference)
```

Explanation:

This function calculates the difference in days between two dates represented as strings in the "YYYY-MM-DD" format by parsing them into datetime objects and subtracting them.

Output:

String and Time Formatting - Question 5

Create a program that converts a string like "2 hours 30 minutes" into a timedelta object.

String and Time Formatting - Question 5

Code:

```
# Program to convert a string like "2 hours 30 minutes" into a
timedelta object
import datetime

def string_to_timedelta(time_string):
    try:
        # Split the input string
        parts = time_string.split()

        # Initialize variables for hours and minutes
        hours = 0
        minutes = 0

        # Iterate through the parts to extract hours and
minutes
        for i in range(len(parts)):
            if parts[i] == "hours" or parts[i] == "hour":
                hours = int(parts[i - 1])
            elif parts[i] == "minutes" or parts[i] == "minute":
                minutes = int(parts[i - 1])

        # Create a timedelta object with the extracted values
        delta = datetime.timedelta(hours=hours,
minutes=minutes)
        return delta
    except (ValueError, IndexError):
        return None

# Example usage:
time_string = "2 hours 30 minutes"
timedelta_obj = string_to_timedelta(time_string)
print(timedelta_obj)
```

Explanation:

This program converts a string like "2 hours 30 minutes" into a timedelta object by parsing and extracting the hours and minutes, then creating a timedelta with those values.

Output:

2:30:00

String and Time Formatting - Question 6

Write a script that formats the current time into a string showing only the hour and AM/PM indicator.

String and Time Formatting - Question 6

Code:

```
# Script to format the current time to show only the hour and
# AM/PM indicator

import datetime

# Get the current time
current_time = datetime.datetime.now().time()

# Format the time as a string
formatted_time = current_time.strftime("%I:%M %p")

print(formatted_time)
```

Explanation:

This script gets the current time and formats it as a string showing only the hour and AM/PM indicator using the `strftime` method.

Output:

04:21 AM

String and Time Formatting - Question 7

Develop a function that takes a list of strings representing dates and sorts them in chronological order.

String and Time Formatting - Question 7

Code:

```
# Function to sort a list of date strings in chronological
order
import datetime

def sort_dates(date_strings):
    try:
        # Parse the date strings into datetime objects and sort
        # them
        date_objects = [datetime.datetime.strptime(date, "%Y-
%m-%d") for date in date_strings]
        sorted_dates = sorted(date_objects)
        return [date.strftime("%Y-%m-%d") for date in
sorted_dates]
    except ValueError:
        return None

# Example usage:
date_strings = ["2023-01-10", "2023-01-01", "2023-01-05"]
sorted_dates = sort_dates(date_strings)
print(sorted_dates)
```

Explanation:

This function takes a list of date strings in the "YYYY-MM-DD" format, converts them into datetime objects, sorts them chronologically, and returns a list of sorted date strings.

Output:

```
['2023-01-01', '2023-01-05', '2023-01-10']
```

String and Time Formatting - Question 8

Create a program that adds a specific number of days to a date given as a string and returns the new date.

String and Time Formatting - Question 8

Code:

```
# Program to add a specific number of days to a date string
import datetime

def add_days_to_date(date_string, days_to_add):
    try:
        # Parse the date string into a datetime object
        date_obj = datetime.datetime.strptime(date_string, "%Y-%m-%d")

        # Calculate the new date by adding days
        new_date_obj = date_obj + datetime.timedelta(days=days_to_add)

        # Format the new date as a string
        new_date_string = new_date_obj.strftime("%Y-%m-%d")
        return new_date_string
    except ValueError:
        return None

# Example usage:
date_string = "2023-01-10"
days_to_add = 7
new_date = add_days_to_date(date_string, days_to_add)
print(new_date)
```

Explanation:

This program takes a date string in the "YYYY-MM-DD" format, adds a specific number of days to it, and returns the new date as a string.

Output:

2023-01-17

String and Time Formatting - Question 9

Write a function that extracts the month and year from a date string and formats them as "Month, Year".

String and Time Formatting - Question 9

Code:

```
# Function to extract and format month and year from a date
string
import datetime

def extract_month_year(date_string):
    try:
        # Parse the date string into a datetime object
        date_obj = datetime.datetime.strptime(date_string, "%Y-
%m-%d")

        # Extract the month and year
        month = date_obj.strftime("%B")
        year = date_obj.strftime("%Y")

        # Format as "Month, Year"
        formatted_string = f"{month}, {year}"
        return formatted_string
    except ValueError:
        return None

# Example usage:
date_string = "2023-01-10"
formatted_date = extract_month_year(date_string)
print(formatted_date)
```

Explanation:

This function extracts the month and year from a date string in the "YYYY-MM-DD" format, and formats them as "Month, Year".

Output:

January, 2023

String and Time Formatting - Question 10

Develop a script that creates a countdown timer string from a datetime object representing a future event.

String and Time Formatting - Question 10

Code:

```
# Script to create a countdown timer from a future datetime
import datetime

def countdown_timer(event_datetime):
    try:
        # Get the current datetime
        current_datetime = datetime.datetime.now()

        # Calculate the time difference
        time_difference = event_datetime - current_datetime

        # Extract days, hours, and minutes
        days = time_difference.days
        hours, remainder = divmod(time_difference.seconds,
3600)
        minutes = remainder // 60

        # Create the countdown timer string
        timer_string = f"{days} days, {hours} hours, {minutes} minutes"
    return timer_string
    except ValueError:
        return None

# Example usage:
event_datetime = datetime.datetime(2023, 12, 31, 23, 59, 59)
countdown = countdown_timer(event_datetime)
print(countdown)
```

Explanation:

This script calculates the countdown timer from the current datetime to a future event's datetime, showing the remaining days, hours, and minutes.

Output:

11 days, 19 hours, 32 minutes

Unicode Normalization - Question 1

Write a program that normalizes a Unicode string to NFC (Normalization Form C) and prints it.

Unicode Normalization - Question 1

Code:

```
import unicodedata

# Unicode string with separate base character and combining
character
unicode_string = "e\u0301clair" # 'e' followed by a combining
acute accent

# Normalize to NFC
normalized_string = unicodedata.normalize("NFC",
unicode_string)

print("Original String:", unicode_string)
print("NFC Normalized String:", normalized_string)
```

Explanation:

This program takes a Unicode string and normalizes it to NFC (Normalization Form C) to ensure consistent representation.

Output:

éclair

Unicode Normalization - Question 2

Create a function that compares two Unicode strings for equivalence after normalizing them to NFD (Normalization Form D).

Unicode Normalization - Question 2

Code:

```
# Function to compare two Unicode strings after normalizing to NFD
import unicodedata

def compare_normalized_strings(string1, string2):
    # Normalize both strings to NFD
    normalized_string1 = unicodedata.normalize("NFD", string1)
    normalized_string2 = unicodedata.normalize("NFD", string2)

    return normalized_string1 == normalized_string2

# Example usage:
result = compare_normalized_strings("café", "caf ")
print(result) # True
```

Explanation:

This function normalizes two Unicode strings to NFD (Normalization Form D) and compares them for equivalence, taking into account different representations.

Output:

False

Unicode Normalization - Question 3

Develop a script that normalizes a list of Unicode strings to NFKC (Normalization Form KC) and checks for duplicates.

Unicode Normalization - Question 3

Code:

```
# Script to normalize a list of Unicode strings to NFKC and
check for duplicates
import unicodedata

def normalize_and_check_duplicates(strings):
    normalized_strings = [unicodedata.normalize("NFKC", s) for
s in strings]
    duplicate_check      =      len(strings)      !=
len(set(normalized_strings))
    return normalized_strings, duplicate_check

# Example usage:
input_strings = ["café", "caf  ", "hello", "world"]
normalized,          has_duplicates           =
normalize_and_check_duplicates(input_strings)
print(normalized)
print("Has Duplicates:", has_duplicates)
```

Explanation:

This script normalizes a list of Unicode strings to NFKC (Normalization Form KC) and checks for duplicates by comparing the normalized versions.

Output:

```
['café', 'caf  ', 'hello', 'world'] Has Duplicates: False
```

Unicode Normalization - Question 4

Write a program that reads a Unicode string with combining characters and normalizes it to NFC for consistent display.

Unicode Normalization - Question 4

Code:

```
# Program to normalize a Unicode string with combining
characters to NFC
import unicodedata

# Input Unicode string with combining characters
unicode_string = "éclair"

# Normalize to NFC
normalized_string = unicodedata.normalize("NFC",
unicode_string)

print(normalized_string)
```

Explanation:

This program takes a Unicode string with combining characters and normalizes it to NFC (Normalization Form C) for consistent display.

Output:

éclair

Unicode Normalization - Question 5

Create a function that normalizes a Unicode string to NFKD (Normalization Form KD) and counts the number of distinct characters.

Unicode Normalization - Question 5

Code:

```
# Function to normalize a Unicode string to NFKD and count
distinct characters
import unicodedata

def count_distinct_normalized_characters(string):
    # Normalize the string to NFKD
    normalized_string = unicodedata.normalize("NFKD", string)

    # Count distinct characters
    distinct_characters = set(normalized_string)
    return len(distinct_characters)

# Example usage:
result = count_distinct_normalized_characters("café")
print(result) # 4 (c, a, f, é)
```

Explanation:

This function normalizes a Unicode string to NFKD (Normalization Form KD) and counts the number of distinct characters in the normalized form.

Output:

8

Unicode Normalization - Question 6

Write a script that normalizes user input in a web form to prevent inconsistencies in stored data.

Unicode Normalization - Question 6

Code:

```
# Script to normalize user input in a web form
import unicodedata

def normalize_user_input(user_input):
    # Normalize user input to NFC
    normalized_input = unicodedata.normalize("NFC", user_input)
    return normalized_input

# Example usage in a web form handler:
user_input = "éclair" # User-submitted input
normalized_input = normalize_user_input(user_input)
# Store normalized_input in the database
print("Stored:", normalized_input)
```

Explanation:

This script normalizes user input from a web form to NFC (Normalization Form C) to ensure consistent data storage and retrieval.

Output:

Stored: éclair

Unicode Normalization - Question 7

Develop a program that demonstrates the difference between the four Unicode normalization forms (NFC, NFD, NFKC, NFKD).

Unicode Normalization - Question 7

Code:

```
import unicodedata

def compare_normalization_forms(string):
    forms = {
        "NFC": unicodedata.normalize("NFC", string),
        "NFD": unicodedata.normalize("NFD", string),
        "NFKC": unicodedata.normalize("NFKC", string),
        "NFKD": unicodedata.normalize("NFKD", string),
    }
    return forms

# Example usage:
input_string = "A\u030A\ufb03"    # 'A' with a combining ring
above, followed by the ligature 'ffi'
result = compare_normalization_forms(input_string)
for form, normalized_string in result.items():
    print(f"{form}: {normalized_string}")
```

Explanation:

This program demonstrates the differences between the four Unicode normalization forms (NFC, NFD, NFKC, NFKD) for a given input string.

Output:

NFC: Åffi NFD: Åffi NFKC: Åffi NFKD: Åffi

Unicode Normalization - Question 8

Create a function that uses normalization to implement a case-insensitive comparison of Unicode strings.

Unicode Normalization - Question 8

Code:

```
# Function to perform case-insensitive comparison of Unicode
strings using normalization
import unicodedata

def case_insensitive_compare(string1, string2):
    # Normalize both strings to NFD and convert to lowercase
    normalized_string1 = unicodedata.normalize("NFD",
string1).lower()
    normalized_string2 = unicodedata.normalize("NFD",
string2).lower()

    return normalized_string1 == normalized_string2

# Example usage:
result = case_insensitive_compare("éclair", "Éclair")
print(result) # True
```

Explanation:

This function performs a case-insensitive comparison of Unicode strings by normalizing them to NFD and converting them to lowercase.

Output:

False

Unicode Normalization - Question 9

Write a script that removes all diacritics from a Unicode string by normalizing it to NFD and filtering out non-spacing marks.

Unicode Normalization - Question 9

Code:

```
# Script to remove diacritics from a Unicode string
import unicodedata

def remove_diacritics(input_string):
    # Normalize to NFD and filter out non-spacing marks
    result = ''.join([c for c in unicodedata.normalize('NFD',
input_string) if not unicodedata.combining(c)])
    return result

# Example usage:
input_string = "éclair"
result = remove_diacritics(input_string)
print(result) # "eclair"
```

Explanation:

This script removes diacritics from a Unicode string by normalizing it to NFD and filtering out non-spacing marks.

Output:

éclair

Unicode Normalization - Question 10

Develop a program that illustrates how Unicode normalization affects string length and indexing.

Unicode Normalization - Question 10

Code:

```
# Program to illustrate the effect of Unicode normalization on
string length and indexing
import unicodedata

def demonstrate_normalization_effect(input_string):
    # Normalize to NFC
    normalized_string = unicodedata.normalize('NFC',
input_string)

    # Calculate and compare string lengths
    original_length = len(input_string)
    normalized_length = len(normalized_string)

    # Demonstrate indexing
    original_char_at_1 = input_string[1]
    normalized_char_at_1 = normalized_string[1]

    return original_length, normalized_length,
original_char_at_1, normalized_char_at_1

# Example usage:
input_string = "éclair"
result = demonstrate_normalization_effect(input_string)
print("Original Length:", result[0])
print("Normalized Length:", result[1])
print("Original Char at 1:", result[2])
print("Normalized Char at 1:", result[3])
```

Explanation:

This program illustrates how Unicode normalization affects string length and indexing by comparing an input string to its normalized form.

Output:

```
Original Length: 12 Normalized Length: 12 Original Char at 1: &
Normalized Char at 1: &
```

Context-Aware Formatting - Question 1

Write a program that uses `string.Template` for creating a personalized greeting message.

Context-Aware Formatting - Question 1

Code:

```
# Program for personalized greeting using `string.Template`
import string

# Define a template with placeholders
template = string.Template("Hello, $name! Welcome to our
website.")

# User-specific data
user_data = {"name": "Alice"}

# Format the template with user data
greeting_message = template.substitute(user_data)

# Display the personalized greeting
print(greeting_message)
```

Explanation:

This program uses `string.Template` to create a personalized greeting message by substituting placeholders with user-specific data.

Output:

Hello, Alice! Welcome to our website.

Context-Aware Formatting - Question 2

Create a function that formats a price in a template string, adding the currency symbol based on a given locale.

Context-Aware Formatting - Question 2

Code:

```
# Function to format price with currency symbol based on locale
using `string.Template`
import string

def format_price(price, locale):
    # Define templates for different locales
    templates = {
        "en_US": string.Template("${symbol}${price}"),
        "fr_FR": string.Template("${price} ${symbol}"),
        "de_DE": string.Template("${price} ${symbol}"),
    }

    # Get the template for the specified locale
    template = templates.get(locale, templates["en_US"])

    # User-specific data
    user_data = {"price": price, "symbol": "$"}

    # Format the price with currency symbol
    formatted_price = template.substitute(user_data)

    return formatted_price

# Example usage:
price = 25.99
locale = "fr_FR"
formatted_price = format_price(price, locale)
print(formatted_price)
```

Explanation:

This function formats a price in a template string and adds the currency symbol based on the specified locale.

Output:

25.99 \$

Context-Aware Formatting - Question 3

Develop a script that uses a template with conditional formatting to create a simple invoice format.

Context-Aware Formatting - Question 3

Code:

```
import string

# Function to create the invoice
def create_invoice(template, data):
    # Conditional logic for the discount
    if data["include_discount"]:
        discount_section = f"Discount Applied: Yes\nDiscount Amount: {data['discount_amount']}"
    else:
        discount_section = "Discount Applied: No"

    # Prepare the data for the template
    formatted_data = {
        "customer_name": data["customer_name"],
        "invoice_date": data["invoice_date"],
        "discount_section": discount_section,
        "total_amount": data["total_amount"]
    }

    # Create the invoice using the template
    return template.substitute(formatted_data)

# Define a template without conditional logic
invoice_template = string.Template("""
Invoice
-----
Bill To: $customer_name
Invoice Date: $invoice_date
$discount_section
Total Amount: $total_amount
""")

# User-specific data
invoice_data = {
    "customer_name": "Alice",
    "invoice_date": "2023-01-15",
    "include_discount": True,
    "discount_amount": "$10.00",
```

```
        "total_amount": "$90.00",
    }

# Generate the invoice
invoice = create_invoice(invoice_template, invoice_data)

# Display the invoice
print(invoice)
```

Explanation:

This script creates a simple invoice format using `string.Template` with conditional formatting based on user-specific data.

Output:

```
Invoice ----- Bill To: Alice Invoice Date: 2023-01-15
Discount Applied: Yes Discount Amount: $10.00 Total Amount:
$90.00
```

Context-Aware Formatting - Question 4

Write a program that allows formatting a date in different styles using `string.Template`.

Context-Aware Formatting - Question 4

Code:

```
# Program to format a date in different styles using
`string.Template`
import string

# Define templates for different date styles
date_templates = {
    "short": string.Template("$day/$month/$year"),
    "long": string.Template("$month_name $day, $year"),
}

# User-specific data
date_data = {
    "day": "15",
    "month": "01",
    "year": "2023",
    "month_name": "January",
    "style": "long",    # Change this to "short" for a different
    style
}

# Get the template for the specified style
date_template      =      date_templates.get(date_data["style"],
date_templates["long"])

# Format the date using the template
formatted_date = date_template.substitute(date_data)

# Display the formatted date
print(formatted_date)
```

Explanation:

This program allows formatting a date in different styles using `string.Template` by selecting the appropriate template based on the chosen style.

Output:

January 15, 2023

Context-Aware Formatting - Question 5

Create a function that uses a template for generating an email with user-specific information.

Context-Aware Formatting - Question 5

Code:

```
# Function to generate an email with user-specific information
# using `string.Template`
import string

def generate_email(recipient, sender, subject):
    # Define an email template
    email_template = string.Template("""
        From: $sender
        To: $recipient
        Subject: $subject

        Dear $recipient,

        This is an automated email.

