FYBCA-SEM2-204 - Programming Skills

UNIT-3: Python Strings and Operators

3.1 Python Strings:

Python string is the collection of the characters surrounded by single quotes, double quotes, or triple quotes. The computer does not understand the characters; internally, it stores manipulated character as the combination of the 0's and 1's.Each character is encoded in the ASCII or Unicode character. So we can say that Python strings are also called the collection of Unicode characters.

In Python, strings can be created by enclosing the character or the sequence of characters in the quotes. Python allows us to use single quotes, double quotes, or triple quotes to create the string.

Consider the following example in Python to create a string.

```
str1 = "Hi Python !"
print(type(str1))
```

In Python, strings are treated as the sequence of characters, which means that Python doesn't support the character data-type; instead, a single character written as 'p' is treated as the string of length 1.

3.1.1 Multiline string, String as character array, triple quotes

Creating String in Python

We can create a string by enclosing the characters in single-quotes or double- quotes. Python also provides triple-quotes to represent the string, but it is generally used for multiline string or docstrings.

```
#Using single quotes
str1 = 'Hello Python'
print(str1)
#Using double quotes
str2 = "Hello Python"
print(str2)

#Using triple quotes
str3 = '''Triple quotes are generally used for
    represent the multiline or
    docstring'''
print(str3)
```

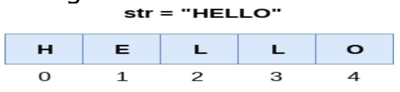
Output:

```
Hello Python
Hello Python
Triple quotes are generally used for
    represent the multiline or
    docstring
```

3.1.2 Slicing string, negative indexing, string length, concatenation

Strings indexing and Slicing (splitting)

Like other languages, the indexing of the Python strings starts from 0. For example, The string "HELLO" is indexed as given in the below figure.



str[0] = 'H'

str[1] = 'E'

str[2] = 'L'

str[3] = 'L'

str[4] = 'O'

Consider the following example:

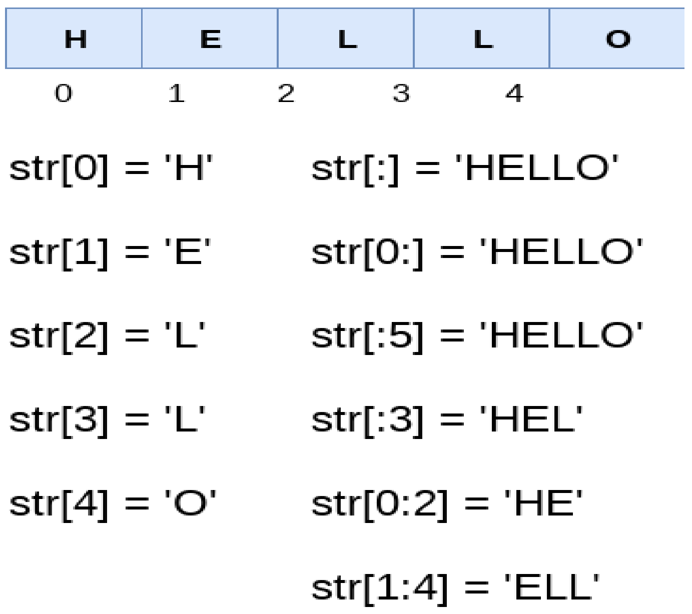
```
str = "HELLO"
print(str[0])
print(str[1])
print(str[2])
print(str[3])
print(str[4])
print(str[6])# It returns the IndexError because 6th index doesn't exist
```

Output:

```
H
E
L
L
O
IndexError: string index out of range
```

As shown in Python, the slice operator [] is used to access the individual characters of the string. However, we can use the **:** (colon) operator in Python to access the substring from the given string. Consider the following example.

str = "HELLO"



H	E	L	L	O
0	1	2	3	4

```
str[0] = 'H'      str[:] = 'HELLO'
str[1] = 'E'      str[0:] = 'HELLO'
str[2] = 'L'      str[:5] = 'HELLO'
str[3] = 'L'      str[:3] = 'HEL'
str[4] = 'O'      str[0:2] = 'HE'
                  str[1:4] = 'ELL'
```

