Q1. Does assigning a value to a string's indexed character violate Python's string immutability?

Ans: Yes, assigning a value to a string's indexed character violates Python's string immutability.

In Python, strings are immutable objects, which means that once a string is created, its contents cannot be modified. If we try to assign a new value to a character within a string using indexing, will get a TypeError indicating that 'str' object does not support item assignment.

**Ex:**

st = 'hello'

st[0] = 'i'

print(st)

**O/P:**

TypeError: 'str' object does not support item assignment.

Q2. Does using the += operator to concatenate strings violate Python's string immutability? Why or why not?

Ans: Using the += operator to concatenate strings does not violate Python's string immutability.

When we use the += operator to concatenate strings, we are not modifying the original string object. Instead, we are creating a new string object that contains the concatenated result of the original string and the string are appending. This new string object is then assigned to the same variable name, effectively replacing the original string object.

For example, consider the following code snippet:

**Ex:**

st = 'hello'

st += 'world'

print(st)

**O/P:**

helloworld

Q3. In Python, how many different ways are there to index a character?

Ans: In Python, there is only one way to index a character in a string, and that is by using square brackets [] with the index value inside.

For example, consider the following code snippet:

st = 'hello'

first\_char = st[0]

print(first\_char)

In this code, the square brackets [] are used to access the character at index 0 in the string "hello", which is the letter "h". The resulting value is then assigned to the variable first\_char.

There is no other way to index a character in a string in Python. However, you can use different techniques to manipulate the string or extract substrings, such as slicing, splitting, or using regular expressions.

Q4. What is the relationship between indexing and slicing?

Ans: The relationship between indexing and slicing is that slicing is a more generalized form of indexing. When we use indexing with a single index value, we get back a single element at that position. However, when we use slicing with a range of index values, we get a new sequence object that contains a portion of the original sequence.

Ex:

st = 'hello'

first\_char = st[0]

sub\_str = st[2:4]

print(sub\_str)

Q5. What is an indexed character's exact data type? What is the data form of a slicing-generated substring?

Ans: An indexed character in a string is of type str, and a substring generated using slicing is also of type str. Both are string objects containing one or more characters.

**Ex:**

st = 'hello'

first\_char = st[0]

sub\_str = st[2:4]

print(type(first\_char))

print(type(sub\_str))

**O/P:**

<class 'str'>

<class 'str'>

Q6. What is the relationship between string and character "types" in Python?

Ans: In Python, a string is a sequence of characters. Each character in a string is represented by a Unicode code point, which is an integer value that uniquely identifies the character.

While strings and characters are not separate "types" in Python, they are related in that a string is made up of one or more characters. We can access individual characters within a string by indexing or iterate over them using a loop. Additionally, we can perform various operations on strings, such as concatenation, slicing, and searching, that involve manipulating one or more characters.

Q7. Identify at least two operators and one method that allow you to combine one or more smaller strings to create a larger string.

Ans: There are several ways to combine one or more smaller strings to create a larger string in Python. Here are two common operators and one method that you can use:

a. The + operator

b. The += operator

c. The join() method

Q8. What is the benefit of first checking the target string with in or not in before using the index method to find a substring?

Ans: By using in or not in to check for the presence of a substring before using index() to find it, you can avoid the ValueError and write more robust code that can handle the case where the substring is not present in the target string.

**Ex:**

my\_string = "hello world"

if "world" in my\_string:

index = my\_string.index("world")

print(f"Found 'world' at index {index}")

else:

print("'world' not found in string")

Q9. Which operators and built-in string methods produce simple Boolean (true/false) results?

Ans: Several operators and built-in string methods in Python produce simple Boolean (true/false) results.

a. Using in and not in operators

b. startswith() and endswith() methods

c. isalpha(), isdigit(), isalnum(), and isspace() methods