        Best regards,
        $sender
    """)

    # User-specific data
    email_data = {
        "recipient": recipient,
        "sender": sender,
        "subject": subject,
    }

    # Generate the email using the template
    email_content = email_template.substitute(email_data)

    return email_content

# Example usage:
recipient = "alice@example.com"
sender = "support@example.com"
subject = "Important Update"
email_content = generate_email(recipient, sender, subject)
print(email_content)
```

Explanation:

This function generates an email with user-specific information using `string.Template`.

Output:

From: support@example.com To: alice@example.com Subject: Important Update Dear alice@example.com, This is an automated email. Best regards, support@example.com

Context-Aware Formatting - Question 6

Write a script that formats a dictionary of data into a readable report using `string.Template`.

Context-Aware Formatting - Question 6

Code:

```
# Script to format a dictionary into a readable report using
`string.Template`
import string

# Define a template for the report
report_template = string.Template("""
    Report
    -----
    Name: $name
    Age: $age
    Country: $country
    Language: $language
""")

# User-specific data in dictionary format
user_data = {
    "name": "Alice",
    "age": 30,
    "country": "United States",
    "language": "English",
}

# Format the report using the template
report = report_template.substitute(user_data)

# Display the formatted report
print(report)
```

Explanation:

This script formats a dictionary of user data into a readable report using `string.Template`.

Output:

```
Report ----- Name: Alice Age: 30 Country: United States
Language: English
```

Context-Aware Formatting - Question 7

Develop a program that creates a custom string formatter to align text in a tabular format.

Context-Aware Formatting - Question 7

Code:

```
# Program to create a custom string formatter for tabular
alignment
import string

class TabularFormatter(string.Formatter):
    def format_field(self, value, format_spec):
        # Check if the format specification contains alignment
        if format_spec.startswith("<"):
            # Left align
            return format(value, format_spec[1:])
        elif format_spec.startswith(">"):
            # Right align
            return format(value, format_spec[1:])

        # Default alignment is left
        return format(value, format_spec)

# User-specific data
name = "Alice"
age = 30
country = "United States"

# Create a custom formatter
custom_formatter = TabularFormatter()

# Define the format string
format_string = """
    Name: {name:<10}
    Age: {age:>10}
    Country:{country:<10}
"""

# Format and align the data in a tabular format using the
# custom formatter
formatted_data = custom_formatter.vformat(format_string, (),
{'name': name, 'age': age, 'country': country})

# Display the formatted data
```

```
print(formatted_data)
```

Explanation:

This program demonstrates the creation of a custom string formatter for aligning text in a tabular format. The name "Alice" is left-aligned within a 10-character space. The age "30" is right-aligned within a 10-character space. The country "United States" is left-aligned within a 10-character space.

Output:

Name: Alice Age: 30 Country:United States

Context-Aware Formatting - Question 8

Create a function that uses a template to format a list of product names and prices into an HTML list.

Context-Aware Formatting - Question 8

Code:

```
import string

def format_product_list_with_string_template(products):
    # Preprocess the list items
    list_items = "".join([f"
        • {product['name']}: ${product['price']}
    " for product in products])

    # Define an HTML template for the list with a placeholder
    for list items
        list_template = string.Template("""
            $items
        """)

    # Substitute the placeholder with the actual list items
    formatted_list = list_template.substitute(items=list_items)

    return formatted_list

# User-specific data (list of products)
product_list = [
    {"name": "Product A", "price": 19.99},
    {"name": "Product B", "price": 29.95},
    {"name": "Product C", "price": 14.50},
]

# Format the product list into an HTML list using
string.Template
formatted_html_list = format_product_list_with_string_template(product_list)
```

```
# Display the formatted HTML list
print(formatted_html_list)
```

Explanation:

This function formats a list of product names and prices into an HTML list using `string.Template`.

Output:

- Product A: \$19.99
- Product B: \$29.95
- Product C: \$14.5

Context-Aware Formatting - Question 9

Write a script that uses context-aware formatting to handle pluralization in strings.

Context-Aware Formatting - Question 9

Code:

```
import string

def format_pluralization(num_items):
    # Determine the pluralization suffix
    suffix = "s" if num_items != 1 else ""

    # Define a pluralization template
    plural_template = string.Template("""
        You have $num_items item$suffix in your cart.
    """)

    # Format the pluralization string using the template
    return plural_template.substitute(num_items=num_items,
suffix=suffix)

# User-specific data
num_items = 3

# Format the pluralization string using the template
formatted_string = format_pluralization(num_items)

# Display the formatted string
print(formatted_string)
```

Explanation:

This script uses context-aware formatting to handle pluralization in strings, adding "s" to "item" when there are multiple items.

Output:

You have 3 items in your cart.

Context-Aware Formatting - Question 10

Develop a program that formats a log message differently based on the log level using `string.Template`.

Context-Aware Formatting - Question 10

Code:

```
import string

# Define log templates for different log levels
log_templates = {
    "info": string.Template("[INFO] $message"),
    "warning": string.Template("[WARNING] $message"),
    "error": string.Template("[ERROR] $message"),
}

# User-specific log data
log_data = {
    "message": "This is a log message.",
    "log_level": "info", # Change this to "warning" or "error"
    for different log levels
}

# Get the template for the specified log level
log_template = log_templates.get(log_data["log_level"],
log_templates["info"])

# Format the log message using the template
formatted_log = log_template.substitute(log_data)

# Display the formatted log message
print(formatted_log)
```

Explanation:

This program formats a log message differently based on the log level using `string.Template`, allowing customization of log messages.

Output:

```
[INFO] This is a log message.
```

String Parsing - Question 1

Write a program that extracts all the numbers from a string and sums them up.

String Parsing - Question 1

Code:

```
# Program to extract and sum numbers from a string
import re

# Input string
input_string = "The price of Item 1 is $25.50, and Item 2 costs
$30.25."

# Regular expression pattern to find numbers
number_pattern = r"\d+\.\d+|\d+"

# Find all numbers in the input string
numbers = re.findall(number_pattern, input_string)

# Convert numbers to floats and calculate the sum
total = sum(float(num) for num in numbers)

# Display the sum of numbers
print("Sum of numbers:", total)
```

Explanation:

This program extracts and sums all numbers from a string using regular expressions.

Output:

Sum of numbers: 58.75

String Parsing - Question 2

Create a function that parses a date string in various formats and converts it to a standard format.

String Parsing - Question 2

Code:

```
import datetime

def parse_and_convert_date(date_string):
    # Define input date formats
    input_formats = ["%Y-%m-%d", "%d/%m/%Y", "%m/%d/%Y", "%B %d, %Y"]

    for format in input_formats:
        try:
            # Try to parse the date using the current format
            parsed_date = datetime.datetime.strptime(date_string, format)

            # Convert to the standard format
            standard_format = parsed_date.strftime("%Y-%m-%d")

            return standard_format
        except ValueError:
            continue

    return None

# Example date strings in various formats
date1 = "2023-12-25"
date2 = "12/31/2023"
date3 = "July 4, 2023"

# Parse and convert date strings to a standard format
result1 = parse_and_convert_date(date1)
result2 = parse_and_convert_date(date2)
result3 = parse_and_convert_date(date3)

# Display the results
print("Date 1:", result1)
print("Date 2:", result2)
print("Date 3:", result3)
```

Explanation:

This function parses date strings in various formats and converts them to a standard "YYYY-MM-DD" format.

Output:

Date 1: 2023-12-25 Date 2: 2023-12-31 Date 3: 2023-07-04

String Parsing - Question 3

Develop a script that extracts email addresses from a block of text.

String Parsing - Question 3

Code:

```
# Script to extract email addresses from a block of text using
regular expressions
import re

# Input block of text
text = """
Please contact support@example.com for assistance.
For more information, email info@company.com.
"""

# Regular expression pattern to find email addresses
email_pattern = r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,7}\b"

# Find all email addresses in the input text
email_addresses = re.findall(email_pattern, text)

# Display the extracted email addresses
for email in email_addresses:
    print("Email:", email)
```

Explanation:

This script extracts email addresses from a block of text using a regular expression pattern.

Output:

Email: support@example.com Email: info@company.com

String Parsing - Question 4

Write a program that parses a CSV string into a list of dictionaries based on column headers.

String Parsing - Question 4

Code:

```
# Program to parse a CSV string into a list of dictionaries
import csv

# Input CSV string
csv_string = """
Name,Age,Location
Alice,30,New York
Bob,25,Los Angeles
Charlie,35,Chicago
"""

# Initialize a list to store dictionaries
data_list = []

# Parse the CSV string
csv_reader = csv.DictReader(csv_string.splitlines())

# Convert each row to a dictionary and add to the list
for row in csv_reader:
    data_list.append(dict(row))

# Display the list of dictionaries
print(data_list)
```

Explanation:

This program parses a CSV string into a list of dictionaries with column headers as keys.

Output:

```
[{'None': ['Name', 'Age', 'Location']}, {'None': ['Alice', '30', 'New York']}, {'None': ['Bob', '25', 'Los Angeles']}, {'None': ['Charlie', '35', 'Chicago']}]
```

String Parsing - Question 5

Create a function that takes a complex mathematical expression as a string and evaluates it.

String Parsing - Question 5

Code:

```
def evaluate_expression(expression):
    try:
        # Evaluate the expression using the 'eval' function
        result = eval(expression)
        return result
    except Exception as e:
        return f"Error: {str(e)}"

# Example mathematical expressions
expression1 = "3 + 5 * 2"
expression2 = "25 ** 0.5"
expression3 = "10 / (2 - 2)"

# Evaluate the expressions
result1 = evaluate_expression(expression1)
result2 = evaluate_expression(expression2)
result3 = evaluate_expression(expression3)

# Display the results
print("Result 1:", result1)
print("Result 2:", result2)
print("Result 3:", result3)
```

Explanation:

This function takes a complex mathematical expression as a string and evaluates it using the 'eval' function.

Output:

```
ERROR! Result 1: 13 Result 2: 5.0 Result 3: Error: division by zero
```

String Parsing - Question 6

Write a script that parses a JSON string and extracts specific elements.

String Parsing - Question 6

Code:

```
# Script to parse a JSON string and extract specific elements
import json

# Input JSON string
json_string = '{"name": "Alice", "age": 30, "city": "New York"}'

# Parse the JSON string
data = json.loads(json_string)

# Extract specific elements
name = data.get("name")
age = data.get("age")

# Display the extracted elements
print("Name:", name)
print("Age:", age)
```

Explanation:

This script parses a JSON string and extracts specific elements by using the `json.loads()` function.

Output:

Name: Alice Age: 30

String Parsing - Question 7

Develop a program that reads a log file and extracts error messages into a separate file.

String Parsing - Question 7

Code:

```
import re

# Function to extract error messages from a log file
def extract_error_messages(input_file, output_file):
    with open(input_file, "r") as log_file, open(output_file,
"w") as error_file:
        # Regular expression pattern to find error messages
        error_pattern = r"ERROR: (.+)"

        for line in log_file:
            match = re.search(error_pattern, line)
            if match:
                error_message = match.group(1)
                error_file.write(error_message + "\n")

# Example usage
extract_error_messages("logfile.txt", "errors.txt")
```

Explanation:

Explanation not provided

Output:

```
logfile.txt:ERROR: Hello World! This is an error message!
errors.txt: This is an error message!
```

String Parsing - Question 8

Create a function that converts a string representation of a list (e.g., "[1, 2, 3]") into an actual Python list.

String Parsing - Question 8

Code:

```
def string_to_list(input_string):
    try:
        # Remove brackets and split the string by commas
        elements = input_string.strip("[]").split(",")
        
        # Convert elements to integers
        integer_list = [int(element.strip()) for element in
elements]
        
        return integer_list
    except ValueError:
        return None

# Example string representation of a list
input_string = "[1, 2, 3, 4, 5]"

# Convert the string to a Python list
result = string_to_list(input_string)

# Display the result
print("Result:", result)
```

Explanation:

This function converts a string representation of a list into an actual Python list.

Output:

Result: [1, 2, 3, 4, 5]

String Parsing - Question 9

Write a script that takes a string containing nested data structures and parses it into Python objects.

String Parsing - Question 9

Code:

```
# Script to parse a string containing nested data structures
# into Python objects
import ast

# Input string
input_string = "[1, 2, {'key': 'value'}, [3, 4]]"

# Parse the input string into Python objects
parsed_objects = ast.literal_eval(input_string)

# Display the parsed objects
print(parsed_objects)
```

Explanation:

This script parses a string containing nested data structures into Python objects using the `ast.literal_eval()` function.

Output:

```
[1, 2, {'key': 'value'}, [3, 4]]
```

String Parsing - Question 10

Develop a program that uses regular expressions to parse a web page's HTML source code and extracts all hyperlinks.

String Parsing - Question 10

Code:

```
# Program to parse hyperlinks from a web page's HTML source code
import re

# Function to extract hyperlinks from HTML source code
def extract_hyperlinks(html_source):
    # Regular expression pattern to find hyperlinks
    hyperlink_pattern = r'href=["\'](https?://[^"\']+)+["\']'

    # Find all matches using the pattern
    matches = re.findall(hyperlink_pattern, html_source)

    return matches

# Example HTML source code
html_source = 'Example WebsiteAnother Website'

# Extract hyperlinks from the HTML source
hyperlinks = extract_hyperlinks(html_source)

# Display the extracted hyperlinks
for link in hyperlinks:
    print("Hyperlink:", link)
```

Explanation:

This program uses regular expressions to parse a web page's HTML source code and extract all hyperlinks.

Output:

Hyperlink: <https://example.com> Hyperlink: <https://another.com>

String Literals and Raw String Literals - Question 1

Write a program that demonstrates the difference in behavior between a regular string literal and a raw string literal when using backslashes.

String Literals and Raw String Literals - Question 1

Code:

```
# Program to demonstrate the difference between regular and raw
string literals
regular_string = "C:\\Users\\Alice\\Documents"
raw_string = r"C:\\Users\\Alice\\Documents"

print("Regular String:", regular_string)
print("Raw String:", raw_string)
```

Explanation:

This program illustrates the difference between a regular string literal and a raw string literal when dealing with backslashes. In a raw string literal, backslashes are treated as literal characters and do not escape the following character.

Output:

```
Regular String: C:\\Users\\Alice\\Documents      Raw String:
C:\\Users\\Alice\\Documents
```

String Literals and Raw String Literals - Question 2

Create a function that takes a file path as input and correctly handles it using a raw string literal.

String Literals and Raw String Literals - Question 2

Code:

```
# Function to handle a file path using a raw string literal
def handle_file_path(file_path):
    raw_file_path = r"C:\Users\Alice\Documents"

    if file_path == raw_file_path:
        return True
    else:
        return False

# Example usage
file_path = r"C:\Users\Alice\Documents"
result = handle_file_path(file_path)

# Display the result
if result:
    print("File paths match.")
else:
    print("File paths do not match.")
```

Explanation:

This function takes a file path as input and correctly handles it using a raw string literal. It compares the input file path with a predefined raw file path.

Output:

File paths match.

String Literals and Raw String Literals - Question 3

Develop a script that shows how a regular expression pattern changes when using a raw string literal versus a normal string.

String Literals and Raw String Literals - Question 3

Code:

```
# Script to compare regular expression patterns with and
without raw string literals
import re

# Regular expression pattern using a normal string
pattern_normal = "\d+"

# Regular expression pattern using a raw string literal
pattern_raw = r"\d+"

# Test strings
test_string = "123 456 789"

# Search using the normal pattern
matches_normal = re.findall(pattern_normal, test_string)

# Search using the raw pattern
matches_raw = re.findall(pattern_raw, test_string)

print("Matches with Normal Pattern:", matches_normal)
print("Matches with Raw Pattern:", matches_raw)
```

Explanation:

This script compares regular expression patterns defined using a normal string and a raw string literal. Both patterns match and extract digits from the test string.

Output:

```
Matches with Normal Pattern: ['123', '456', '789'] Matches with
Raw Pattern: ['123', '456', '789']
```

String Literals and Raw String Literals - Question 4

Write a program that constructs a Windows file path using both a regular string literal and a raw string, illustrating the differences.

String Literals and Raw String Literals - Question 4

Code:

```
# Program to construct a Windows file path using regular and
# raw string literals
regular_file_path = "C:\\\\Users\\\\Alice\\\\Documents"
raw_file_path = r"C:\\Users\\Alice\\Documents"

print("Regular File Path:", regular_file_path)
print("Raw File Path:", raw_file_path)
```

Explanation:

This program constructs a Windows file path using both a regular string literal and a raw string literal, illustrating that they produce the same result.

Output:

```
Regular File Path: C:\\Users\\Alice\\Documents Raw File Path:
C:\\Users\\Alice\\Documents
```

String Literals and Raw String Literals - Question 5

Create a function that uses a raw string literal to define a regular expression for matching multiline text.

String Literals and Raw String Literals - Question 5

Code:

```
import re

def match_multiline_text(text):
    # Define the regular expression pattern using a raw string
    # literal
    pattern = r"(?m)^.*$"

    # Search for matches
    matches = re.findall(pattern, text)

    return matches

# Example usage
multiline_text = """Line 1
Line 2
Line 3"""

# Find matches in the multiline text
matches = match_multiline_text(multiline_text)

# Display the matches
for match in matches:
    print(match)
```

Explanation:

This function uses a raw string literal to define a regular expression pattern for matching multiline text. The `^(?m).*\$` pattern matches each line in the input text.

Output:

```
Line 1 Line 2 Line 3
```

String Literals and Raw String Literals - Question 6

Write a script that demonstrates the escaping of special characters in a regular string literal and the lack of escaping in a raw string.

String Literals and Raw String Literals - Question 6

Code:

```
# Script to demonstrate escaping in regular and raw string literals
regular_string = "This is a regular string with a newline character: \n"
raw_string = r"This is a raw string with a newline character: \n"

print("Regular String:")
print(regular_string)

print("Raw String:")
print(raw_string)
```

Explanation:

This script demonstrates the difference between a regular string literal and a raw string literal when it comes to escaping special characters. In the regular string, the newline character `'\n'` is interpreted as a newline, while in the raw string, it is treated as a literal backslash and the letter 'n'.

Output:

```
Regular String: This is a regular string with a newline character: Raw String: This is a raw string with a newline character: \n
```

String Literals and Raw String Literals - Question 7

Develop a program that reads a regex pattern as input and explains why using a raw string literal is beneficial for this purpose.

String Literals and Raw String Literals - Question 7

Code:

```
import re

def explain_pattern(input_pattern):
    # Using a regular string literal
    try:
        re.compile(input_pattern)
        print("Using Regular String Literal: Pattern compiled successfully.")
    except re.error as e:
        print("Using Regular String Literal: Pattern compilation error:", e)

    # Using a raw string literal
    try:
        re.compile(r"" + input_pattern)
        print("Using Raw String Literal: Pattern compiled successfully.")
    except re.error as e:
        print("Using Raw String Literal: Pattern compilation error:", e)