Figure: String with Slice operator

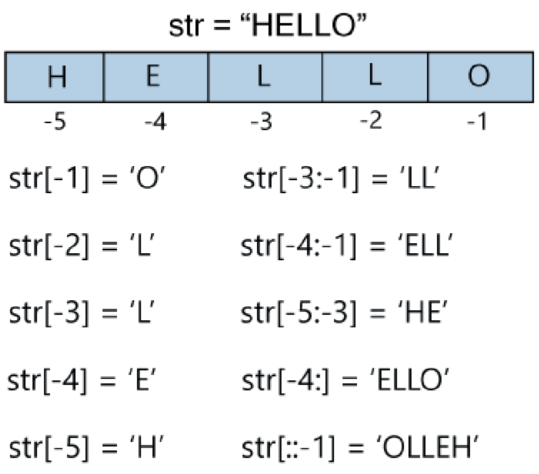
Here, we must notice that the upper range given in the slice operator is always exclusive i.e., if str = 'HELLO' is given, then str[1:3] will always include str[1] = 'E', str[2] = 'L' and nothing else.**Example:**

```
str = "JAVATPOINT"
# Start Oth index to end
print(str[0:])
# Starts 1th index to 4th index
print(str[1:5])
# Starts 2nd index to 3rd index
print(str[2:4])
# Starts 0th to 2nd index
print(str[:3])
#Starts 4th to 6th index
print(str[4:7])
```

Output:

```
JAVATPOINT
AVAT
VA
JAV
TPO
```

We can do the negative slicing in the string; it starts from the rightmost character, which is indicated as -1. The second rightmost index indicates -2, and so on. Consider the following image.



```
str[-1] = 'O'      str[-3:-1] = 'LL'
str[-2] = 'L'      str[-4:-1] = 'ELL'
str[-3] = 'L'      str[-5:-3] = 'HE'
str[-4] = 'E'      str[-4:] = 'ELLO'
str[-5] = 'H'      str[::-1] = 'OLLEH'
```

Figure: Negative slicing in the string

Consider the following example

```
str = 'JAVATPOINT'
print(str[-1])
print(str[-3])
print(str[-2:])
print(str[-4:-1])
print(str[-7:-2])
print(str[::-1])# Reversing the given string
print(str[-12])
```

Output:

```
T
I
NT
OIN
ATPOI
TNIOPTAVAJ
```

Reassigning Strings: Updating the content of the strings is as easy as assigning it to a new string. The string object doesn't support item assignment i.e., A string can only be replaced with new string since its content cannot be partially replaced. Strings are immutable in Python.
Consider the following example.

```
str = "HELLO"
str[0] = "h"
print(str)
```

Output:

```
Traceback (most recent call last):
  File "12.py", line 2, in <module>
    str[0] = "h";
TypeError: 'str' object does not support item assignment
```

However, in example 1, the string str can be assigned completely to a new content as specified in the following example.

```
str = "HELLO"
print(str)
str = "hello"
print(str)
```

OUTPUT:

```
HELLO
hello
```

Deleting the String

As we know that strings are immutable. We cannot delete or remove the characters from the string. But we can delete the entire string using the del keyword.

```
str = "JAVATPOINT"
del str[1]
```
Output:
```
TypeError: 'str' object doesn't support item deletion
```

Now we are deleting entire string.

```
str1 = "JAVATPOINT"
del str1
print(str1)
```
Output:
```
NameError: name 'str1' is not defined
```

String Operators

Operator	Description
+	It is known as concatenation operator used to join the strings given either side of the operator.
*****	It is known as repetition operator. It concatenates the multiple copies of the same string.
[]	It is known as slice operator. It is used to access the sub-strings of a particular string.
[:]	It is known as range slice operator. It is used to access the characters from the specified range.
in	It is known as membership operator. It returns if a particular sub-string is present in the specified string.
not in	It is also a membership operator and does the exact reverse of in. It returns true if a particular substring is not present in the specified string.
r/R	It is used to specify the raw string. Raw strings are used in the cases where we need to print the actual meaning of escape characters such as "C://python". To define any string as a raw string, the character r or R is followed by the string.
%	It is used to perform string formatting. It makes use of the format specifiers used in C programming like %d or %f to map their values in python. We will discuss how formatting is done in python.

