Strings

Introduction

- A character is simply a symbol. For example, the English language has 26 characters; while a string is a contiguous sequence of characters.
- In Python, a string is a sequence of Unicode characters.
- Python strings are **immutable**, which means they cannot be altered after they are created.

Note: Unicode was introduced to include every character in all languages and bring uniformity in encoding. You can learn more about Unicode from Python Unicode. Given below are some examples of Unicode Encoding:

`string ='pythön!'` → Default **UTF-8** Encoding: `b'pyth\xc3\xb6n!'`

`string ='python!'`→ Default **UTF-8** Encoding: `b'python!'`

How to create a string in Python?

Strings can be created by enclosing a sequence of characters inside a single quote or double-quotes.

```
# Defining strings in Python
# Both of the following are equivalent
s1 = 'Hello'
print(s1)
s2 = "Hello"
print(s2)
```

Output:

```
Hello
Hello
```

Note:

We can use triple quotes to create docstrings and/or multiline strings.

```
s3 = '''Hello
        World'''
print(s3) #Multiline string
```

Output

```
Hello
World
```

If you wish to print a string in a new line you can use the new line character `'\n'`. This is used within the strings. For example:

```
print('First line\nSecond line')
```

This would yield the result:

```
First Line
Second Line
```

Accessing Characters in a String

String indices start at 0 and go on till 1 less than the length of the string. We can use the index operator **[ ]** to access a particular character in a string. Eg.

Index:	**0**	**1**	**2**	**3**	**4**
String `"hello"`:	`"h"`	`"e"`	`"l"`	`"l"`	`"o"`

Note: Trying to access indexes out of the range `(0,lengthOfString-1)`, will raise an **IndexError**. Also, the index must be an integer. We can't use float or other data types;this will result in **TypeError**.

Let us take an example to understand how to access characters in a String:

```
s= "hello"
>>> print(s[0]) #Output: h
>>> print(s[2]) #Output: l
>>> print(s[4]) #Output: o
```

Negative Indexing

Python allows negative indexing for strings. The index of **-1** refers to the last character, **-2** to the second last character, and so on. The negative indexing starts from the last character in the string.

Positive Indexing:	**0**	**1**	**2**	**3**	**4**
String `"hello"`:	`"h"`	`"e"`	`"l"`	`"l"`	`"o"`
Negative Indexing:	**-5**	**-4**	**-3**	**-2**	**-1**

Let us take an example to understand how to access characters using negative indexing in a string:

```
s= "hello"
>>> print(s[-1]) #Output: o
>>> print(s[-2]) #Output: l
>>> print(s[-3]) #Output: l
```

Concatenation Of Strings

Joining of two or more strings into a single string is called string concatenation. The **+** operator does this in Python.

```
# Python String Operations
str1 = 'Hello'
str2 ='World!'
# using +
print('str1 + str2 = ', str1 + str2)
```

When we run the above program, we get the following output:

```
str1 + str2 =  HelloWorld!
```

Simply writing two string literals together also concatenates them.

```
>>> # two string literals together
>>> 'Hello ''World!'
'Hello World!'
```

If we want to concatenate strings in different lines, we can use parentheses.

```
>>> # using parentheses
>>> s = ('Hello '
...      'World')
>>> s
'Hello World'
```

Note: You cannot combine numbers and strings using the **+** operator. If you do so, you will get an error:

```
>>> "Hello" + 11
TypeError: can only concatenate str (not "int") to str
```

Repeating/Replicating Strings

You can use ***** operator to replicate a string specified number of times. Consider the example given below:

```
>>> s= "hello"
>>> print(s*3) #s*3 will produce a new string with s repeated thrice
hellohellohello #Output
```

Note: You cannot use ***** operator between two strings i.e. you cannot multiply a string by another string.

String Slicing

String slicing refers to accessing a specific portion or a subset of a string with the original string remaining unaffected. You can use the indexes of string characters to create string slices as per the following syntax:

```
slice= <String Name>[StartIndex : StopIndex : Steps]
```

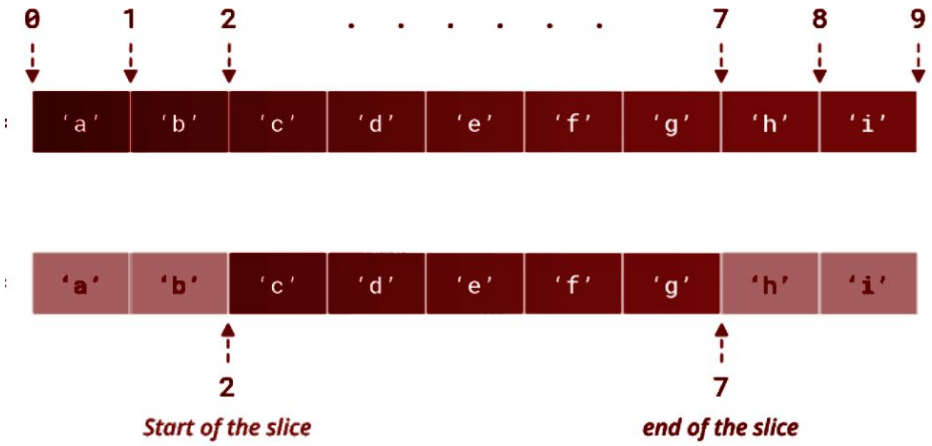
- The **`StartIndex`** represents the index from where the string slicing is supposed to begin. Its default value is 0, i.e. the string begins from index 0 if

no **`StartIndex`** is specified.