    # Read regex pattern from user
    pattern_input = input("Enter a regex pattern: ")

    # Explain the use of both string types
    explain_pattern(pattern_input)
```

Explanation:

This program reads a regex pattern as input and demonstrates the use of both regular and raw string literals when compiling the pattern. Both string types work in this case, but using a raw string literal is beneficial because it avoids accidental escape character conflicts.

Output:

```
Enter a regex pattern: ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ Using Regular String Literal: Pattern compiled successfully. Using Raw String Literal: Pattern compiled successfully.
```

String Literals and Raw String Literals - Question 8

Create a function that compares the length of a string given as a regular literal and the same string as a raw literal.

String Literals and Raw String Literals - Question 8

Code:

```
# Function to compare the length of a regular and raw string
literal
def compare_string_lengths(input_string):
    regular_length = len(input_string)
    raw_length = len(r"" + input_string)

    return regular_length, raw_length

# Example usage
string_to_compare = "Hello\nWorld"

# Compare string lengths
regular_length, raw_length = compare_string_lengths(string_to_compare)

# Display the results
print("Regular String Length:", regular_length)
print("Raw String Length:", raw_length)
```

Explanation:

This function compares the length of a string given as a regular literal and the same string as a raw literal. The raw string literal includes an extra character ('\\') due to the escaped newline character.

Output:

Regular String Length: 11 Raw String Length: 11

String Literals and Raw String Literals - Question 9

Write a script that uses both regular and raw string literals to handle complex strings with mixed escape sequences.

String Literals and Raw String Literals - Question 9

Code:

```
# Script to handle complex strings with mixed escape sequences
# using regular and raw string literals
complex_string = "This is a complex\nstring with mixed\t
escape sequences: \\\n and \n\t"
complex_raw_string = r"This is a complex\nstring with mixed\t
escape sequences: \\\n and \n\t"

print("Complex String:")
print(complex_string)

print("Complex Raw String:")
print(complex_raw_string)
```

Explanation:

This script demonstrates the use of both regular and raw string literals to handle complex strings with mixed escape sequences. Raw string literals preserve the literal characters and are useful when you want to avoid interpretation of escape sequences.

Output:

```
Complex String: This is a complex string with mixed\t escape
sequences: \\n and \t Complex Raw String: This is a
complex\nstring with mixed\\t escape sequences: \\\n and \n\t
```

String Literals and Raw String Literals - Question 10

Develop a program that illustrates the use of raw string literals in JSON parsing scenarios.

String Literals and Raw String Literals - Question 10

Code:

```
# Program to illustrate the use of raw string literals in JSON parsing
import json

# Properly formatted JSON string
json_string = '{"name": "Alice", "city": "New York", "bio": "I'm a programmer.\nPython enthusiast."}'

# Parse the JSON string
data = json.loads(json_string)

# Display the parsed JSON data
print("Name:", data["name"])
print("City:", data["city"])
print("Bio:", data["bio"])
```

Explanation:

This program illustrates the use of raw string literals in JSON parsing scenarios. Raw string literals can be helpful when working with JSON strings that contain escape sequences, ensuring that they are interpreted as literal characters.

Output:

```
Name: Alice City: New York Bio: I'm a programmer. Python enthusiast.
```

Grapheme Clusters in Unicode Strings - Question 1

Write a program that counts the number of grapheme clusters in a given Unicode string.

Grapheme Clusters in Unicode Strings - Question 1

Code:

```
import unicodedata

def count_grapheme_clusters(input_string):
    grapheme_count = 0
    index = 0

    while index < len(input_string):
        try:
            char = input_string[index]
            cluster_length = len(unicodedata.normalize("NFC",
char))
            grapheme_count += 1
            index += cluster_length
        except ValueError:
            # Invalid character, skip it
            index += 1

    return grapheme_count

# Example usage
unicode_string = "café ልbc 123"
count = count_grapheme_clusters(unicode_string)
print("Number of Grapheme Clusters:", count)
```

Explanation:

This program counts the number of grapheme clusters in a Unicode string. It iterates through the string while handling surrogate pairs and combining characters, using the `unicodedata.normalize` function to determine cluster boundaries.

Output:

Number of Grapheme Clusters: 41

Grapheme Clusters in Unicode Strings - Question 2

Create a function that splits a Unicode string into individual grapheme clusters.

Grapheme Clusters in Unicode Strings - Question 2

Code:

```
import unicodedata

def split_into_grapheme_clusters(input_string):
    clusters = []
    index = 0

    while index < len(input_string):
        try:
            char = input_string[index]
            cluster_length = len(unicodedata.normalize("NFC",
char))
            cluster = input_string[index:index + cluster_length]
            clusters.append(cluster)
            index += cluster_length
        except ValueError:
            # Invalid character, skip it
            index += 1

    return clusters

# Example usage
unicode_string = "café Åbc 123"
clusters = split_into_grapheme_clusters(unicode_string)
print("Grapheme Clusters:", clusters)
```

Explanation:

This function splits a Unicode string into individual grapheme clusters, handling surrogate pairs and combining characters. It uses the `unicodedata.normalize` function to determine cluster boundaries.

Output:

Grapheme Clusters: ['c', 'a', 'f', '&', '#', '2', '3', '3',
';', ' ', '&', '#', '1', '1', '9', '9', '6', '4', ';', '&',
'#', '1', '1', '9', '9', '9', '1', ';', '&', '#', '1', '1',
'9', '9', '9', '2', ' ', '1', '2', '3']

Grapheme Clusters in Unicode Strings - Question 3

Develop a script that concatenates two Unicode strings and handles grapheme clusters correctly.

Grapheme Clusters in Unicode Strings - Question 3

Code:

```
import unicodedata

def concatenate_with_clusters(string1, string2):
    # Normalize both input strings to NFC form
    string1 = unicodedata.normalize("NFC", string1)
    string2 = unicodedata.normalize("NFC", string2)

    # Concatenate the normalized strings
    result = string1 + string2

    return result

# Example usage
unicode_string1 = "café"
unicode_string2 = "Ābc"
concatenated      = concatenate_with_clusters(unicode_string1,
                                              unicode_string2)
print("Concatenated String:", concatenated)
```

Explanation:

This script concatenates two Unicode strings while handling grapheme clusters correctly. It first normalizes both input strings to NFC form using the `unicodedata.normalize` function to ensure that combining characters are properly combined.

Output:

Concatenated String: caféĀbc

Grapheme Clusters in Unicode Strings - Question 4

Write a function that reverses a Unicode string while preserving the order of grapheme clusters.

Grapheme Clusters in Unicode Strings - Question 4

Code:

```
import unicodedata

def reverse_unicode_string(input_string):
    # Normalize the input string to NFC form
    normalized_string = unicodedata.normalize("NFC",
input_string)

    # Split the normalized string into grapheme clusters
    clusters = []
    index = 0
    while index < len(normalized_string):
        try:
            char = normalized_string[index]
            cluster_length = len(unicodedata.normalize("NFC",
char))
            cluster = normalized_string[index:index +
cluster_length]
            clusters.append(cluster)
            index += cluster_length
        except ValueError:
            # Invalid character, skip it
            index += 1

    # Reverse the order of clusters and join them
    reversed_string = "".join(reversed(clusters))

    return reversed_string

# Example usage
unicode_string = "café Abc 123"
reversed_str = reverse_unicode_string(unicode_string)
print("Reversed String:", reversed_str)
```

Explanation:

This function reverses a Unicode string while preserving the order of grapheme clusters. It first normalizes the input string to NFC form, splits it into clusters, reverses the order of clusters, and then joins them to create the reversed string.

Output:

Reversed String: 321 ;299911#&199911#&469911#& ;332#&fac

Grapheme Clusters in Unicode Strings - Question 5

Create a program that removes a specific grapheme cluster from a Unicode string.

Grapheme Clusters in Unicode Strings - Question 5

Code:

```
import unicodedata

def split_into_grapheme_clusters(input_string):
    clusters = []
    index = 0

    while index < len(input_string):
        try:
            char = input_string[index]
            cluster_length = len(unicodedata.normalize("NFC",
char))
            cluster = input_string[index:index + cluster_length]
            clusters.append(cluster)
            index += cluster_length
        except ValueError:
            # Invalid character, skip it
            index += 1

    return clusters

def remove_grapheme_cluster(input_string, cluster_to_remove):
    # Normalize the input string to NFC form
    normalized_string = unicodedata.normalize("NFC",
input_string)

    # Split the normalized string into grapheme clusters
    clusters = split_into_grapheme_clusters(normalized_string)

    # Remove the specified cluster from the list
    cleaned_clusters = [cluster for cluster in clusters if
cluster != cluster_to_remove]

    # Join the cleaned clusters to create the resulting string
    cleaned_string = "".join(cleaned_clusters)

    return cleaned_string
```

```
# Example usage
unicode_string = "café ልbc 123"
cluster_to_remove = "b"
result          =         remove_grapheme_cluster(unicode_string,
cluster_to_remove)
print("Resulting String:", result)
```

Explanation:

This program removes a specific grapheme cluster from a Unicode string while preserving the order of other clusters. It first normalizes the input string to NFC form, splits it into clusters, removes the specified cluster, and then joins the cleaned clusters to create the resulting string.

Output:

Resulting String: café አbc 123

Grapheme Clusters in Unicode Strings - Question 6

Write a script that identifies and prints all unique grapheme clusters in a given text.

Grapheme Clusters in Unicode Strings - Question 6

Code:

```
import unicodedata

def split_into_grapheme_clusters(input_string):
    clusters = []
    index = 0

    while index < len(input_string):
        try:
            char = input_string[index]
            cluster_length = len(unicodedata.normalize("NFC",
char))
            cluster = input_string[index:index + cluster_length]
            clusters.append(cluster)
            index += cluster_length
        except ValueError:
            # Invalid character, skip it
            index += 1

    return clusters

def unique_grapheme_clusters(input_string):
    # Normalize the input string to NFC form
    normalized_string = unicodedata.normalize("NFC",
input_string)

    # Split the normalized string into grapheme clusters
    clusters = split_into_grapheme_clusters(normalized_string)

    # Use a set to store unique clusters
    unique_clusters = set(clusters)

    return unique_clusters

# Example usage
unicode_string = "café Ábc 123"
unique_clusters = unique_grapheme_clusters(unicode_string)
```

```
print("Unique Grapheme Clusters:", unique_clusters)
```

Explanation:

This script identifies and prints all unique grapheme clusters in a given Unicode string. It normalizes the input string to NFC form, splits it into clusters, and then uses a set to collect unique clusters.

Output:

```
Unique Grapheme Clusters: {'c', '4', ' ', 'f', 'a', '2', '1',
'9', '6', '&', ';', '3', '#'}
```

Grapheme Clusters in Unicode Strings - Question 7

Develop a program that compares two Unicode strings for equality, taking grapheme clusters into account.

Grapheme Clusters in Unicode Strings - Question 7

Code:

```
import unicodedata

def compare_with_clusters(string1, string2):
    # Normalize both input strings to NFC form
    normalized_string1 = unicodedata.normalize("NFC", string1)
    normalized_string2 = unicodedata.normalize("NFC", string2)

    return normalized_string1 == normalized_string2

# Example usage
unicode_string1 = "café Abc 123"
unicode_string2 = "café Abc 1 2 3"
result          = compare_with_clusters(unicode_string1,
                                         unicode_string2)
print("Strings are Equal (Considering Clusters):", result)
```

Explanation:

This program compares two Unicode strings for equality while taking grapheme clusters into account. It normalizes both input strings to NFC form and then performs the comparison.

Output:

Strings are Equal (Considering Clusters): False

Grapheme Clusters in Unicode Strings - Question 8

Create a function that truncates a Unicode string to a specified length in grapheme clusters, not code points.

Grapheme Clusters in Unicode Strings - Question 8

Code:

```
# Function to truncate a Unicode string to a specified length
# in grapheme clusters
import unicodedata

def split_into_grapheme_clusters(s):
    """
    Split a Unicode string into grapheme clusters.
    """
    grapheme_clusters = []
    i = 0
    while i < len(s):
        cluster = s[i]
        j = i + 1
        while j < len(s) and unicodedata.combining(s[j]):
            cluster += s[j]
            j += 1
        grapheme_clusters.append(cluster)
        i = j
    return grapheme_clusters

def truncate_with_clusters(input_string, max_clusters):
    # Normalize the input string to NFC form
    normalized_string = unicodedata.normalize("NFC",
input_string)

    # Split the normalized string into grapheme clusters
    clusters = split_into_grapheme_clusters(normalized_string)

    # Truncate the clusters based on the specified length
    truncated_clusters = clusters[:max_clusters]

    # Join the truncated clusters to create the resulting
    # string
    truncated_string = "".join(truncated_clusters)

    return truncated_string
```

```
# Example usage
unicode_string = "café Abc 123"
max_clusters = 5
truncated_str      =      truncate_with_clusters(unicode_string,
max_clusters)
print("Truncated String (", max_clusters, " clusters):",
truncated_str)
```

Explanation:

This function truncates a Unicode string to a specified length in grapheme clusters, ensuring that the truncation does not split graphemes. It normalizes the input string to NFC form, splits it into clusters, truncates the clusters, and then joins them to create the resulting string.

Output:

Truncated String (5 clusters): caf&#

Grapheme Clusters in Unicode Strings - Question 9

Write a script that replaces a grapheme cluster in a Unicode string with another cluster.

Grapheme Clusters in Unicode Strings - Question 9

Code:

```
import unicodedata

def split_into_grapheme_clusters(s):
    """
    Split a Unicode string into grapheme clusters.
    """
    grapheme_clusters = []
    i = 0
    while i < len(s):
        cluster = s[i]
        j = i + 1
        while j < len(s) and unicodedata.combining(s[j]):
            cluster += s[j]
            j += 1
        grapheme_clusters.append(cluster)
        i = j
    return grapheme_clusters

def replace_grapheme_cluster(input_string, old_cluster, new_cluster):
    # Normalize the input string to NFC form
    normalized_string = unicodedata.normalize("NFC", input_string)

    # Split the normalized string into grapheme clusters
    clusters = split_into_grapheme_clusters(normalized_string)

    # Create a list with replaced clusters
    replaced_clusters = []
    i = 0
    while i < len(clusters):
        if clusters[i] == old_cluster:
            replaced_clusters.append(new_cluster)
            i += 1
        else:
            replaced_clusters.append(clusters[i])
            i += 1
```

```
# Join the replaced clusters to create the resulting string
replaced_string = "".join(replaced_clusters)

return replaced_string

# Example usage
unicode_string = "café ልbc 123"
old_cluster = "b"
new_cluster = "χ"
result = replace_grapheme_cluster(unicode_string, old_cluster,
new_cluster)
print("Resulting String:", result)
```

Explanation:

This script replaces a specified grapheme cluster in a Unicode string with another cluster while preserving the order of other clusters. It normalizes the input string to NFC form, splits it into clusters, replaces the specified cluster, and then joins the replaced clusters to create the resulting string.

Output:

Resulting String: café ልbc 123

Grapheme Clusters in Unicode Strings - Question 10

Develop a program that extracts substrings from a Unicode string while preserving entire grapheme clusters.

Grapheme Clusters in Unicode Strings - Question 10

Code:

```
import unicodedata

def split_into_grapheme_clusters(s):
    """
    Split a Unicode string into grapheme clusters.
    """
    grapheme_clusters = []
    i = 0
    while i < len(s):
        cluster = s[i]
        j = i + 1
        while j < len(s) and unicodedata.combining(s[j]):
            cluster += s[j]
            j += 1
        grapheme_clusters.append(cluster)
        i = j
    return grapheme_clusters

def extract_substrings_with_clusters(input_string, start, end):
    # Normalize the input string to NFC form
    normalized_string = unicodedata.normalize("NFC",
input_string)

    # Split the normalized string into grapheme clusters
    clusters = split_into_grapheme_clusters(normalized_string)

    # Extract substrings based on grapheme clusters
    extracted_clusters = clusters[start:end]

    # Join the extracted clusters to create the resulting
string
    extracted_string = "".join(extracted_clusters)

    return extracted_string

# Example usage
unicode_string = "café Abc 123"
```

```
start_index = 5
end_index = 9
substring = extract_substrings_with_clusters(unicode_string,
start_index, end_index)
print("Extracted Substring:", substring)
```

Explanation:

This program extracts substrings from a Unicode string while preserving entire grapheme clusters. It normalizes the input string to NFC form, splits it into clusters, extracts the specified clusters, and then joins them to create the resulting string.

Output:

Extracted Substring: 233;

String Algorithms - Question 1

Write a function that implements the Knuth-Morris-Pratt (KMP) algorithm for substring search.

String Algorithms - Question 1

Code:

```
# Function to implement the Knuth-Morris-Pratt (KMP) algorithm
for substring search
def kmp_search(text, pattern):
    def compute_prefix_array(pattern):
        prefix_array = [0] * len(pattern)
        j = 0
        for i in range(1, len(pattern)):
            while j > 0 and pattern[i] != pattern[j]:
                j = prefix_array[j - 1]
            if pattern[i] == pattern[j]:
                j += 1
            prefix_array[i] = j
        return prefix_array

    prefix_array = compute_prefix_array(pattern)
    matches = []
    j = 0
    for i in range(len(text)):
        while j > 0 and text[i] != pattern[j]:
            j = prefix_array[j - 1]
        if text[i] == pattern[j]:
            j += 1
        if j == len(pattern):
            matches.append(i - j + 1)
            j = prefix_array[j - 1]
    return matches

# Example usage
text = "ABABDABACDABABCABAB"
pattern = "ABABCABAB"
matches = kmp_search(text, pattern)
print("Pattern found at positions:", matches)
```

Explanation:

The provided function implements the Knuth-Morris-Pratt (KMP) algorithm for substring search. It efficiently finds all

occurrences of a pattern within a given text.

Output:

Pattern found at positions: [10]

String Algorithms - Question 2

Create a script that finds all anagrams of a given word in a list of words.

String Algorithms - Question 2

Code:

```
from collections import Counter

def find_anagrams(word, word_list):
    anagrams = []
    word_counter = Counter(word)
    for candidate in word_list:
        if Counter(candidate) == word_counter:
            anagrams.append(candidate)
    return anagrams

# Example usage
word = "listen"
word_list = ["silent", "enlist", "tinsel", "hello", "world"]
anagrams = find_anagrams(word, word_list)
print("Anagrams of '{}':".format(word), anagrams)
```

Explanation:

This script finds all anagrams of a given word in a list of words. It uses the Counter class from the collections module to compare character frequencies.

Output:

```
Anagrams of 'listen': ['silent', 'enlist', 'tinsel']
```

String Algorithms - Question 3

Develop a program that implements the Levenshtein distance algorithm to measure the similarity between two strings.

String Algorithms - Question 3

Code:

```
def levenshtein_distance(str1, str2):
    if len(str1) < len(str2):
        return levenshtein_distance(str2, str1)
    if len(str2) == 0:
        return len(str1)

    previous_row = range(len(str2) + 1)
    for i, c1 in enumerate(str1):
        current_row = [i + 1]
        for j, c2 in enumerate(str2):
            insertions = previous_row[j + 1] + 1
            deletions = current_row[j] + 1
            substitutions = previous_row[j] + (c1 != c2)
            current_row.append(min(insertions, deletions,
substitutions))
        previous_row = current_row

    return previous_row[-1]

# Example usage
str1 = "kitten"
str2 = "sitting"
distance = levenshtein_distance(str1, str2)
print("Levenshtein distance between '{}' and '{}': {}".format(str1, str2), distance)
```

Explanation:

This program implements the Levenshtein distance algorithm, which calculates the minimum number of single-character edits (insertions, deletions, or substitutions) needed to transform one string into another.

Output:

Levenshtein distance between 'kitten' and 'sitting': 3

String Algorithms - Question 4

Write a function that performs a longest common subsequence (LCS) computation between two strings.

String Algorithms - Question 4

Code:

```
def longest_common_subsequence(str1, str2):
    m, n = len(str1), len(str2)
    lcs_matrix = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if str1[i - 1] == str2[j - 1]:
                lcs_matrix[i][j] = lcs_matrix[i - 1][j - 1] + 1
            else:
                lcs_matrix[i][j] = max(lcs_matrix[i - 1][j], lcs_matrix[i][j - 1])

    i, j = m, n
    lcs = []
    while i > 0 and j > 0:
        if str1[i - 1] == str2[j - 1]:
            lcs.append(str1[i - 1])
            i -= 1
            j -= 1
        elif lcs_matrix[i - 1][j] > lcs_matrix[i][j - 1]:
            i -= 1
        else:
            j -= 1

    return "".join(reversed(lcs))

# Example usage
str1 = "AGGTAB"
str2 = "GXTXAYB"
lcs = longest_common_subsequence(str1, str2)
print("Longest Common Subsequence between {} and {}".format(str1, str2), lcs)
```

Explanation:

The provided function calculates the Longest Common Subsequence (LCS) between two input strings, which is the longest sequence

of characters that appear in the same order in both strings.

Output:

Longest Common Subsequence between 'AGGTAB' and 'GXTXAYB': GTAB

String Algorithms - Question 5

Create a program that uses a trie data structure for efficient string searching.

String Algorithms - Question 5

Code:

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word

# Example usage
trie = Trie()
words = ["apple", "banana", "app", "apricot", "bat"]
for word in words:
    trie.insert(word)

print(trie.search("apple"))      # True
print(trie.search("app"))       # True
print(trie.search("banana"))     # True
print(trie.search("apricot"))    # True
print(trie.search("orange"))    # False
```

Explanation:

This program implements a trie data structure for efficient string searching. It allows inserting words into the trie and searching for words efficiently, making it suitable for tasks like autocomplete and dictionary lookups.

Output:

True True True True False

String Algorithms - Question 6

Write a script that implements the Rabin-Karp algorithm for string matching.

String Algorithms - Question 6

Code:

```
def rabin_karp_search(text, pattern):
    result = []
    text_len = len(text)
    pattern_len = len(pattern)

    if text_len < pattern_len:
        return result

    # Calculate the hash of the pattern
    pattern_hash = sum(ord(pattern[i]) for i in range(pattern_len))

    # Calculate the initial hash of the first window in the text
    text_hash = sum(ord(text[i]) for i in range(pattern_len))

    for i in range(text_len - pattern_len + 1):
        if text_hash == pattern_hash and text[i:i+pattern_len] == pattern:
            result.append(i)

        if i < text_len - pattern_len:
            # Update the rolling hash by removing the left character and adding the right character
            text_hash = text_hash - ord(text[i]) + ord(text[i+pattern_len])

    return result

# Example usage
text = "abracadabra"
pattern = "abra"
matches = rabin_karp_search(text, pattern)
print("Pattern found at positions:", matches)
```

Explanation:

This script implements the Rabin-Karp algorithm for string matching. It calculates the hash of the pattern and the initial hash of the first window in the text. It then iterates through the text, updating the rolling hash, and checks if the hash matches that of the pattern.

Output:

Pattern found at positions: [0, 7]

String Algorithms - Question 7

Develop a function that finds the longest palindromic substring within a given string.

String Algorithms - Question 7

Code:

```
def longest_palindromic_substring(s):
    if not s:
        return ""

    def expand_around_center(left, right):
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return s[left + 1:right]

    longest = ""
    for i in range(len(s)):
        # Check for odd-length palindromes
        palindrome1 = expand_around_center(i, i)
        if len(palindrome1) > len(longest):
            longest = palindrome1

        # Check for even-length palindromes
        palindrome2 = expand_around_center(i, i + 1)
        if len(palindrome2) > len(longest):
            longest = palindrome2

    return longest

# Example usage
input_string = "babad"
result = longest_palindromic_substring(input_string)
print("Longest Palindromic Substring:", result)
```

Explanation:

This function finds the longest palindromic substring within a given string using the expand-around-center technique. It iterates through the string, treating each character as a potential center of a palindrome, and expands around it to find the longest palindromes.

Output:

Longest Palindromic Substring: bab

String Algorithms - Question 8

Create a program that uses a suffix array to find all occurrences of a substring in a given string.

String Algorithms - Question 8

Code:

```
def build_suffix_array(text):
    suffixes = [(text[i:], i) for i in range(len(text))]
    suffixes.sort()
    return [suffix[1] for suffix in suffixes]

def search_substring_using_suffix_array(text, substring):
    suffix_array = build_suffix_array(text)
    substring_len = len(substring)
    result = []

    for i in suffix_array:
        suffix = text[i:]
        if substring == suffix[:substring_len]:
            result.append(i)

    return result

# Example usage
text = "banana"
substring = "an"
matches = search_substring_using_suffix_array(text, substring)
print("Substring found at positions:", matches)
```

Explanation:

This program uses a suffix array to find all occurrences of a substring in a given string. It first builds the suffix array by sorting the suffixes of the text. Then, it iterates through the suffixes and checks if the substring matches the prefix of the current suffix.

Output:

Substring found at positions: [3, 1]

String Algorithms - Question 9

Write a function that implements a basic text justification algorithm for a block of text.

String Algorithms - Question 9

Code:

```
def justify_text(text, line_length):
    words = text.split()
    lines = []
    current_line = []
    current_length = 0

    for word in words:
        if current_length + len(word) + len(current_line) <=
line_length:
            current_line.append(word)
            current_length += len(word)
        else:
            lines.append(current_line)
            current_line = [word]
            current_length = len(word)

    if current_line:
        lines.append(current_line)

justified_lines = []

for line in lines:
    num_words = len(line)
    total_word_length = sum(len(word) for word in line)
    spaces_needed = line_length - total_word_length
    if num_words == 1:
        justified_lines.append(line[0] + " " * spaces_needed)
    else:
        spaces_per_gap = spaces_needed // (num_words - 1)
        extra_spaces = spaces_needed % (num_words - 1)
        justified_line = line[0]
        for i in range(1, num_words):
            if i <= extra_spaces:
                justified_line += " " * (spaces_per_gap +
1) + line[i]
            else:
                justified_line += " " * spaces_per_gap +
```

```
        line[i]
                justified_lines.append(justified_line)

        return "\n".join(justified_lines)

# Example usage
input_text = "This is a sample text to demonstrate text
justification."
line_length = 20
justified_text = justify_text(input_text, line_length)
print(justified_text)
```

Explanation:

This function implements a basic text justification algorithm for a block of text. It splits the text into words, organizes them into lines, and then justifies each line to a specified line length by adding spaces between words.

Output:

This is a sample text to demonstrate text justification.

String Algorithms - Question 10

Develop a script that uses the Boyer-Moore algorithm for fast string searching in a large text.

String Algorithms - Question 10

Code:

```
def boyer_moore_search(text, pattern):
    def bad_character_table(pattern):
        table = {}
        for i in range(len(pattern) - 1):
            table[pattern[i]] = len(pattern) - 1 - i
        return table

    def good_suffix_table(pattern):
        table = [-1] * len(pattern)
        j = len(pattern) - 1
        for i in range(len(pattern) - 1, -1, -1):
            if all(pattern[i] == pattern[i - k] for k in range(j, 0, -1)):
                j = len(pattern) - 1 - i
            table[i] = j
        return table

    m = len(pattern)
    n = len(text)
    if m == 0 or m > n:
        return []

    bc_table = bad_character_table(pattern)
    gs_table = good_suffix_table(pattern)

    i = 0
    while i <= n - m:
        j = m - 1
        while j >= 0 and pattern[j] == text[i + j]:
            j -= 1
        if j < 0:
            yield i
            i += gs_table[0]
        else:
            i += max(bc_table.get(text[i + j], 0), gs_table[j])

# Example usage
text = "ABAAABCDABCDABCABCABCDABCDABCDABCD"
```

```
pattern = "ABCDABCDABCDABCDABCD"  
matches = list(boyer_moore_search(text, pattern))  
print("Pattern found at positions:", matches)
```

Explanation:

This script uses the Boyer-Moore algorithm for fast string searching in a large text. It builds bad character and good suffix tables, then iterates through the text to find all occurrences of the pattern efficiently.

Output:

Pattern found at positions: [19]

String Data in Network Communication -

Question 1

Write a program that simulates sending a string over a network, encoding it to bytes and then decoding it back to a string.

String Data in Network Communication -

Question 1

Code:

```
import socket
import threading # Import threading module for concurrent
execution

# Server side
def server():
    host = "127.0.0.1"
    port = 12345
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
s:
    s.bind((host, port))
    s.listen()
    print("Server listening...")
    conn, addr = s.accept()
    with conn:
        print(f"Connected by {addr}")
        data = conn.recv(1024)
        received_string = data.decode("utf-8")
        print("Received:", received_string)
        conn.sendall(received_string.encode("utf-8"))

# Client side
def client():
    host = "127.0.0.1"
    port = 12345
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
s:
    s.connect((host, port))
    message = "Hello, Server!"
    s.sendall(message.encode("utf-8"))
    data = s.recv(1024)
    received_string = data.decode("utf-8")
    print("Received from server:", received_string)

if __name__ == "__main__":
    # Create threads to run server and client concurrently
    server_thread = threading.Thread(target=server)
```

```
client_thread = threading.Thread(target=client)

# Start server and client threads
server_thread.start()
client_thread.start()

# Wait for both threads to finish
server_thread.join()
client_thread.join()
```

Explanation:

This program simulates sending a string over a network. The server listens for incoming connections, receives a string from the client, decodes it, and sends it back. The client sends a string to the server, receives the response, and prints it.

Output:

```
Server listening... Connected by ('127.0.0.1', 50113) Received:  
Hello, Server! Received from server: Hello, Server!
```

String Data in Network Communication - Question 2

Create a script that reads a string message from a client socket, decodes it, and prints it to the console.

String Data in Network Communication - Question 2

Code:

```
import socket

def server():
    host = "127.0.0.1"
    port = 12345
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((host, port))
        s.listen()
        print("Server listening...")
        conn, addr = s.accept()
        with conn:
            print(f"Connected by {addr}")
            data = conn.recv(1024)
            received_string = data.decode("utf-8")
            print("Received:", received_string)

    if __name__ == "__main__":
        server()
```

Explanation:

This script sets up a server that listens for incoming connections, receives a string from a client, decodes it as UTF-8, and prints it to the console.

Output:

```
#Run this script and then run question 1 script in another cmd
window. Server listening... Connected by ('127.0.0.1', 50210)
Received: Hello, Server!
```

String Data in Network Communication - Question 3

Develop a function that sends a JSON string from a client to a server over a TCP connection.

String Data in Network Communication - Question 3

Code:

```
import socket
import json
import threading

def server():
    host = "127.0.0.1"
    port = 12345
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((host, port))
        s.listen()
        print("Server listening...")

    while True:
        conn, addr = s.accept()
        with conn:
            print(f"Connected by {addr}")
            data = conn.recv(1024)
            if not data:
                break # Exit the loop if no data is received
            received_data = data.decode("utf-8")
            json_data = json.loads(received_data)
            print("Received JSON data:", json_data)

def client():
    host = "127.0.0.1"
    port = 12345
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((host, port))
        data = {"message": "Hello, Server!", "name": "Client"}
        json_data = json.dumps(data)
        s.sendall(json_data.encode("utf-8"))

if __name__ == "__main__":
    server_thread = threading.Thread(target=server)
```

```
client_thread = threading.Thread(target=client)

server_thread.start()
client_thread.start()

server_thread.join()
client_thread.join()
```

Explanation:

This function sets up a client that sends a JSON string to a server over a TCP connection. The JSON data is encoded as UTF-8 before sending.

Output:

```
Server listening... Connected by ('127.0.0.1', 59065) Received
JSON data: {'message': 'Hello, Server!', 'name': 'Client'}
```

String Data in Network Communication - Question 4

Write a program that handles UTF-8 encoded string data received from a network socket.

String Data in Network Communication - Question 4

Code:

```
import socket
import threading

def server():
    host = "127.0.0.1"
    port = 12345
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((host, port))
        s.listen()
        print("Server listening...")
        conn, addr = s.accept()
        with conn:
            print(f"Connected by {addr}")
            data = conn.recv(1024)
            received_string = data.decode("utf-8")
            print("Received:", received_string)

def client():
    host = "127.0.0.1"
    port = 12345
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((host, port))
        message = "Hello, Server!"
        s.sendall(message.encode("utf-8"))

if __name__ == "__main__":
    server_thread = threading.Thread(target=server)
    client_thread = threading.Thread(target=client)

    server_thread.start()
    client_thread.start()

    server_thread.join()
    client_thread.join()
```

Explanation:

This program sets up a server that listens for incoming connections, receives a UTF-8 encoded string from a client, and prints it to the console.

Output:

```
Server listening... Connected by ('127.0.0.1', 59085) Received:  
Hello, Server!
```

String Data in Network Communication - Question 5

Create a server script that accepts string data, processes it, and sends a response back to the client.

String Data in Network Communication - Question 5

Code:

```
import socket
import threading

def server():
    host = "127.0.0.1"
    port = 12345
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((host, port))
        s.listen()
        print("Server listening...")
        conn, addr = s.accept()
        with conn:
            print(f"Connected by {addr}")
            data = conn.recv(1024)
            received_string = data.decode("utf-8")
            print("Received:", received_string)
            response = f"Server processed: {received_string}"
            conn.sendall(response.encode("utf-8"))

def client():
    host = "127.0.0.1"
    port = 12345
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((host, port))
        message = "Hello, Server!"
        s.sendall(message.encode("utf-8"))

if __name__ == "__main__":
    server_thread = threading.Thread(target=server)
    client_thread = threading.Thread(target=client)

    server_thread.start()
    client_thread.start()

    server_thread.join()
```

```
client_thread.join()
```

Explanation:

This server script listens for incoming connections, receives a string from a client, processes it, sends a response back to the client, and prints both the received and processed strings.

Output:

```
Server listening... Connected by ('127.0.0.1', 59113) Received:  
Hello, Server!
```

String Data in Network Communication - Question 6

Write a script that demonstrates error handling for incorrectly encoded string data received over a network.

String Data in Network Communication - Question 6

Code:

```
import socket
import threading

def server():
    host = "127.0.0.1"
    port = 12345
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((host, port))
        s.listen()
        print("Server listening...")
        conn, addr = s.accept()
        with conn:
            print(f"Connected by {addr}")
            try:
                data = conn.recv(1024)
                received_string = data.decode("utf-8")
                print("Received:", received_string)
            except UnicodeDecodeError as e:
                print("Error decoding received data:", e)

def client():
    host = "127.0.0.1"
    port = 12345
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((host, port))
        message = "Hello, Server!"
        s.sendall(message.encode("utf-8"))

if __name__ == "__main__":
    server_thread = threading.Thread(target=server)
    client_thread = threading.Thread(target=client)

    server_thread.start()
    client_thread.start()
```

```
server_thread.join()  
client_thread.join()
```

Explanation:

This script sets up a server that listens for incoming connections, attempts to receive and decode a UTF-8 encoded string from a client, and handles a `UnicodeDecodeError` if the received data cannot be correctly decoded.

Output:

```
Server listening... Connected by ('127.0.0.1', 59143) Received:  
Hello, Server!
```

String Data in Network Communication - Question 7

Develop a program that compresses a string using zlib before sending it over a network and decompresses it on receipt.

String Data in Network Communication - Question 7

Code:

```
import socket
import zlib
import threading

def server():
    host = "127.0.0.1"
    port = 12345
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((host, port))
        s.listen()
        print("Server listening...")
        conn, addr = s.accept()
        with conn:
            print(f"Connected by {addr}")
            data = conn.recv(1024)
            decompressed_data = zlib.decompress(data)
            received_string = decompressed_data.decode("utf-8")
            print("Received:", received_string)

def client():
    host = "127.0.0.1"
    port = 12345
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((host, port))
        message = "Compress me before sending!"
        compressed_data = zlib.compress(message.encode("utf-8"))
        s.sendall(compressed_data)

if __name__ == "__main__":
    server_thread = threading.Thread(target=server)
    client_thread = threading.Thread(target=client)

    server_thread.start()
    client_thread.start()
```

```
server_thread.join()
client_thread.join()
```

Explanation:

This program demonstrates data compression and decompression using the zlib library. The server receives compressed data, decompresses it, and then decodes it to retrieve the original string.

Output:

```
Server listening... Connected by ('127.0.0.1', 59155) Received:
Compress me before sending!
```

String Data in Network Communication - Question 8

Create a function that encrypts a string message before sending it over a network and decrypts it upon receipt.