Consider the following example to understand the real use of Python operators.

```
str = "Hello"
str1 = " world"
print(str*3) # prints HelloHelloHello
print(str+str1)# prints Hello world
print(str[4]) # prints o
print(str[2:4]); # prints ll
print('w' in str) # prints false as w is not present in str
print('wo' not in str1) # prints false as wo is present in str1.
print(r'C://python39') # prints C://python37 as it is written
print("The string str : %s"%(str)) # prints The string str : Hello
```
Output:
```
HelloHelloHello
Hello world
o
ll
False
False
C://python39
```

```
The string str : Hello
```

Escape Sequence
Let's suppose we need to write the text as - They said, "Hello what's going on?"- the given statement can be written in single quotes or double quotes but it will raise the SyntaxError as it contains both single and double-quotes.
Consider the following example to understand the real use of Python operators.

```
str = "They said, "Hello what's going on?""
print(str)
```

Output:

```
SyntaxError: invalid syntax
```

- We can use the triple quotes to accomplish this problem but Python provides the escape sequence.
- The backslash(/) symbol denotes the escape sequence. The backslash can be followed by a special character and it interpreted differently.
- The single quotes inside the string must be escaped. We can apply the same as in the double quotes.

Example

```
# using triple quotes
print('''They said, "What's there?"''')

# escaping single quotes
print('They said, "What\'s going on?"')

# escaping double quotes
print("They said, \"What's going on?\"")
```

OUTPUT:

```
They said, "What's there?"
They said, "What's going on?"
They said, "What's going on?"
```

The list of an escape sequence is given below:

Sr.	**Escape Sequence**	**Description**	**Example**
1.	\newline	It ignores the new line.	print("Python1 \ Python2 \ Python3") **Output:** Python1 Python2 Python3
2.	\\\\	Backslash	print("\\\\") **Output:** \\
3.	\\'	Single Quotes	print('\\'') **Output:** '
4.	\\"	Double Quotes	print("\\"") **Output:** "
5.	\a	ASCII Bell	print("\a")
6.	\b	ASCII Backspace(BS)	print("Hello \b World") **Output:** Hello World
7.	\f	ASCII Formfeed	print("Hello \f World!") Hello World!
8.	\n	ASCII Linefeed	print("Hello \n World!")

			Output: Hello World!
9.	\r	ASCII Carriege Return(CR)	print("Hello \r World!") **Output:** World!
10.	\t	ASCII Horizontal Tab	print("Hello \t World!") **Output:**Hello World!
11.	\v	ASCII Vertical Tab	print("Hello \v World!") **Output:** Hello World!
12.	\ooo	Character with octal value	print("\110\145\154\154\157") **Output:**Hello
13	\xHH	Character with hex value.	print("\x48\x65\x6c\x6c\x6f") **Output:**Hello

Here is the simple example of escape sequence.

```
print("C:\\Users\\Bhumika\\Python39\\Lib")
print("This is the \n multiline quotes")
print("This is \x48\x45\x58 representation")
Output:
C:\Users\Bhumika\Python39\Lib
This is the
 multiline quotes
This is HEX representation
```

Python String Formatting Using % Operator

Python allows us to use the format specifiers used in C's printf statement. The format specifiers in Python are treated in the same way as they are treated in C. However, Python provides an additional operator **%**, which is used as an interface between the format specifiers and their values. In other words, we can say that it binds the format specifiers to the values.

