**ASSIGNMENT – 12**

1. **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 their value cannot be modified after they are created.

When you assign a value to a string's indexed character, you are attempting to modify the string's value, which is not allowed. Instead, Python raises a TypeError with the message " 'str' object does not support item assignment".

For example

s = "hello"

s[0] = "H"

This code will raise a TypeError because it is attempting to modify the value of the string `s` by assigning a new value to its first character.

If you want to modify a string in Python, you need to create a new string with the desired modifications. For example, you can use string slicing and concatenation to create a new string that is similar to the original one, but with some modifications:

s = "hello"

s = "H" + s[1:]

This code creates a new string with the same characters as `s`, except that the first character is replaced with `"H"`.

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

**Ans:**

Using the `+=` operator to concatenate strings in Python does not violate Python's string immutability. Although strings are immutable objects in Python, the `+=` operator works differently for strings than it does for mutable objects like lists.

When you use the `+=` operator to concatenate strings, Python creates a new string object that is the concatenation of the original string and the added string, and then assigns the new object to the same variable. This means that the original string object is not modified, and a new object is created instead.

For example, consider the following code:

s = "hello"

s += " world"

1. **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 of the character inside the brackets. For example, given a string `s`, you can access the character at index `i` using `s[i]`.

It's worth noting that strings in Python are immutable, which means that you cannot modify a single character in a string. If you need to modify a string, you need to create a new string with the desired modifications.

1. **What is the relationship between indexing and slicing?**

**Ans:**

In Python, indexing and slicing are both used to access specific elements of a sequence, such as a string or a list.

Indexing refers to accessing a single element of the sequence using its position or index. In Python, indexing starts at 0, so the first element of a sequence has an index of 0, the second element has an index of 1, and so on. You can access a specific element of a sequence using square brackets with the index inside, like this: `my\_sequence[index]`.

Slicing, on the other hand, refers to accessing a contiguous subset of elements of the sequence, specified by a range of indices. You can use slicing to extract a substring from a string or a sublist from a list, for example. Slicing in Python is done using the colon `:` operator inside the square brackets, like this: `my\_sequence[start:end]`. This will return a new sequence containing the elements from `start` up to (but not including) `end`.

1. **What is an indexed character’s exact data type? What is the data form of a slicing-generated substring?**

**Ans:**

In Python, an indexed character in a string is of type `str`. That means that it is a single-character string.

A slicing-generated substring is also of type `str`. It is a new string that is created by extracting a range of characters from the original string. The data form of the slicing-generated substring is still a string, but it may contain multiple characters. For example, if you slice a string `"hello"` to get the substring ell, the resulting substring is a new string of length 3, with the characters e, l, and l.

1. **What is the relationship between string and character ‘types’; in Python?**

Ans:

In Python, a string is a sequence of characters. The string type represents a sequence of Unicode characters, while the character type represents a single Unicode character.

In other words, a string is composed of one or more characters. Each character in a string is of the character type, which is a type of its own. Characters are represented as strings of length 1.

You can access individual characters in a string using indexing, as mentioned before, and the result of the indexing operation is a string of length 1 representing the character at the specified index.

Overall, the character "type" is a fundamental building block of the string type in Python, which is a sequence of zero or more characters.

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

**Ans:**

In Python, there are several operators and methods that allow you to combine one or more smaller strings to create a larger string:

1. The concatenation operator `+` can be used to join two or more strings together. For example, `"hello" + " " + "world"` would result in the string `"hello world"`.

2. The multiplication operator `\*` can be used to repeat a string a specified number of times. For example, `"spam" \* 3` would result in the string `"spamspamspam"`.

3. The `join()` method can be used to join a list of strings into a single string, using a specified delimiter. For example, `", ".join(["apple", "banana", "cherry"])` would result in the string `"apple, banana, cherry"`.

1. **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:

The `in` and `not in` operators can be used to check if a substring is present or absent in a larger string, respectively. The `index()` method can be used to find the index of the first occurrence of a substring in a larger string.

The benefit of first checking with `in` or `not in` before using the `index()` method is that it can prevent an error from occurring if the substring is not present in the larger string.

If you try to use the `index()` method on a substring that is not present in the larger string, a `ValueError` will be raised. By checking with `in` or `not in` first, you can avoid this error by only calling `index()` when you know that the substring is present in the larger string.

Using `in` or `not in` first also allows you to handle the case where you need to perform different actions depending on whether the substring is present or absent, without raising an error.

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

Ans:

In Python, there are several operators and built-in string methods that produce simple Boolean (true/false) results:

Operators:

1. `in`: This operator returns `True` if a substring is present in a larger string, and `False` otherwise. For example, `"hello" in "hello world"` would return `True`, while `"goodbye" in "hello world"` would return `False`.

2. `not in`: This operator returns `True` if a substring is not present in a larger string, and `False` otherwise. For example, `"goodbye" not in "hello world"` would return `True`, while `"hello" not in "hello world"` would return `False`.

Built-in String Methods:

1. `startswith()`: This method returns `True` if a string starts with a specified substring, and `False` otherwise. For example, `"hello world".startswith("hello")` would return `True`, while `"hello world".startswith("goodbye")` would return `False`.

2. `endswith()`: This method returns `True` if a string ends with a specified substring, and `False` otherwise. For example, `"hello world".endswith("world")` would return `True`, while `"hello world".endswith("goodbye")` would return `False`.

3. `isalnum()`: This method returns `True` if a string is alphanumeric (contains only letters and numbers), and `False` otherwise. For example, `"hello123".isalnum()` would return `True`, while `"hello 123".isalnum()` would return `False`.

4. `isdigit()`: This method returns `True` if a string contains only digits (0-9), and `False` otherwise. For example, `"123".isdigit()` would return `True`, while `"hello123".isdigit()` would return `False`.

5. `isalpha()`: This method returns `True` if a string contains only letters (A-Z or a-z), and `False` otherwise. For example, `"hello".isalpha()` would return `True`, while `"hello123".isalpha()` would return `False`.