String Data in Network Communication - Question 8

Code:

```
import socket
from cryptography.fernet import Fernet
import threading

# Generate a random key for encryption
key = Fernet.generate_key()
cipher_suite = Fernet(key)

def encrypt_message(message):
    return cipher_suite.encrypt(message.encode("utf-8"))

def decrypt_message(ciphertext):
    return cipher_suite.decrypt(ciphertext).decode("utf-8")

def server():
    host = "127.0.0.1"
    port = 12345
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((host, port))
        s.listen()
        print("Server listening...")
        conn, addr = s.accept()
        with conn:
            print(f"Connected by {addr}")
            data = conn.recv(1024)
            received_string = decrypt_message(data)
            print("Received:", received_string)

def client():
    host = "127.0.0.1"
    port = 12345
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((host, port))
        message = "Encrypt me before sending!"
        encrypted_message = encrypt_message(message)
```

```
s.sendall(encrypted_message)

if __name__ == "__main__":
    server_thread = threading.Thread(target=server)
    client_thread = threading.Thread(target=client)

    server_thread.start()
    client_thread.start()

    server_thread.join()
    client_thread.join()
```

Explanation:

This program demonstrates data encryption and decryption using the Fernet encryption scheme. The client encrypts a message before sending it to the server, which then decrypts and processes the received data.

Output:

```
#Make sure you have the cryptography module installed in your
environment: "pip install cryptography" Server listening...
Connected by ('127.0.0.1', 59192) Received: Encrypt me before
sending!
```

String Data in Network Communication - Question 9

Write a script that simulates a chat application, handling incoming and outgoing messages as strings.

String Data in Network Communication - Question 9

Code:

```
import socket
import threading
import signal
import sys

# Global flag for running the server and client
running = True

# Server Code
def client_handler(client_socket, client_address):
    global running
    while running:
        try:
            message = client_socket.recv(1024).decode('utf-8')
            if message:
                print(f"[{client_address}] {message}")
                response = f"Echo: {message}"
                client_socket.send(response.encode('utf-8'))
                if message.lower() == 'exit':
                    break
        except ConnectionResetError:
            break
    client_socket.close()

def start_server():
    global running
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(('127.0.0.1', 12345))
    server.listen()

    print("Server listening on 127.0.0.1:12345")
    while running:
        client_socket, client_address = server.accept()
        print(f"Connection from {client_address} has been
established.")
        thread = threading.Thread(target=client_handler, args=
(client_socket, client_address))
```

```

        thread.start()
server.close()

# Client Code
def start_client():
    global running
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client.connect(('127.0.0.1', 12345))

    while running:
        message = input("You: ")
        if message.lower() == 'exit':
            running = False
            break
        client.send(message.encode('utf-8'))
        response = client.recv(1024).decode('utf-8')
        print(f"Server: {response}")
    client.close()

# Signal Handler for Graceful Shutdown
def signal_handler(sig, frame):
    global running
    print('Shutting down gracefully...')
    running = False

# Running Server and Client
if __name__ == "__main__":
    signal.signal(signal.SIGINT, signal_handler)

    server_thread = threading.Thread(target=start_server)
    server_thread.daemon = True # Server runs in background
    server_thread.start()

    client_thread = threading.Thread(target=start_client)
    client_thread.start()

    client_thread.join() # Wait for the client thread to
finish
    print("Application closed.")

```

Explanation:

This script simulates a basic chat application where clients can send and receive text messages. It handles incoming and outgoing messages as strings encoded in UTF-8.

Output:

```
Server listening on 127.0.0.1:12345 Connection from
('127.0.0.1', 51179) has been established. You: Hello
[('127.0.0.1', 51179)] Hello Server: Echo: Hello You: exit
Application closed.
```

String Data in Network Communication - Question 10

Develop a program that uses HTTP headers in string format to exchange information between a client and server.

String Data in Network Communication - Question 10

Code:

```
import threading
import http.server
import requests
import time

# HTTP Server Handler
class SimpleHTTPRequestHandler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/plain')
        self.send_header('Custom-Header', 'SomeValue')
        self.end_headers()
        self.wfile.write(b'Hello from the server!')

# Function to start the server
def start_server():
    server_address = ('', 8000)
    httpd = http.server.HTTPServer(server_address,
SimpleHTTPRequestHandler)
    httpd.serve_forever()

# Function to run the client
def run_client():
    time.sleep(1) # Wait for the server to start
    response = requests.get('http://localhost:8000')
    print('Response:', response.text)
    print('Custom Header:', response.headers.get('Custom-Header'))

# Start the server in a new thread
server_thread = threading.Thread(target=start_server)
server_thread.daemon = True # This ensures the thread exits
when the main program does
server_thread.start()

# Run the client in the main thread
```

```
run_client()
```

Explanation:

This program demonstrates a simple HTTP server and client interaction using string-formatted HTTP headers. The client sends an HTTP GET request, and the server responds with an HTTP 200 OK response containing an HTML page.

Output:

```
127.0.0.1 - - [20/Dec/2023 13:01:07] "GET / HTTP/1.1" 200 -
Response: Hello from the server! Custom Header: SomeValue
```

String Encoding Schemes - Question 1

Write a program that encodes a string into UTF-8 and then decodes it back to ensure data integrity.

String Encoding Schemes - Question 1

Code:

```
def encode_decode_utf8(input_string):
    # Encoding the string to UTF-8
    utf8_bytes = input_string.encode("utf-8")

    # Decoding the UTF-8 bytes back to a string
    decoded_string = utf8_bytes.decode("utf-8")

    return decoded_string

# Example usage:
input_string = "Hello, UTF-8 Encoding and Decoding!"
decoded_result = encode_decode_utf8(input_string)

print(f"Original string: {input_string}")
print(f"Decoded string: {decoded_result}")

# Output:
# Original string: Hello, UTF-8 Encoding and Decoding!
# Decoded string: Hello, UTF-8 Encoding and Decoding!
```

Explanation:

This program demonstrates encoding a string into UTF-8 and then decoding it back to ensure data integrity. Encoding and decoding the string should result in the same original string, confirming that the data has not been altered during the process.

Output:

```
Original string: Hello, UTF-8 Encoding and Decoding! Decoded
string: Hello, UTF-8 Encoding and Decoding!
```

String Encoding Schemes - Question 2

Create a script that converts a string into ASCII encoding, handling any encoding errors gracefully.

String Encoding Schemes - Question 2

Code:

```
def convert_to_ascii(input_string):
    try:
        ascii_encoded = input_string.encode("ascii",
errors="replace")
        return ascii_encoded.decode("ascii")
    except UnicodeEncodeError:
        return "Encoding error: Some characters cannot be
represented in ASCII."

# Example usage:
input_string = "Hello, ASCII Encoding!"
ascii_result = convert_to_ascii(input_string)

print(f"Original string: {input_string}")
print(f"ASCII encoded: {ascii_result}")

# Output:
# Original string: Hello, ASCII Encoding!
# ASCII encoded: Hello, ASCII Encoding!
```

Explanation:

This script converts a string into ASCII encoding while handling encoding errors gracefully. If there are characters in the input string that cannot be represented in ASCII, it replaces them with the "?" character (or another specified replacement character) to avoid encoding errors.

Output:

```
Original string: Hello, ASCII Encoding! ASCII encoded: Hello,
ASCII Encoding!
```

String Encoding Schemes - Question 3

Develop a function that reads a text file encoded in UTF-16 and prints its content as a Python string.

String Encoding Schemes - Question 3

Code:

```
def read_utf16_file(file_path):
    try:
        with open(file_path, "rb") as utf16_file:
            utf16_bytes = utf16_file.read()
            decoded_text = utf16_bytes.decode("utf-16")
            return decoded_text
    except FileNotFoundError:
        return "File not found"
    except UnicodeDecodeError:
        return "Error decoding UTF-16 data"

# Example usage:
file_path = "utf16_encoded.txt"
result = read_utf16_file(file_path)
print(result)
```

Explanation:

This function reads a text file encoded in UTF-16, handles potential file not found errors, and decodes the UTF-16 bytes to return the content as a Python string.

Output:

```
Original string: Hello, ASCII Encoding! ASCII encoded: Hello,  
ASCII Encoding!
```

String Encoding Schemes - Question 4

Write a program that compares the size of a string when encoded in UTF-8, UTF-16, and ASCII.

String Encoding Schemes - Question 4

Code:

```
def compare_encoding_sizes(input_string):
    utf8_size = len(input_string.encode("utf-8"))
    utf16_size = len(input_string.encode("utf-16"))
    ascii_size = len(input_string.encode("ascii"))

    return utf8_size, utf16_size, ascii_size

# Example usage:
input_string = "Hello, Encoding Sizes!"
utf8_size,          utf16_size,          ascii_size      =
compare_encoding_sizes(input_string)

print(f"Original string: {input_string}")
print(f"UTF-8 size: {utf8_size} bytes")
print(f"UTF-16 size: {utf16_size} bytes")
print(f"ASCII size: {ascii_size} bytes")

# Output:
# Original string: Hello, Encoding Sizes!
# UTF-8 size: 22 bytes
# UTF-16 size: 46 bytes
# ASCII size: 22 bytes
```

Explanation:

This program compares the sizes of a string when encoded in three different encodings: UTF-8, UTF-16, and ASCII. It demonstrates that the size of the encoded string varies depending on the encoding used.

Output:

```
Original string: Hello, Encoding Sizes! UTF-8 size: 22 bytes
UTF-16 size: 46 bytes ASCII size: 22 bytes
```

String Encoding Schemes - Question 5

Create a script that encodes a string with special characters (like emojis) in UTF-8 and then decodes it.

String Encoding Schemes - Question 5

Code:

```
def encode_decode_special_characters(input_string):
    utf8_encoded = input_string.encode("utf-8")
    decoded_string = utf8_encoded.decode("utf-8")
    return decoded_string

# Example usage:
input_string = "Hello, ● and ●!"
decoded_result = encode_decode_special_characters(input_string)

print(f"Original string: {input_string}")
print(f"Decoded string: {decoded_result}")

# Output:
# Original string: Hello, ● and ●!
# Decoded string: Hello, ● and ●!
```

Explanation:

This script encodes a string containing special characters (like emojis) in UTF-8 and then decodes it. It demonstrates that UTF-8 encoding and decoding can handle special characters correctly.

Output:

```
Original string: Hello, ● and ●! Decoded string: Hello, ● and ●!
```

String Encoding Schemes - Question 6

Write a function that detects the encoding of a given byte string and converts it to a Python string.

String Encoding Schemes - Question 6

Code:

```
def detect_and_decode(byte_string):
    encodings_to_try = ["utf-8", "utf-16", "ascii"]
    for encoding in encodings_to_try:
        try:
            decoded_string = byte_string.decode(encoding)
            return decoded_string, encoding
        except UnicodeDecodeError:
            continue
    return None, None

# Example usage:
byte_string = b"This is a byte string encoded in utf-8"
decoded_string, detected_encoding = detect_and_decode(byte_string)
if decoded_string is not None:
    print(f"Detected encoding: {detected_encoding}")
    print(f"Decoded string: {decoded_string}")
else:
    print("Unable to detect encoding.")
```

Explanation:

The `detect_and_decode` function attempts to detect the encoding of a given byte string by trying different encodings (UTF-8, UTF-16, and ASCII) and returning the first successful decoding along with the detected encoding. This allows you to safely convert a byte string to a Python string with the correct encoding.

Output:

```
Detected encoding: utf-8 Decoded string: This is a byte string
encoded in utf-8
```

String Encoding Schemes - Question 7

Develop a program that converts a string to bytes using Base64 encoding and then back to a string.

String Encoding Schemes - Question 7

Code:

```
import base64

def string_to_base64(input_string):
    byte_string = input_string.encode("utf-8")
    base64_bytes = base64.b64encode(byte_string)
    return base64_bytes

def base64_to_string(base64_bytes):
    byte_string = base64.b64decode(base64_bytes)
    decoded_string = byte_string.decode("utf-8")
    return decoded_string

# Example usage:
input_string = "Hello, Base64!"
base64_encoded = string_to_base64(input_string)
decoded_string = base64_to_string(base64_encoded)

print(f"Original string: {input_string}")
print(f"Base64 encoded: {base64_encoded}")
print(f"Decoded string: {decoded_string}")

# Output:
# Original string: Hello, Base64!
# Base64 encoded: b'SGVsbG8sIEJhc2U2NCE='
# Decoded string: Hello, Base64!
```

Explanation:

This program demonstrates the conversion of a string to bytes using Base64 encoding and then back to a string. Base64 encoding is commonly used for safely encoding binary data as text, and this example shows how to perform the encoding and decoding steps.

Output:

```
Original string: Hello, Base64! Base64 encoded:
b'SGVsbG8sIEJhc2U2NCE=' Decoded string: Hello, Base64!
```

String Encoding Schemes - Question 8

Create a script that demonstrates the difference in handling strings with UTF-8 and UTF-16 encodings.

String Encoding Schemes - Question 8

Code:

```
def utf8_vs_utf16():
    text = "Hello, Unicode! 你好 , Unicode !"
    utf8_encoded = text.encode("utf-8")
    utf16_encoded = text.encode("utf-16")

    print(f"Original text: {text}")
    print(f"UTF-8 encoded: {utf8_encoded}")
    print(f"UTF-16 encoded: {utf16_encoded}")

    utf8_decoded = utf8_encoded.decode("utf-8")
    utf16_decoded = utf16_encoded.decode("utf-16")

    print(f"Decoded from UTF-8: {utf8_decoded}")
    print(f"Decoded from UTF-16: {utf16_decoded}")

# Call the function
utf8_vs_utf16()
```

Explanation:

This script demonstrates the difference in handling strings with UTF-8 and UTF-16 encodings. It encodes a Unicode text in both UTF-8 and UTF-16, showing the differences in encoded bytes. Then, it decodes both encodings back to the original text, demonstrating that UTF-8 and UTF-16 encodings can correctly represent the same text.

Output:

```
Original text: Hello, Unicode! 你好 , Unicode !
UTF-8 encoded:
b'Hello, Unicode! 你好 , Unicode !'
UTF-16 encoded:
b'\ufffe\ufe0e\x00e\x00l\x00l\x00o\x00,\x00
\x00U\x00n\x00i\x00c\x00o\x00d\x00e\x00!\x00
\x00&\x00#\x002\x000\x003\x002\x000\x00;\x00&\x00#\x002\x002\x00
\x09\x000\x009\x00;\x00&\x00#\x006\x005\x002\x009\x002\x00;\x00U\
\x00n\x00i\x00c\x00o\x00d\x00e\x00&\x00#\x006\x005\x002\x008\x00
```

1\x00;\x00' Decoded from UTF-8: Hello, Unicode! 你好 , Unicode !
Decoded from UTF-16: Hello, Unicode! 你好 , Unicode !

String Encoding Schemes - Question 9

Write a function that takes a string, encodes it in UTF-8, and then encodes the UTF-8 bytes in hexadecimal format.

String Encoding Schemes - Question 9

Code:

```
def encode_to_utf8_hex(input_string):
    utf8_encoded = input_string.encode("utf-8")
    utf8_hex_encoded = utf8_encoded.hex()
    return utf8_hex_encoded

# Example usage:
input_string = "Hello, UTF-8 to Hex!"
utf8_hex_encoded = encode_to_utf8_hex(input_string)

print(f"Original string: {input_string}")
print(f"UTF-8 Hex encoded: {utf8_hex_encoded}")

# Output:
# Original string: Hello, UTF-8 to Hex!
# UTF-8 Hex encoded: 48656c6c6f2c205554462d3820746f2048657821
```

Explanation:

This function takes a string, encodes it in UTF-8, and then encodes the UTF-8 bytes in hexadecimal format. Hexadecimal encoding is often used for representing binary data as a sequence of hexadecimal characters.

Output:

```
Original string: Hello, UTF-8 to Hex!  UTF-8 Hex encoded:
48656c6c6f2c205554462d3820746f2048657821
```

String Encoding Schemes - Question 10

Develop a program that reads a binary file (like an image) as a byte string, encodes it in UTF-8, and handles any errors that occur.

String Encoding Schemes - Question 10

Code:

```
def read_binary_file(file_path):
    try:
        with open(file_path, "rb") as binary_file:
            binary_data = binary_file.read()
            utf8_encoded = binary_data.decode("utf-8")
            return utf8_encoded
    except FileNotFoundError:
        return "File not found"
    except UnicodeDecodeError:
        return "Error decoding binary data to UTF-8"

# Example usage:
file_path = "image.png"
result = read_binary_file(file_path)
print(result)
```

Explanation:

This program reads a binary file (such as an image) as a byte string and attempts to encode it in UTF-8. If the file is not found, it handles the `FileNotFoundError`, and if there is an error decoding binary data to UTF-8, it handles the `UnicodeDecodeError`. This is a practical approach to handle binary data that may not be compatible with UTF-8 encoding.

Output:

This is an example text file. It contains some text data. You can replace this text with your own content.

Strings and File Handling - Question 1

Write a program that reads a text file line by line and prints each line as a string.

Strings and File Handling - Question 1

Code:

```
def read_file_line_by_line(file_path):
    try:
        with open(file_path, "r") as file:
            for line in file:
                print(line.strip())
    except FileNotFoundError:
        print("File not found")

    # Example usage:
file_path = "sample.txt"
read_file_line_by_line(file_path)
```

Explanation:

This program reads a text file line by line and prints each line as a string. It uses the `open` function to open the file in read mode and iterates through the lines, stripping any leading or trailing whitespace.

Output:

Input File: Hello World! Output File:Hello World!

Strings and File Handling - Question 2

Create a script that writes a list of strings to a file, with each string on a separate line.

Strings and File Handling - Question 2

Code:

```
def write_strings_to_file(file_path, string_list):
    try:
        with open(file_path, "w") as file:
            for string in string_list:
                file.write(string + "\n")
    except FileNotFoundError:
        print("File not found")

    # Example usage:
file_path = "output.txt"
string_list = ["Line 1", "Line 2", "Line 3"]
write_strings_to_file(file_path, string_list)
```

Explanation:

This script writes a list of strings to a file, with each string on a separate line. It uses the `open` function with write mode ("w") to create or overwrite the file and writes each string followed by a newline character.

Output:

Line 1 Line 2 Line 3

Strings and File Handling - Question 3

Develop a function that reads a file and returns its content as a single string.

Strings and File Handling - Question 3

Code:

```
def read_file_as_string(file_path):
    try:
        with open(file_path, "r") as file:
            file_content = file.read()
            return file_content
    except FileNotFoundError:
        return "File not found"

# Example usage:
file_path = "sample.txt"
file_content = read_file_as_string(file_path)
print(file_content)
```

Explanation:

This function reads a file and returns its content as a single string using the `read` method. It handles the case where the file is not found by returning an appropriate message.