Consider the following example.

```
Integer = 10
Float = 1.290
String = "Mihir"
print("Hi I am Integer ... My value is %d\nHi I am float ... My value is %f\nHi I am string
... My value is %s"%(Integer,Float,String))
Output:
Hi I am Integer ... My value is 10
Hi I am float ... My value is 1.290000
Hi I am string ... My value is Mihir
```

3.1.3 String Methods:(center, count, join, len, max, min, replace, lower, upper, replace, split)

No	Method & Description

1	format(value) : It returns a formatted version of S, using the passed value.
	Eg: # Using Curly braces Or empty placeholders
	print("{} and {} both are the best friend".format("Devansh","Abhishek"))
	# Positional Argument
	print("{1} and {0} best players ".format("Virat","Rohit"))
	# Named Argument
	print("{a},{b} ,{c}".format(a="James",b="Peter",c="Ricky"))

2	center(width ,fillchar) : It returns a space padded string with the original string centred with equal number of left and right spaces.
	Eg: str1 = "Hello Javatpoint"
	# Calling function
	str2 = str1.center(20,'#')
	# Displaying result
	print("Old value:", str1)
	print("New value:", str2)

3	string.count(value, start, end) : It returns the number of times a specified value appears in the string. **value:** Required. A String. The string to value to search for **start:** Optional. An Integer. The position to start the search. Default is 0 **end:** Optional. An Integer. The position to end the search. Default is the end of the string **#Eg1** txt = "I love apples, apple are my favorite fruit" x = txt.count("apple") print("Total Count of apple is :" ,x) **#Eg2:** txt = "I love apples, apple are my favorite fruit" x = txt.count("apple", 10, 24) print("Total Count of apple from position :",x)

4	join(seq) : It merges the strings representation of the given sequence.
	Eg :str = "->" # string
	list = {'Java','C#','Python'} # iterable
	str2 = str.join(list)
	print(str2)

5	len(string) : It returns the length of a string. **Eg:** x = len("Hello") print("length is :",x)
6	max() : It returns the item with the highest value, or the item with the highest value in an iterable. If the values are strings, an alphabetically comparison is done. **Eg:** x = max("Ram", "John", "Vicky") print("Max Value is :",x)
7	min():It returns the item with the lowest value, or the item with the lowest value in an iterable. If the values are strings, an alphabetically comparison is done. **Eg:** x = min("Ram", "John", "Vicky") print("Min Value is :",x)

8	replace(old,new[,count]) : It replaces the old sequence of characters with the new sequence. The max characters are replaced if max is given.
	Eg1: txt = "one one was a race horse, two two was one too."
	x = txt.replace("one", "three")
	print(x)

	Eg2: txt = "one one was a race horse, two two was one too."
	x = txt.replace("one", "three", 2)
	print(x)

9	upper(): It converts all the characters of a string to Upper Case.
	Eg: txt = " sdj international college"
	x = txt.upper()
	print("Value is in Upper case : ", x)

10	lower() : It converts all the characters of a string to Lower case.
	Eg: txt = "SDJ INTERNATIONAL COLLEGE"
	x = txt.lower()
	print("Value is in Lower case : ", x)

11	split(separator, maxsplit) : Splits the string according to the delimiter str. The string splits according to the space if the delimiter is not provided. It returns the list of substring concatenated with the delimiter.
	separator: Optional. Specifies the separator to use when splitting the string. By default any whitespace is a separator
	maxsplit: Optional. Specifies how many splits to do. Default value is -1, which is "all occurrences"
	Eg1: txt = "apple#banana#cherry#orange"
	x = txt.split("#")
	print(x)
	Eg2: txt = "apple#banana#cherry#orange"
	# setting the maxsplit parameter to 1, will return a list with 2 elements!
	x = txt.split("#", 1)
	print(x)

3.2 Operators:

The operator can be defined as a symbol which is responsible for a particular operation between two operands. Operators are the pillars of a program on which the logic is built in a specific programming language. Python provides a variety of operators, which are described as follows.

3.2.1 Arithmetic Operators (+,-,*, /, %,**,//)

Arithmetic operators are used to perform arithmetic operations between two operands. It includes + (addition), - (subtraction), *(multiplication), /(divide), %(reminder), //(floor division), and exponent (**) operators.

Consider the following table for a detailed explanation of arithmetic operators.

Operator	**Description**
+ (Addition)	It is used to add two operands. For example, if a = 20, b = 10 => a+b = 30
- (Subtraction)	It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value results negative. For example, if a = 20, b = 10 => a - b = 10
/ (divide)	It returns the quotient after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a/b = 2.0
*** (Multiplication)**	It is used to multiply one operand with the other. For example, if a = 20, b = 10 => a * b = 200
% (reminder)	It returns the reminder after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a%b = 0
**** (Exponent)**	It is an exponent operator represented as it calculates the first operand power to the second operand.
// (Floor division)	It gives the floor value of the quotient produced by dividing the two operands.