- The **`StopIndex`** represents the last index up to which the string slicing will go on. Its default value is **`(length(string)-1)`** or the index of the last character in the string.
- **`steps`** represent the number of steps. It is an optional parameter. **`steps,`** if defined, specifies the number of characters to jump over while counting from StartIndex to StopIndex. By default, it is 1.
- The string slices created, include characters falling between the indexes **`StartIndex`** and **`StopIndex`**, including **`StartIndex`** and not including **`StopIndex`**.

Here is a basic example of string slicing.

```
s = "abcdefghi"
print(s[2:7])
```



As you can see from the figure given above, we get the output as:

```
cdefg
```

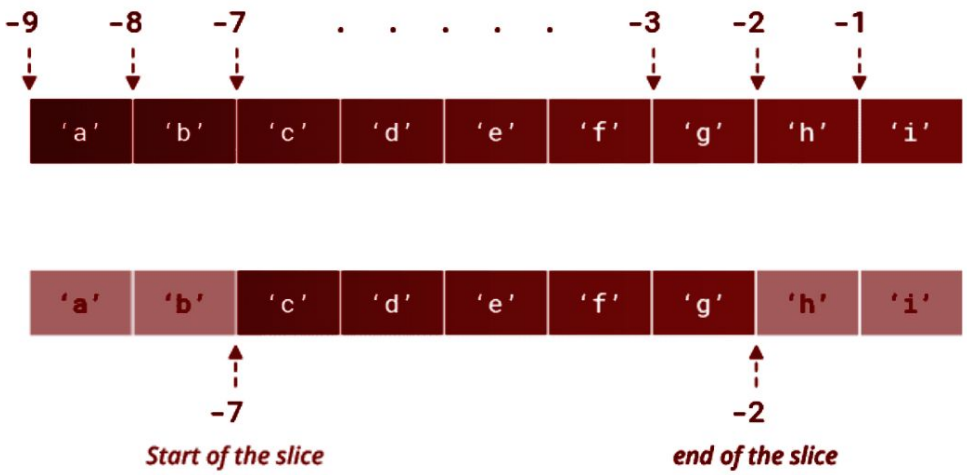
Slice Using Negative Indices

You can also specify negative indices while slicing a string. Consider the example given below.

```
s = "abcdefghi"
print(s[-7:-2])
```

Thus, we get the output as:

```
cdefg
```

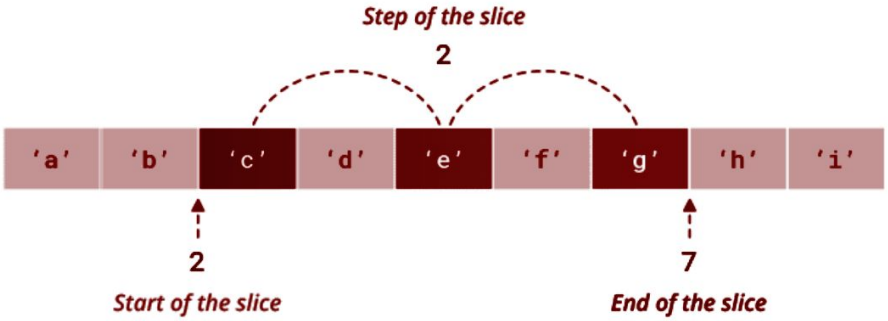


Specify Step of the Slicing

You can specify the step of the slicing using the `steps` parameter. The `steps` parameter is optional and by default 1.

```
# Every 2nd character between position 2 to 7
```

```
s = "abcdefghi"
print(s[2:7:2])
```



The output will be:

```
ceg
```

You can even specify a negative step size:

```
# Print every 2nd item between position 6 to 1
s = "abcdefghi"
print(s[6:1:-2])
```

The output will be:

```
gec
```

Slice at Beginning & End

Omitting the **`StartIndex`** starts the slice from the index 0. Meaning, **`S[:stop]`** is equivalent to **`S[0:stop]`**.

```
# Slice the first three items from the string
s = "abcdefghi"
print(s[:3])
```

Output

```
abc
```

Whereas, omitting the **`StopIndex`** extends the slice to the end of the string.

Meaning, **`S[start:]`** is equivalent to **`S[start:len(S)]`**.

```
# Slice the last three items from the string
s = "abcdefghi"
print(s[6:])
```

Output

```
ghi
```

Comparing Strings

1. In string comparison, we aim to identify whether two strings are equivalent to each other and if not, which one is greater.
2. String comparison in Python takes place character by character. That is, characters in the same positions are compared from both the strings.
3. If the characters fulfill the given comparison condition, it moves to the characters in the next position. Otherwise, it merely returns **False**.

Note: Some points to remember when using string comparison operators:

- The comparisons are **case-sensitive**, hence same letters in different letter cases(upper/lower) will be treated as separate characters.
- If two characters are different, then their Unicode value is compared; the character with the smaller Unicode value is considered to be lower.

This is done using the following operators:

- **`==`:** This checks whether two strings are equal
- **`!=`**: This checks if two strings are not equal

- `<`: This checks if the string on its left is smaller than that on its right
- `<=`: This checks if the string on its left is smaller than or equal to that on its right
- `>`: This checks if the string on its left is greater than that on its right
- `>=`: This checks if the string on its left is greater than or equal to that on its right

Iterating On Strings

There are multiple ways to iterate over a string in Python.

Using `for` loop

```
s="13579"
# Using for loop
for i in li:
    print(i) #Print the character in the string
```

Output:

```
1
3
5
7
9
```

Using `for` loop and `range()`

```
s="13579"
length = len(s) #Getting the length of the string
for i in range(length): #Iterations from 0 to (length-1)
    print(i)
```

Output:

```
1
3
5
7
9
```

*You can even use **while()** loops. Try using the **while()** loops on your own.*