Output:

```
#Input File: Line 1 Line 2 Line 3 Output File: Line 1 Line 2
Line 3
```

Strings and File Handling - Question 4

Write a program that appends a string to the end of a text file.

Strings and File Handling - Question 4

Code:

```
def append_string_to_file(file_path, string_to_append):
    try:
        with open(file_path, "a") as file:
            file.write(string_to_append)
    except FileNotFoundError:
        print("File not found")

    # Example usage:
file_path = "output.txt"
string_to_append = "Appended line."
append_string_to_file(file_path, string_to_append)
```

Explanation:

This program appends a string to the end of a text file using the "a" mode when opening the file with the `open` function. It allows you to add content to an existing file without overwriting it.

Output:

Line 1 Line 2 Line 3 Appended line.

Strings and File Handling - Question 5

Create a script that reads a text file and counts the occurrence of a particular word.

Strings and File Handling - Question 5

Code:

```
def count_word_occurrence(file_path, word_to_count):
    try:
        with open(file_path, "r") as file:
            file_content = file.read()
            word_count      =
file_content.lower().count(word_to_count.lower())
            return word_count
    except FileNotFoundError:
        return "File not found"

# Example usage:
file_path = "sample.txt"
word_to_count = "line"
occurrences = count_word_occurrence(file_path, word_to_count)
print(f"The word '{word_to_count}' appears {occurrences} times
in the file.")
```

Explanation:

This script reads a text file and counts the occurrences of a particular word (case-insensitive). It uses the `count` method to count the occurrences of the word in the file's content.

Output:

The word 'line' appears 3 times in the file.

Strings and File Handling - Question 6

Write a function that opens a file with UTF-8 encoding and handles any encoding errors.

Strings and File Handling - Question 6

Code:

```
def read_file_utf8(file_path):
    try:
        with open(file_path, "r", encoding="utf-8") as file:
            file_content = file.read()
        return file_content
    except FileNotFoundError:
        return "File not found"
    except UnicodeDecodeError:
        return "Unable to decode the file with UTF-8 encoding"

# Example usage:
file_path = "sample.txt"
file_content = read_file_utf8(file_path)
print(file_content)
```

Explanation:

This function opens a file with UTF-8 encoding and handles any encoding errors that may occur. If the file is not found or cannot be decoded with UTF-8, it returns an appropriate error message.

Output:

Line 1 Line 2 Line 3

Strings and File Handling - Question 7

Develop a program that reads a binary file (like an image) and converts it to a Base64 encoded string.

Strings and File Handling - Question 7

Code:

```
import base64

def binary_file_to_base64(file_path):
    try:
        with open(file_path, "rb") as binary_file:
            binary_data = binary_file.read()
            base64_data = base64.b64encode(binary_data).decode("utf-8")
        return base64_data
    except FileNotFoundError:
        return "File not found"

# Example usage:
file_path = "image.jpg"
base64_string = binary_file_to_base64(file_path)
print(f"Base64 encoded image:\n{base64_string[:100]}...") # Print the first 100 characters
```

Explanation:

This program reads a binary file (such as an image) and converts it to a Base64 encoded string. It uses the `base64` module to perform the encoding.

Output:

```
Base64 encoded image:
/9j/4AAQSkZJRgABAQEAYABgAAD/4QLaRXhpZgAATU0AKgAAAAgABAE7AAIAAAA
FAAABSodpAAQAAAABAAABUJydAAEAAAACKAAAC...
```

Strings and File Handling - Question 8

Create a script that writes a string to a file and then reads the file back to verify the written content.

Strings and File Handling - Question 8

Code:

```
def write_and_verify(file_path, content_to_write):
    try:
        # Write content to the file
        with open(file_path, "w") as file:
            file.write(content_to_write)

        # Read content from the file
        with open(file_path, "r") as file:
            read_content = file.read()

        return read_content
    except FileNotFoundError:
        return "File not found"

# Example usage:
file_path = "output.txt"
content_to_write = "This is a test."
read_content = write_and_verify(file_path, content_to_write)
print(f"Written content: {content_to_write}")
print(f"Read content: {read_content}")
```

Explanation:

This script writes a string to a file and then reads the file back to verify the written content. It uses separate `open` calls for writing and reading.

Output:

Written content: This is a test. Read content: This is a test.

Strings and File Handling - Question 9

Write a function that implements a simple file-based text search, returning lines that contain a given string.

Strings and File Handling - Question 9

Code:

```
def file_text_search(file_path, search_string):
    try:
        matching_lines = []
        with open(file_path, "r") as file:
            for line in file:
                if search_string in line:
                    matching_lines.append(line.strip())
    return matching_lines
except FileNotFoundError:
    return ["File not found"]

# Example usage:
file_path = "sample.txt"
search_string = "Line"
matching_lines = file_text_search(file_path, search_string)
for line in matching_lines:
    print(line)
```

Explanation:

This function performs a simple file-based text search, returning lines that contain a given search string. It reads each line of the file and checks if the search string is present.

Output:

Line 1 Line 2 Line 3

Strings and File Handling - Question 10

Develop a program that reads a CSV file into a string and then processes it to create a list of dictionaries.

Strings and File Handling - Question 10

Code:

```
import csv

def read_csv_to_list_of_dicts(file_path):
    try:
        data = []
        with open(file_path, "r") as file:
            csv_reader = csv.DictReader(file)
            for row in csv_reader:
                data.append(row)
        return data
    except FileNotFoundError:
        return []

# Example usage:
file_path = "data.csv"
data = read_csv_to_list_of_dicts(file_path)
for record in data:
    print(record)
```

Explanation:

This program reads a CSV file into a string and then processes it to create a list of dictionaries using the `csv` module. It assumes that the CSV file has a header row with column names.

Output:

```
Input File: id,title,author,genre,price
1,The Great Gatsby,F. Scott Fitzgerald,Fiction,10.99
2,To Kill a Mockingbird,Harper Lee,Fiction,8.99
3,1984,George Orwell,Science Fiction,9.50
4,Becoming,Michelle Obama,Biography,11.25
5,The Catcher in the Rye,J.D. Salinger,Fiction,7.20
6,Sapiens,Yuval Noah Harari,Non-Fiction,12.80
7,The Alchemist,Paulo Coelho,Fiction,6.80
8,Educated,Tara Westover,Biography,9.99
9,Invisible Man,Ralph Ellison,Fiction,8.50
10,The Silent Patient,Alex Michaelides,Thriller,10.00
Output: {'id': '1', 'title': 'The Great Gatsby', 'author': 'F. Scott Fitzgerald', 'genre': 'Fiction', 'price': '10.99'}
{'id': '2', 'title': 'To Kill a
```

```
Mockingbird', 'author': 'Harper Lee', 'genre': 'Fiction',  
'price': '8.99'} {'id': '3', 'title': '1984', 'author': 'George  
Orwell', 'genre': 'Science Fiction', 'price': '9.50'} {'id':  
'4', 'title': 'Becoming', 'author': 'Michelle Obama', 'genre':  
'Biography', 'price': '11.25'} {'id': '5', 'title': 'The  
Catcher in the Rye', 'author': 'J.D. Salinger', 'genre':  
'Fiction', 'price': '7.20'} {'id': '6', 'title': 'Sapiens',  
'author': 'Yuval Noah Harari', 'genre': 'Non-Fiction', 'price':  
'12.80'} {'id': '7', 'title': 'The Alchemist', 'author': 'Paulo  
Coelho', 'genre': 'Fiction', 'price': '6.80'} {'id': '8',  
'title': 'Educated', 'author': 'Tara Westover', 'genre':  
'Biography', 'price': '9.99'} {'id': '9', 'title': 'Invisible  
Man', 'author': 'Ralph Ellison', 'genre': 'Fiction', 'price':  
'8.50'} {'id': '10', 'title': 'The Silent Patient', 'author':  
'Alex Michaelides', 'genre': 'Thriller', 'price': '10.00'}
```

String Performance Considerations - Question 1

Write a program that compares the performance of string concatenation using `+` versus using a `StringIO` buffer.

String Performance Considerations - Question 1

Code:

```
import time
from io import StringIO

# Using string concatenation with +
def string_concatenation(num_iterations):
    result = ""
    for _ in range(num_iterations):
        result += "hello"
    return result

# Using StringIO buffer
def string_io_buffer_concatenation(num_iterations):
    buffer = StringIO()
    for _ in range(num_iterations):
        buffer.write("hello")
    result = buffer.getvalue()
    buffer.close()
    return result

# Measure execution time
num_iterations = 100000
start_time = time.time()
string_concatenation(num_iterations)
end_time = time.time()
print(f"String Concatenation with +: {end_time - start_time} seconds")

start_time = time.time()
string_io_buffer_concatenation(num_iterations)
end_time = time.time()
print(f"StringIO Buffer Concatenation: {end_time - start_time} seconds")
```

Explanation:

This program compares the performance of string concatenation using `+` with using a `StringIO` buffer. The `StringIO` buffer approach is significantly faster for large numbers of concatenations.

Output:

```
String Concatenation with +: 0.009848594665527344 seconds
StringIO Buffer Concatenation: 0.010567188262939453 seconds
```

String Performance Considerations - Question 2

Create a script that measures the execution time of building a large string using list comprehension and `join()` versus using a loop with `+=`.

String Performance Considerations - Question 2

Code:

```
import time

# Using list comprehension and join
def build_large_string_with_join(num_iterations):
    words = ["hello"] * num_iterations
    large_string = "".join(words)
    return large_string

# Using loop with +=
def build_large_string_with_loop(num_iterations):
    large_string = ""
    for _ in range(num_iterations):
        large_string += "hello"
    return large_string

# Measure execution time
num_iterations = 100000
start_time = time.time()
build_large_string_with_join(num_iterations)
end_time = time.time()
print(f"Build Large String with join(): {end_time - start_time} seconds")

start_time = time.time()
build_large_string_with_loop(num_iterations)
end_time = time.time()
print(f"Build Large String with +=: {end_time - start_time} seconds")
```

Explanation:

This script measures the execution time of building a large string using list comprehension and `join()` versus using a loop with `+=`. The `join()` approach is significantly faster for concatenating large numbers of strings.

Output:

```
Build Large String with join(): 0.002432107925415039 seconds
Build Large String with +=: 0.008912086486816406 seconds
```

String Performance Considerations - Question 3

Develop a function that optimizes the processing of a large number of strings for memory usage.

String Performance Considerations - Question 3

Code:

```
import time

def process_large_number_of_strings(strings):
    result = ""
    for string in strings:
        result += string
    return result

def optimized_process_large_number_of_strings(strings):
    result = []
    for string in strings:
        result.append(string)
    return "".join(result)

# Create a large list of strings for testing
string_list = ["Hello", " ", "World", "! "] * 10000      #
Replicate the list many times to make it large

# Measure time for the original function
start_time = time.time()
process_large_number_of_strings(string_list)
end_time = time.time()
print(f"Original method: {end_time - start_time} seconds")

# Measure time for the optimized function
start_time = time.time()
optimized_process_large_number_of_strings(string_list)
end_time = time.time()
print(f"Optimized method: {end_time - start_time} seconds")
```

Explanation:

The first function processes a large number of strings by concatenating them directly, which can lead to high memory usage. The improved function optimizes memory usage by appending the strings to a list and then using `join()` to

concatenate them, reducing the number of temporary string objects.

Output:

```
Original method: 0.0014793872833251953 seconds
Optimized
method: 0.001033782958984375 seconds
```

String Performance Considerations - Question 4

Write a program that benchmarks the performance of checking membership in a long string versus a set of characters.

String Performance Considerations - Question 4

Code:

```
import time

# Function to check membership in a long string
def membership_in_long_string(long_string, target_char):
    return target_char in long_string

# Function to check membership in a set of characters
def membership_in_set(char_set, target_char):
    return target_char in char_set

# Benchmark performance
long_string = "abcdefghijklmnopqrstuvwxyz" * 10000
char_set = set(long_string)
target_char = "z"

# Membership check in long string
start_time = time.time()
result_string_check = membership_in_long_string(long_string,
target_char)
end_time = time.time()
time_string_check = end_time - start_time

# Membership check in set of characters
start_time = time.time()
result_set_check = membership_in_set(char_set, target_char)
end_time = time.time()
time_set_check = end_time - start_time

print(f"Membership Check in Long String: {time_string_check} seconds, Result: {result_string_check}")
print(f"Membership Check in Set of Characters: {time_set_check} seconds, Result: {result_set_check}")
```

Explanation:

This program benchmarks the performance of checking membership of a character in a long string versus a set of characters. Membership checks in a set are significantly faster than in a long string.

Output:

Membership Check in Long String: 1.1920928955078125e-06 seconds, Result: True Membership Check in Set of Characters: 9.5367431640625e-07 seconds, Result: True

String Performance Considerations - Question 5

Create a script that demonstrates the performance difference between using immutable strings and mutable `bytarray` for modifying large strings.

String Performance Considerations - Question 5

Code:

```
import time

# Using immutable strings
def modify_immutable_string(input_string, char_to_replace,
                             replacement_char):
    return input_string.replace(char_to_replace,
                                replacement_char)

# Using mutable bytearray
def modify_mutable_bytearray(input_bytearray, char_to_replace,
                             replacement_char):
    for i in range(len(input_bytearray)):
        if input_bytearray[i] == ord(char_to_replace):
            input_bytearray[i] = ord(replacement_char)

# Measure execution time
large_string = "a" * 1000000
char_to_replace = "a"
replacement_char = "b"

# Modify immutable string
start_time = time.time()
modify_immutable_string(large_string, char_to_replace,
                        replacement_char)
end_time = time.time()
print(f"Modify Immutable String: {end_time - start_time} seconds")

# Modify mutable bytearray
bytearray_string = bytearray(large_string, "utf-8")
start_time = time.time()
modify_mutable_bytearray(bytearray_string, char_to_replace,
                        replacement_char)
end_time = time.time()
print(f"Modify Mutable Bytearray: {end_time - start_time} seconds")
```

Explanation:

This script demonstrates the performance difference between using immutable strings and mutable `bytearray` for modifying large strings. Mutable `bytearray` is faster for in-place modifications.

Output:

```
Modify Immutable String: 0.0016012191772460938 seconds Modify  
Mutable Bytearray: 0.08346271514892578 seconds
```

String Performance Considerations - Question 6

Write a function that efficiently removes all whitespace from a very large string.

String Performance Considerations - Question 6

Code:

```
def remove_whitespace(input_string):
    return "".join(input_string.split())

# Test the function
large_string = "    This is a large string with whitespace.    "
result = remove_whitespace(large_string)
print(result)
```

Explanation:

The `remove_whitespace` function efficiently removes all whitespace characters (including spaces, tabs, and newlines) from a very large string by splitting the string into words using `split()` and then joining the words without spaces.

Output:

Thisisalargeststringwithwhitespace.

String Performance Considerations - Question 7

Develop a program that uses memory profiling to compare the memory usage of different string concatenation methods.

String Performance Considerations - Question

7

Code:

```
import memory_profiler

@profile
def concatenate_strings(num_iterations):
    result = ""
    for i in range(num_iterations):
        result += "hello"
    return result

@profile
def join_strings(num_iterations):
    words = ["hello"] * num_iterations
    return "".join(words)

if __name__ == "__main__":
    num_iterations = 100000
    concatenate_strings(num_iterations)
    join_strings(num_iterations)
```

Explanation:

This program uses the `memory_profiler` module to profile the memory usage of different string concatenation methods (`+=` and `join()`) for a large number of iterations. It helps compare memory consumption between the two approaches.

Output:

```
#Make sure the module is installed: pip install memory_profiler
#run the following command: python -m memory_profiler test.py
Line #  Mem usage Increment  Occurrences   Line Contents
===== 3
22.359 MiB  22.359 MiB      1 @profile 4 def
concatenate_strings(num_iterations): 5 22.363 MiB 0.004 MiB 1
result = "" 6 24.004 MiB -31.945 MiB 100001 for i in
range(num_iterations): 7 24.004 MiB -30.312 MiB 100000 result
+= "hello" 8 24.004 MiB 0.000 MiB 1 return result
Filename:
```

```
test.py Line # Mem usage Increment Occurrences Line Contents
=====
10    22.570    MiB   22.570    MiB     1    @profile    11    def
join_strings(num_iterations): 12 23.336 MiB 0.766 MiB 1 words =
["hello"] * num_iterations 13 23.816 MiB 0.480 MiB 1 return
"".join(words)
```

String Performance Considerations - Question 8

Create a script that optimizes the conversion of a large list of numbers into a string representation.

String Performance Considerations - Question 8

Code:

```
def optimize_number_list_to_string(numbers):
    result = " ".join(map(str, numbers))
    return result

# Test the function
large_list = list(range(1, 100001))
result = optimize_number_list_to_string(large_list)
print(result)
```

Explanation:

The `optimize_number_list_to_string` function efficiently converts a large list of numbers into a space-separated string representation using `map()` and `join()`, which is more memory-efficient than string concatenation.

Output:

```
#not all numbers shown: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58
59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 . .
```

.

String Performance Considerations - Question 9

Write a function that minimizes the use of temporary strings in a complex string manipulation task.

String Performance Considerations - Question 9

Code:

```
def minimize_temporary_strings(input_string):
    result = []
    for char in input_string:
        if char.isalpha():
            result.append(char)
    return "".join(result)

# Test the function
input_str = "Hello, World! This is a string with punctuation."
result = minimize_temporary_strings(input_str)
print(result)
```

Explanation:

The `minimize_temporary_strings` function minimizes the use of temporary strings in a complex string manipulation task by iterating over the input string character by character and appending only alphabetic characters to a list. It then joins the list to form the result.

Output:

HelloWorldThisisastringwithpunctuation

String Performance Considerations - Question 10

Develop a program that illustrates the use of interning to improve performance in an application with many identical strings.

String Performance Considerations - Question 10

Code:

```
import sys

def main():
    # Create a list of identical strings
    identical_strings = ["hello" * 1000] * 10000

    # Calculate memory usage before interning
    before_memory_usage = sys.getsizeof(identical_strings)

    # Intern the strings
    identical_strings = [sys.intern(string) for string in
identical_strings]

    # Calculate memory usage after interning
    after_memory_usage = sys.getsizeof(identical_strings)

                print(f"Memory Usage Before Interning:
{before_memory_usage} bytes")
    print(f"Memory Usage After Interning: {after_memory_usage}
bytes")

if __name__ == "__main__":
    main()
```

Explanation:

This program illustrates the use of string interning to improve performance in an application with many identical strings. It creates a list of identical strings, interns them using `sys.intern()`, and compares memory usage before and after interning.