3.2.2 Assignment Operators (=,+=,-=,/=,*=,//=)

The assignment operators are used to assign the value of the right expression to the left operand. The assignment operators are described in the following table.

Operator	Description
=	It assigns the value of the right expression to the left operand.
+=	It increases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 10, b = 20 => a+ = b will be equal to a = a+ b and therefore, a = 30.
-=	It decreases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 20, b = 10 => a- = b will be equal to a = a- b and therefore, a = 10.
*=	It multiplies the value of the left operand by the value of the right operand and assigns the modified value back to then the left operand. For example, if a = 10, b = 20 => a* = b will be equal to a = a* b and therefore, a = 200.
%=	It divides the value of the left operand by the value of the right operand and assigns the reminder back to the left operand. For example, if a = 20, b = 10 => a % = b will be equal to a = a % b and therefore, a = 0.
**=	a**=b will be equal to a=a**b, for example, if a = 4, b =2, a**=b will assign 4**2 = 16 to a.
//=	A//=b will be equal to a = a// b, for example, if a = 4, b = 3, a//=b will assign 4//3 = 1 to a.

3.2.3 Comparison Operators (==, !=, >,<,>=,<=)

Comparison operators are used to comparing the value of the two operands and returns Boolean true or false accordingly. The comparison operators are described in the following table.

Operator	Description
==	If the value of two operands is equal, then the condition becomes true.
!=	If the value of two operands is not equal, then the condition becomes true.
<=	If the first operand is less than or equal to the second operand, then the condition becomes true.
>=	If the first operand is greater than or equal to the second operand, then the condition becomes true.
>	If the first operand is greater than the second operand, then the condition becomes true.
<	If the first operand is less than the second operand, then the condition becomes true.

3.2.4 Logical Operators (and, or, not)

The logical operators are used primarily in the expression evaluation to make a decision. Python supports the following logical operators.

Operator	Description
and	If both the expression are true, then the condition will be true. If a and b are the two expressions, a → true, b → true => a and b → true.
or	If one of the expressions is true, then the condition will be true. If a and b are the two expressions, a → true, b → false => a or b → true.
not	If an expression a is true, then not (a) will be false and vice versa.

3.2.5 Identity and member operators (is, is not, in, not in)

Membership Operators

Python membership operators are used to check the membership of value inside a Python data structure. If the value is present in the data structure, then the resulting value is true otherwise it returns false.

Operator	Description
in	It is evaluated to be true if the first operand is found in the second operand (list, tuple, or dictionary).
not in	It is evaluated to be true if the first operand is not found in the second operand (list, tuple, or dictionary).

Identity Operators

The identity operators are used to decide whether an element certain class or type.

Operator	Description
is	It is evaluated to be true if the reference present at both sides point to the same object.
is not	It is evaluated to be true if the reference present at both sides do not point to the same object.

Operator Precedence

The precedence of the operators is essential to find out since it enables us to know which operator should be evaluated first. The precedence table of the operators in Python is given below.

Operator	Description	Associativity
( )	Parentheses	left to right
**	Exponent	right to left
* / %	Multiplication / division / modulus	left to right
+ -	Addition / subtraction	left to right
<< >>	Bitwise left shift / Bitwise right shift	left to right
< <= > >=	Relational operators: less than / less than or equal to / greater than / greater than or equal to	left to right
== !=	Relational operators: is equal to / is not equal to	left to right
is, is not in, not in	Identity operators Membership operators	left to right
&	Bitwise AND operator	left to right
^	Bitwise exclusive OR operator	left to right
|	Bitwise inclusive OR operator	left to right
not	Logical NOT	right to left
and	Logical AND	left to right
or	Logical OR	left to right
= += -= *= /= %= &= ^= |= <<= >>=	Assignment operators: Addition / subtraction Multiplication / division Modulus / bitwise AND Bitwise exclusive / inclusive OR Bitwise shift left / right shift	right to left