Output:

```
Memory Usage Before Interning: 80056 bytes Memory Usage After
Interning: 85176 bytes
```

String Mutability Emulation - Question 1

Write a program that uses a

String Mutability Emulation - Question 1

Code:

```
def mutable_string_example():
    original_string = "Hello, World!"
    byte_array = bytearray(original_string, 'utf-8')

    # Modify the string in-place
    byte_array[0] = ord('J')  # Change 'H' to 'J'
    byte_array[7] = ord('w')  # Change 'W' to 'w'

    # Convert the bytearray back to a string
    modified_string = byte_array.decode('utf-8')

    # Print the original and modified strings
    print("Original String:", original_string)
    print("Modified String:", modified_string)

if __name__ == "__main__":
    mutable_string_example()
```

Explanation:

This program demonstrates the mutability of strings using a `bytearray`. It converts a string into a bytearray, modifies the bytearray in-place, and then converts it back to a string.

Output:

Original String: Hello, World! Modified String: Jello, world!

String Mutability Emulation - Question 2

Create a script that reads a text file, emulates string mutability using

String Mutability Emulation - Question 2

Code:

```
def emulate_string_mutability(filename, word_to_replace,
replacement):
    # Read the contents of the file
    with open(filename, 'r') as file:
        text = file.read()

    # Convert the text to a bytearray for mutability
    byte_array = bytearray(text, 'utf-8')

    # Replace specific words
    byte_array.replace(word_to_replace.encode('utf-8'),
                       replacement.encode('utf-8'))

    # Convert the bytearray back to a string
    modified_text = byte_array.decode('utf-8')

    # Write the modified text back to the file
    with open(filename, 'w') as file:
        file.write(modified_text)

if __name__ == "__main__":
    filename = "sample.txt"
    word_to_replace = "old"
    replacement = "new"
    emulate_string_mutability(filename, word_to_replace,
                               replacement)
```

Explanation:

This script reads a text file, emulates string mutability using a `bytearray`, and replaces specific words in the file. It demonstrates how to modify a text file in-place.

Output:

Before file text: old Line 1 old Line 2 old Line 3 After file
text: new Line 1 new Line 2 new Line 3

String Mutability Emulation - Question 3

Develop a function that emulates string mutability by using

String Mutability Emulation - Question 3

Code:

```
import io

def emulate_string_mutability_append(input_string,
appended_text):
    # Create a StringIO object
    string_io = io.StringIO(input_string)

    # Append text to the StringIO object
    string_io.write(appended_text)

    # Get the resulting string
    modified_string = string_io.getvalue()

    return modified_string

if __name__ == "__main__":
    input_string = "Hello, "
    appended_text = "World!"
    modified_string = emulate_string_mutability_append(input_string, appended_text)
    print("Modified String:", modified_string)
```

Explanation:

This function emulates string mutability by using `io.StringIO` to append characters to a string. It creates a StringIO object, appends text to it, and retrieves the modified string.

Output:

Modified String: World!

String Mutability Emulation - Question 4

Write a program that performs in-place character replacement in a string by emulating mutability with

String Mutability Emulation - Question 4

Code:

```
def in_place_character_replace(input_string, target_char, replacement_char):
    # Convert the string to a bytearray for mutability
    byte_array = bytearray(input_string, 'utf-8')

    for i, char in enumerate(byte_array):
        if char == ord(target_char):
            byte_array[i] = ord(replacement_char)

    # Convert the bytearray back to a string
    modified_string = byte_array.decode('utf-8')

    return modified_string

if __name__ == "__main__":
    input_string = "Hello, World!"
    target_char = 'o'
    replacement_char = 'x'
    modified_string = in_place_character_replace(input_string,
                                                target_char, replacement_char)
    print("Modified String:", modified_string)
```

Explanation:

This program performs in-place character replacement in a string by emulating mutability with a `bytearray`. It converts the string into a `bytearray`, iterates over its characters, and replaces the target character with the replacement character.

Output:

Modified String: Hellx, Wxrld!

String Mutability Emulation - Question 5

Create a script that emulates string mutability by using

String Mutability Emulation - Question 5

Code:

```
import io

def emulate_string_mutability_insert(input_string,
                                     insertion_text, position):
    # Create a StringIO object
    string_io = io.StringIO()

    # Write the original text up to the insertion position
    string_io.write(input_string[:position])

    # Insert the new text
    string_io.write(insertion_text)

    # Write the remaining part of the original text
    string_io.write(input_string[position:])

    # Get the resulting string
    modified_string = string_io.getvalue()

    return modified_string

if __name__ == "__main__":
    input_string = "Hello, Python!"
    insertion_text = " World"
    position = 5
    modified_string = emulate_string_mutability_insert(input_string, insertion_text, position)
    print("Modified String:", modified_string)
```

Explanation:

This script emulates string mutability by using `io.StringIO` to insert text at a specific position in a string. It creates a `StringIO` object, reads the original text up to the insertion position, inserts the text, and combines the parts to get the modified string.

Output:

Modified String: Hello World, Python!

String Mutability Emulation - Question 6

Develop a function that emulates string mutability with

String Mutability Emulation - Question 6

Code:

```
def reverse_string_in_place(input_string):
    # Convert the string to a bytearray for mutability
    byte_array = bytearray(input_string, 'utf-8')

    # Reverse the bytearray in-place
    byte_array.reverse()

    # Convert the bytearray back to a string
    reversed_string = byte_array.decode('utf-8')

    return reversed_string

if __name__ == "__main__":
    input_string = "Hello, World!"
    reversed_string = reverse_string_in_place(input_string)
    print("Reversed String:", reversed_string)
```

Explanation:

This function emulates string mutability with a `bytearray` and reverses a string in-place. It converts the string into a `bytearray`, reverses it in-place, and then converts it back to a string.

Output:

Reversed String: !dlroW ,olleH

String Mutability Emulation - Question 7

Write a program that uses

String Mutability Emulation - Question 7

Code:

```
def count_and_replace_characters(input_string, target_char, replacement_char):
    # Convert the string to a bytearray for mutability
    byte_array = bytearray(input_string, 'utf-8')

    # Initialize a count for the target character
    count = 0

        # Iterate through the bytearray to count and replace
        characters
        for i, char in enumerate(byte_array):
            if char == ord(target_char):
                count += 1
                byte_array[i] = ord(replacement_char)

    # Convert the bytearray back to a string
    modified_string = byte_array.decode('utf-8')

    return count, modified_string

if __name__ == "__main__":
    input_string = "Hello, World!"
    target_char = 'o'
    replacement_char = 'x'
                                count,      modified_string      =
count_and_replace_characters(input_string,          target_char,
replacement_char)
    print("Count of '{}' replaced: {}".format(target_char, count))
    print("Modified String:", modified_string)
```

Explanation:

This program uses a `bytearray` to efficiently count and replace all occurrences of a specific character in a string. It converts the string into a `bytearray`, iterates through it to

count and replace characters, and then converts it back to a string.

Output:

Count of 'o' replaced: 2 Modified String: Hellx, Wxrld!

String Mutability Emulation - Question 8

Create a script that emulates string mutability by using

String Mutability Emulation - Question 8

Code:

```
import io

def emulate_string_mutability_delete(input_string, start, end):
    # Create a StringIO object
    string_io = io.StringIO(input_string)

    # Read the original text up to the start position
    original_text = string_io.read(start)

    # Skip the characters to be deleted
    string_io.seek(end)

    # Read the remaining text
    remaining_text = string_io.read()

    # Combine the parts to get the modified string
    modified_string = original_text + remaining_text

    return modified_string

if __name__ == "__main__":
    input_string = "Hello, Python!"
    start = 5
    end = 11
    modified_string = emulate_string_mutability_delete(input_string, start, end)
    print("Modified String:", modified_string)
```

Explanation:

This script emulates string mutability by using `io.StringIO` to delete a range of characters from a string. It creates a `StringIO` object, reads the original text up to the start position, skips the characters to be deleted, reads the remaining text, and combines the parts to get the modified string.

Output:

Modified String: Helloon!

String Mutability Emulation - Question 9

Develop a function that emulates string mutability with

String Mutability Emulation - Question 9

Code:

```
def text_editor(input_string, operations):
    # Convert the string to a bytearray for mutability
    byte_array = bytearray(input_string, 'utf-8')

    for op in operations:
        if op[0] == 'insert':
            index, text_to_insert = op[1], op[2]
            byte_array[index:index] =
text_to_insert.encode('utf-8')
        elif op[0] == 'delete':
            start, end = op[1], op[2]
            del byte_array[start:end]

    # Convert the bytearray back to a string
    modified_string = byte_array.decode('utf-8')

    return modified_string

if __name__ == "__main__":
    input_string = "Hello, World!"
    operations = [('insert', 5, ' Beautiful'), ('delete', 0,
6)]
    modified_string = text_editor(input_string, operations)
    print("Modified String:", modified_string)
```

Explanation:

This function emulates string mutability with a `bytearray` and implements a simple text editor. It accepts a list of operations, including insertions and deletions, and applies them to the string using `bytearray` for mutability.

Output:

Modified String: Beautiful, World!

String Mutability Emulation - Question 10

Write a program that emulates string mutability using

String Mutability Emulation - Question 10

Code:

```
def emulate_string_mutability_replace(input_string,
old_substring, new_substring):
    # Convert the string to a bytearray for mutability
    byte_array = bytearray(input_string, 'utf-8')

    # Find and replace all occurrences of the old substring
    index = 0
    while index < len(byte_array):
        index = byte_array.find(old_substring.encode('utf-8')), index)
        if index == -1:
            break
        # Delete the old substring
        del byte_array[index:index + len(old_substring)]
        # Insert the new substring
        byte_array[index:index] = new_substring.encode('utf-8')
        index += len(new_substring)

    # Convert the bytearray back to a string
    modified_string = byte_array.decode('utf-8')

    return modified_string

if __name__ == "__main__":
    input_string = "Hello, Hello, Hello!"
    old_substring = "Hello"
    new_substring = "Hi"
    modified_string = emulate_string_mutability_replace(input_string, old_substring, new_substring)
    print("Modified String:", modified_string)
```

Explanation:

This program emulates string mutability using a `bytearray` and performs efficient substring replacement in a string. It finds

and replaces all occurrences of the old substring with the new substring using `bytarray` for mutability.

Output:

Modified String: Hi, Hi, Hi!

Cross-Language String Handling - Question 1

Write a program that calls a C/C++ function that returns a string, and then use the returned string in Python.

Cross-Language String Handling - Question 1

Code:

```
import ctypes
import os

#C++           installed          with          the
following:  https://www.freecodecamp.org/news/how-to-install-c-
and-cpp-compiler-on-windows/

# Full path to the Python DLL
dll_path = r'C:\msys64\mingw64\bin\libpython3.11.dll'

try:
    # Check if the DLL exists
    if not os.path.exists(dll_path):
        raise FileNotFoundError(f"Could not find {dll_path}")

    # Load the Python DLL
    python_lib = ctypes.CDLL(dll_path)

    # Define the return type of Py_GetVersion
    python_lib.Py_GetVersion.restype = ctypes.c_char_p

    # Call the function
    version_string = python_lib.Py_GetVersion()

    # Check if the function call returned a result
    if not version_string:
        raise ValueError("Function call to Py_GetVersion
returned an empty response")

    # Convert the C string to a Python string
    python_version = version_string.decode('utf-8')

    print("Python Version:", python_version)

except FileNotFoundError as e:
    print(f"Error: {e}")
except ValueError as e:
    print(f"Error: {e}")
```

```
except Exception as e:  
    print(f"An unexpected error occurred: {e}")
```

Explanation:

This program calls a C function from a shared library that returns a C string (`char*`). It uses `ctypes` to load the shared library, define the function signature, call the C function, and then converts the C string to a Python string for further use.

Output:

```
Python Version: 3.11.7 (main, Dec 7 2023, 09:07:50) [GCC 13.2.0  
64 bit (AMD64)]
```

Cross-Language String Handling - Question 2

Create a script that communicates with a Java program and exchanges string data between the two languages.

Cross-Language String Handling - Question 2

Code:

```
import subprocess

# Run the Java program that communicates with Python
java_process = subprocess.Popen(['java', 'JavaProgram'],
                                stdout=subprocess.PIPE, stdin=subprocess.PIPE, text=True)

# Send data to the Java program
java_process.stdin.write("Hello from Python!\n")
java_process.stdin.flush()

# Read the response from Java
response = java_process.stdout.readline()

# Close the Java process
java_process.stdin.close()
java_process.wait()

print("Response from Java:", response)

#####
##### install Java and compile the following code with "javac JavaProgram.java"
##### place the following code and compile it in the same directory
##### you are running python code from
#####
##### Java Code:
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class JavaProgram {
    public static void main(String[] args) {
        try (BufferedReader reader = new BufferedReader(new InputStreamReader(System.in))) {
```

```
// Read a line of text from the standard input
String line = reader.readLine();

// Process the input and generate a response
String response = "Received from Python: " + line;

// Send the response back to Python
System.out.println(response);
} catch (IOException e) {
    e.printStackTrace();
}
}
```

Explanation:

This script communicates with a Java program using the `subprocess` module. It runs the Java program as a subprocess, sends data to it, reads the response, and closes the Java process.

Output:

Response from Java: Received from Python: Hello from Python!

Cross-Language String Handling - Question 3

Develop a function that uses the `ctypes` library to call a C function that manipulates a Python string.

Cross-Language String Handling - Question 3

Code:

```
#C++           installed           with           the
following:  https://www.freecodecamp.org/news/how-to-install-c-
and-cpp-compiler-on-windows/

#DLL code:
#include
#include
#include

extern "C" __declspec(dllexport) void toUpperCase(char* str) {
    if(str != nullptr) {
        size_t len = std::strlen(str);
        for(size_t i = 0; i < len; i++) {
            str[i] = std::toupper(str[i]);
        }
    }
}

# Compiled at command line with: g++ -shared -o
string_manipulation.dll string_manipulation.cpp

# Python function that calls a C function to manipulate a
Python string

import ctypes

# Load the DLL
mylib = ctypes.CDLL('./string_manipulation.dll')

# Define the argument and return types
mylib.toUpperCase.argtypes = [ctypes.c_char_p]
mylib.toUpperCase.restype = None

def manipulate_string(input_string):
    buffer =
        ctypes.create_string_buffer(input_string.encode('utf-8'))
    mylib.toUpperCase(buffer)
```

```
    return buffer.value.decode('utf-8')

if __name__ == "__main__":
    input_string = "Hello, Python!"
    modified_string = manipulate_string(input_string)
    print("Modified String:", modified_string)
```

Explanation:

This function uses the `ctypes` library to call a C function that manipulates a Python string. It converts the input string to a C string, calls the C function, and then converts the modified C string back to a Python string.

Output:

Modified String: HELLO, PYTHON!

Cross-Language String Handling - Question 4

Write a program that communicates with a Rust application and exchanges string data via a foreign function interface (FFI).

Cross-Language String Handling - Question 4

Code:

```
# installing RUST -> https://www.rust-lang.org/tools/install

# Step 1: Write Your Rust Code
# =====
# In Rust, write a function to be exposed to Python using the C
calling convention.
# Example Rust function:

// string_processing.rs

use libc::{c_char, malloc};
use std::ptr;

#[no_mangle]
pub extern "C" fn process_string(input: *const c_char) -> *mut
c_char {
    unsafe {
        // Get the length of the input string
        let len = libc::strlen(input);

        // Allocate memory for the new string
        let buffer = malloc(len + 1) as *mut c_char;
        if buffer.is_null() {
            return ptr::null_mut();
        }

        // Copy the input string into the new buffer
        ptr::copy_nonoverlapping(input, buffer, len);

        // Null-terminate the new string
        *buffer.add(len) = 0;

        buffer
    }
}

# Step 2: Add Dependencies
# =====
```

```
# In your Rust project's Cargo.toml, include necessary dependencies, like libc.
[dependencies]
libc = "0.2"

# Step 3: Configure Cargo for Building a Shared Library
# =====
Edit the Cargo.toml file to specify crate-type as "cdylib" in the [lib] section.
[lib]
name = "my_rust_lib"
crate-type = ["cdylib"]

# Step 4: Create a lib.rs file
# =====
#Inside lib.rs, you need to declare any modules (Rust files) that are part of your library.

// lib.rs
mod string_processing;
// other code or module declarations can go here

# Step 4: Build the Shared Library
# =====
# In the Rust project directory, run `cargo build --release` to compile the code into a shared library.
# The output will be in `target/release`, named `libmyrust.so` on Linux.

# Step 5: Using the Library in Python
# =====
# Place the generated `libmyrust.so` in a location accessible to your Python script.
# In Python, use ctypes to load and interact with this shared library.

# Step 6: Memory Management
# =====
# Ensure proper memory allocation and deallocation in your Rust code.
# If the Rust function allocates memory, consider providing a Rust function to free that memory.
# This is crucial to avoid memory leaks when interfacing between Rust and Python.
```

```

# Python program that communicates with Rust via FFI

import ctypes

# Load the Rust shared library
rust_lib = ctypes.CDLL(r'C:\Users\User\my_rust_project\target\release\my_rust_lib.dll')

# Define the Rust function signature
rust_lib.process_string.argtypes = [ctypes.c_char_p]
rust_lib.process_string.restype = ctypes.c_char_p

def communicate_with_rust(input_string):
    # Convert the input string to a C string
    c_input_string = ctypes.c_char_p(input_string.encode('utf-8'))

    # Call the Rust function to process the string
    processed_c_string = rust_lib.process_string(c_input_string)

    # Convert the processed C string back to a Python string
    processed_string = processed_c_string.decode('utf-8')

    return processed_string

if __name__ == "__main__":
    input_string = "Hello from Python!"
    result = communicate_with_rust(input_string)
    print("Result from Rust:", result)

```

Explanation:

This program communicates with a Rust application using FFI (Foreign Function Interface). It loads the Rust shared library, defines the Rust function signature, and calls the Rust function to process a string, exchanging string data between Python and Rust.

Output:

Result from Rust: Hello from Python!

Cross-Language String Handling - Question 5

Create a script that interoperates with a C# library and passes strings back and forth between Python and C#.

Cross-Language String Handling - Question 5

Code:

```
#Compile the following code in a C# .Net Environment like VS
Code, Visual Studio, or C# compiler. Create the library file
"MyCSharpLibrary.dll"
# and copy into the same folder as the python code below:

namespace MyCSharpLibrary
{
    public class MyCSharpClass
    {
        public string GetHelloString()
        {
            return "Hello from C#!";
        }

        public void PrintString(string message)
        {
            Console.WriteLine(message);
        }
    }
}

#####
#####
# Python script that inter-operates with a C# library

import clr
clr.AddReference("MyCSharpLibrary") # Reference the C# library

from MyCSharpLibrary import MyCSharpClass # Import the C#
class

# Create an instance of the C# class
my_csharp_instance = MyCSharpClass()

# Call a C# method that returns a string
csharp_string = my_csharp_instance.GetHelloString()

# Print the C# string from Python
```

```
print("C# String:", csharp_string)

# Pass a Python string to a C# method
python_string = "Hello from Python!"
my_csharp_instance.PrintString(python_string)
```

Explanation:

This script demonstrates interoperability with a C# library using the `clr` module. It references the C# library, imports a C# class, and demonstrates calling a C# method that returns a string and passing a Python string to a C# method.

Output:

```
C# String: Hello from C#! Hello from Python!
```

Cross-Language String Handling - Question 6

Develop a program that uses JNI (Java Native Interface) to call a Java method that works with Python strings.

Cross-Language String Handling - Question 6

Code:

```
# Instructions for Compiling and Packaging Java Class
#
# -----
# 1. Compile the Java Class:
#     - Save your Java class (MyJavaClass.java) in a directory.
#     - Open a terminal or command prompt in that directory.
#     - Compile the class using: javac MyJavaClass.java
#         This will generate MyJavaClass.class in the directory.

# 2. Create a JAR File:
#     - In the same directory, create a JAR file using:
#         jar cvf MyJavaLibrary.jar MyJavaClass.class
#         - This creates MyJavaLibrary.jar containing the compiled
#           class.

# 3. Use the JAR in Python:
#     - Place MyJavaLibrary.jar in a location accessible to your
#       Python script.
#     - Reference the JAR file in your Python code using
#       jppye.startJVM().

# Python Code to Use the Java Class
# -----
```

#Java Code:

```
package com.example;

public class MyJavaClass {

    // Method to be called from Python to get a string
    public String getStringFromJava() {
        return "String from Java";
    }

    // Method to be called from Python with a string parameter
    public void printStringFromPython(String str) {
        System.out.println("Received in Java: " + str);
    }
}
```

```

        }
    }

#####
##### Python program that uses JNI to call a Java method

import jpy

# Start the JVM
jpy.startJVM(r"C:/Program Files/Java/jdk-21/bin/server/jvm.dll", classpath=['./MyJavaLibrary.jar'])

# Import the Java class
MyJavaClass = jpy.JClass('MyJavaClass')

# Create an instance of the Java class
java_instance = MyJavaClass()

# Call a Java method that returns a string
java_string = java_instance.getStringFromJava()

# Print the Java string from Python
print("Java String:", java_string)

# Pass a Python string to a Java method
python_string = "Hello from Python!"
java_instance.printStringFromPython(python_string)

# Shut down the JVM
jpy.shutdownJVM()

```

Explanation:

This program uses JPy to interact with Java using JNI (Java Native Interface). It starts the JVM, imports a Java class, creates an instance of the Java class, calls a Java method that returns a string, passes a Python string to a Java method, and shuts down the JVM.

Output:

Java String: String from Java Received in Java: Hello from Python!

Cross-Language String Handling - Question 7

Write a function that communicates with a C++ DLL (Dynamic Link Library) and passes a Python string as an argument.

Cross-Language String Handling - Question 7

Code:

```
#C++           installed           with           the
following:  https://www.freecodecamp.org/news/how-to-install-c-
and-cpp-compiler-on-windows/
```

```
# Python function that communicates with a C++ DLL
```

```
"""
```

Steps to Compile a C++ DLL Using MSYS2 and MinGW-w64:

1. Open MSYS2 Terminal:

```
# Start the MSYS2 terminal on your Windows system.
```

2. Install MinGW-w64 (if not already installed):

```
# In the MSYS2 terminal, install MinGW-w64 using the package
manager if it's not already installed.
```

```
# Example command: pacman -S mingw-w64-x86_64-gcc
```

3. Navigate to Your Code's Directory:

```
# Use the 'cd' command to change directory to where your C++
source file is located.
```

```
# Example: cd path/to/your/code
```

4. Compile the C++ Code into a DLL:

```
# Use the g++ compiler to compile your C++ code into a DLL.
```

```
# Command format: g++ -shared -o [output_dll_name].dll
[your_cpp_file].cpp -Wl,--out-implib,[lib_name].a
# Example: g++ -shared -o mycpp.dll mycpp.cpp -Wl,--out-
implib,libmycpp.a
```

```
# This command creates a DLL named 'mycpp.dll' and an import
library 'libmycpp.a'.
```

5. Check for the DLL:

```
# Ensure 'mycpp.dll' is created in your current directory
after running the command.
```

6. Use the DLL in Python:

```
# Place the DLL in a location where your Python script can
```

```

access it, such as the same directory.

# Make sure your Python script is properly set up to load
and use functions from the DLL.

"""

// mycpp.cpp
#include
#include
#include // For strcpy

extern "C" {
    // Function that takes a C-style string and returns a new
    // dynamically allocated C-style string
    __declspec(dllexport) char* process_string(const char*
input) {
        std::string inputStr(input);
        std::string result = inputStr + " - Processed by C++";

        char* cstr = new char[result.length() + 1]; // Allocate
        memory for the result
        strcpy(cstr, result.c_str()); // Copy the result into
        the newly allocated memory

        return cstr; // Return the new C-style string
    }
}

```

```

#####
###  

import ctypes

# Load the C++ DLL - Update with your path
cpp_dll = ctypes.CDLL(r'C:\Users\User\OneDrive\Documents\PythonBook\mycpp
.dll')

# Define the C++ function signature
cpp_dll.process_string.argtypes = [ctypes.c_char_p]
cpp_dll.process_string.restype = ctypes.c_char_p

def communicate_with_cpp(input_string):
    # Convert the input string to a C string
```

```
c_input_string = ctypes.c_char_p(input_string.encode('utf-8'))  
  
# Call the C++ function and pass the C string  
result_c_string = cpp_dll.process_string(c_input_string)  
  
# Convert the result C string back to a Python string  
result_string = result_c_string.decode('utf-8')  
  
return result_string  
  
if __name__ == "__main__":  
    input_string = "Hello from Python!"  
    result = communicate_with_cpp(input_string)  
    print("Result from C++:", result)
```

Explanation:

This function communicates with a C++ DLL using `ctypes`. It loads the C++ DLL, defines the C++ function signature, converts a Python string to a C string, calls the C++ function, and converts the result C string back to a Python string.

Output:

Result from C++: Hello from Python! - Processed by C++

Cross-Language String Handling - Question 8

Create a script that integrates Python with PowerShell to perform advanced file system operations and handle string data seamlessly.

Cross-Language String Handling - Question 8

Code:

```
import subprocess

def run_powershell_command(command):
    # Execute the PowerShell command
    result = subprocess.run(["powershell", "-Command",
command], capture_output=True)
    return result.stdout.decode()

# Example PowerShell command to list all files in a directory
command = 'Get-ChildItem -Path "C:\Python311\Scripts" -Recurse
| Select-Object FullName'

# Run the command and print the output
output = run_powershell_command(command)
print(output)
```

Explanation:

This script demonstrates how to integrate Python with PowerShell to perform file system operations. It uses Python's subprocess module to run a PowerShell command that lists all files in a specified directory. The output of the PowerShell command is captured and decoded into a Python-readable format. This approach can be extended to other file system operations or any PowerShell tasks that involve string data.

Output:

```
FullName      -----  C:\Python311\Scripts\django-admin.exe
C:\Python311\Scripts\estimator_ckpt_converter.exe
C:\Python311\Scripts\f2py.exe      C:\Python311\Scripts\google-
oauthlib-tool.exe
C:\Python311\Scripts\import_pb_to_tensorboard.exe
C:\Python311\Scripts\markdown_py.exe
C:\Python311\Scripts\pip.exe      C:\Python311\Scripts\pip3.11.exe
C:\Python311\Scripts\pip3.exe    C:\Python311\Scripts\py.test.exe
C:\Python311\Scripts\pyrsa-decrypt.exe
C:\Python311\Scripts\pyrsa-encrypt.exe
C:\Python311\Scripts\pyrsa-keygen.exe
```

C:\Python311\Scripts\pyrsa-priv2pub.exe
C:\Python311\Scripts\pyrsa-sign.exe C:\Python311\Scripts\pyrsa-
verify.exe C:\Python311\Scripts\pytest.exe
C:\Python311\Scripts\saved_model_cli.exe
C:\Python311\Scripts\sqlformat.exe
C:\Python311\Scripts\tensorboard.exe
C:\Python311\Scripts\tflite_convert.exe
C:\Python311\Scripts\tf_upgrade_v2.exe
C:\Python311\Scripts\toco.exe
C:\Python311\Scripts\toco_from_protos.exe
C:\Python311\Scripts\wheel.exe

Cross-Language String Handling - Question 9

Write a Python program that sends a string to a Bash script. The Bash script should convert the string to uppercase and return it. How would you implement this in Python and Bash?

Cross-Language String Handling - Question 9

Code:

```
# Python Code
import subprocess

def send_string_to_bash(input_string):
    result = subprocess.run(['bash', 'script.sh',
input_string], capture_output=True, text=True)
    return result.stdout.strip()

input_string = "hello world"
print("Original String:", input_string)
print("Modified String:", send_string_to_bash(input_string))

#####
#####

# Bash Script (script.sh)
#!/bin/bash
echo $1 | tr '[:lower:]' '[:upper:]'
```

Explanation:

Explanation not provided

Output:

Original String: hello world Modified String: HELLO WORLD

Cross-Language String Handling - Question 10

Write a script that demonstrates interoperability with a Go program and transfers strings between Python and Go code.

Cross-Language String Handling - Question 10

Code:

```
# Python script that interoperates with a Go program

import subprocess

def run_go_program_with_argument(arg=None):
    command = ['go', 'run', 'mygocode.go']
    if arg:
        command.append(arg)

        go_process      = subprocess.Popen(command,
stdout=subprocess.PIPE, stderr=subprocess.PIPE)
        go_output, go_errors = go_process.communicate()

    if go_errors:
        print("Error running Go program:", go_errors.decode())
    else:
        return go_output.decode().strip()

def main():
    # Run the Go program and print its initial output
    output = run_go_program_with_argument()
    print("Output from Go program:", output)

    # Send a string to the Go program and print its response
    response = run_go_program_with_argument("Hello from
Python!")
    print("Response from Go program:", response)

if __name__ == "__main__":
    main()
```

Explanation:

This Python script interoperates with a Go program. It runs the Go program as a subprocess, reads and decodes the program's

output, and demonstrates passing a Python string to the Go program and receiving its response.

Output:

```
Output from Go program: Hello from Go! Response from Go
program: Hello from Go! Received 'Hello from Python!' from
Python
```

Advanced Regular Expressions - Question 1

Write a program that uses a look-ahead assertion in a regex pattern to find words that are followed by a specific word.

Advanced Regular Expressions - Question 1

Code:

```
import re

text = "The quick brown fox jumps over the lazy dog and the
quick fox."
pattern = r"\b\w+(?=quick\b)"

matches = re.findall(pattern, text)
print(matches)
```

Explanation:

This program uses a regex pattern with a look-ahead assertion `(?=quick\b)` to find words that are followed by the word "quick" (but do not include "quick" in the match).

Output:

```
['The', 'the']
```

Advanced Regular Expressions - Question 2

Create a script that extracts email addresses from a text using a regex pattern with a look-behind assertion.

Advanced Regular Expressions - Question 2

Code:

```
import re

text = "Contact us at email1@example.com and email2@example.org
for assistance."

pattern = r"(?<=\s)\S+@\S+"

matches = re.findall(pattern, text)
print(matches)
```

Explanation:

This script uses a regex pattern with a look-behind assertion to extract email addresses from the text.

Output:

```
['email1@example.com', 'email2@example.org']
```

Advanced Regular Expressions - Question 3

Develop a function that uses a non-capturing group in a regex pattern to find and count occurrences of a pattern.

Advanced Regular Expressions - Question 3

Code:

```
import re

def count_occurrences(text, pattern):
    matches = re.findall(pattern, text)
    return len(matches)

text = "The quick brown fox jumps over the quick dog, and the
fox is quick."

pattern = r"(?:\bquick\b)"

count = count_occurrences(text, pattern)
print("Occurrences:", count)
```

Explanation:

This function uses a regex pattern with a non-capturing group `(?:\bquick\b)` to find and count occurrences of the word "quick" in the text.

Output:

Occurrences: 3

Advanced Regular Expressions - Question 4

Write a program that extracts all URL links from an HTML document using a regex pattern with named capturing groups.

Advanced Regular Expressions - Question 4

Code:

```
import re

html_text = 'Example Sample'

pattern = r'(?P.*?)'

matches = re.finditer(pattern, html_text)

for match in matches:
    url = match.group('url')
    text = match.group('text')
    print(f"URL: {url}, Text: {text}")
```

Explanation:

This program uses a regex pattern with named capturing groups to extract URL links and their associated text from an HTML document.

Output:

```
URL: https://example.com, Text: Example URL:
https://sample.org, Text: Sample
```

Advanced Regular Expressions - Question 5

Create a script that parses CSV data with irregularly quoted fields using a regex pattern and non-greedy quantifiers.

Advanced Regular Expressions - Question 5

Code:

```
import re

csv_text = 'John,Doe,"123 Main St, Apt 45",555-1234\nAlice,Smith,42 Elm St,555-5678'

pattern = r'"(^[^"]*?)"|([^\"]+)'

matches = re.findall(pattern, csv_text)
rows = []

for match in matches:
    rows.append(''.join(match).strip() if match[0] else
match[1])

print(rows)
```

Explanation:

This script parses CSV data with irregularly quoted fields using a regex pattern with non-greedy quantifiers. It handles both quoted and unquoted fields.

Output:

```
['John', 'Doe', '123 Main St, Apt 45', '555-1234\nAlice',
'Smith', '42 Elm St', '555-5678']
```

Advanced Regular Expressions - Question 6

Develop a function that validates complex passwords using a regex pattern with multiple assertions (length, character types, etc.).

Advanced Regular Expressions - Question 6

Code:

```
import re

def validate_password(password):
    # Password must have at least 8 characters, including at
    # least one uppercase letter, one lowercase letter,
    # one digit, and one special character (!@#$%^&*)
    pattern = r"^(?=.*[A-Z])(?=.*[a-z])(?=.*\d)(?=.*[!@#$%^&*]).{8,}$"
    return bool(re.match(pattern, password))

# Test the function
password1 = "P@ssw0rd"
password2 = "weak"
print("Password 1:", validate_password(password1)) # True
print("Password 2:", validate_password(password2)) # False
```

Explanation:

This function validates complex passwords using a regex pattern with multiple assertions, including length, character types (uppercase, lowercase, digit, special character), and a minimum length requirement of 8 characters.

Output:

Password 1: True Password 2: False

Advanced Regular Expressions - Question 7

Write a program that identifies and replaces HTML tags in a text with a specified replacement using a regex pattern.

Advanced Regular Expressions - Question 7

Code:

```
import re

html_text = '
This is important information.

'

# Define a replacement function
def replace_tags(match):
    tag = match.group(0)
    return f"[{tag}]"

pattern = r'<[^>]+>'

# Replace HTML tags with brackets
result = re.sub(pattern, replace_tags, html_text)
print(result)
```

Explanation:

This program uses a regex pattern to identify and replace HTML tags in a text with a specified replacement (in this case, square brackets).

Output:

```
[

]This is []important[] information.[

]
```

Advanced Regular Expressions - Question 8

Create a regex pattern to identify valid email addresses. The pattern should ensure the following criteria are met:

1. The email must start with a letter or number.
2. It should include an '@' symbol followed by a domain name (e.g., example.com).
3. The domain name must:
 - Start with a letter.
 - Only contain letters, numbers, or hyphens.
 - End with a top-level domain (TLD) like .com, .net, .org, etc., which must be at least two characters long.

Advanced Regular Expressions - Question 8

Code:

```
import re

def is_valid_email(email):
    pattern = r'^[A-Za-z0-9][A-Za-z0-9._%+-]*@[A-Za-z0-9][A-Za-z0-9-]*\.[A-Za-z]{2,}\$'
    return bool(re.match(pattern, email))

# Test the function
emails = ["john.doe@example.com", "john.doe@-example.com",
          "john.doe@example", "Jane-123@abc.net"]
for email in emails:
    print(f"{email}: {is_valid_email(email)})
```

Explanation:

This script uses a regex pattern to validate email addresses. The pattern ensures that the email starts with a letter or number, includes an '@' symbol followed by a valid domain name, and ends with a proper top-level domain.

Output:

```
john.doe@example.com: True john.doe@-example.com: False
john.doe@example: False Jane-123@abc.net: True
```

Advanced Regular Expressions - Question 9

Develop a function that extracts phone numbers from text, considering various formats and separators using a regex pattern.

Advanced Regular Expressions - Question 9

Code:

```
import re

def extract_phone_numbers(text):
    pattern = r'(\+?[\d() -]+)'
    matches = re.findall(pattern, text)
    return [match.strip('() -') for match in matches]

text = "Contact us at +1 (555) 123-4567 or 555-7890 (office)."

phone_numbers = extract_phone_numbers(text)
print(phone_numbers)
```

Explanation:

This function extracts phone numbers from text, considering various formats and separators. It uses a regex pattern to capture phone number patterns.

Output:

```
['', '', '', '+1 (555) 123-4567', '555-7890', '']
```

Advanced Regular Expressions - Question 10

Write a program that uses a positive look-behind assertion to find and highlight specific words in a text without capturing them.

Advanced Regular Expressions - Question 10

Code:

```
import re

text = "The quick brown fox jumps over the lazy dog."

pattern = r'(?=<\bquick\b\s)(\w+)'

# Replace 'quick' with '[quick]'
result = re.sub(pattern, r'[\1]', text)
print(result)
```

Explanation:

This program uses a positive look-behind assertion to find and highlight words that follow the word "quick" without capturing the word "quick" itself.

Output:

The quick [brown] fox jumps over the lazy dog.