

Module 3

Strings: Creating and Storing Strings; Accessing String Characters; the str() function; Operations on Strings Concatenation, Comparison, Slicing and Joining, Traversing; Python built-in String Methods.

Python String

In Python, a string is a sequence of characters enclosed in quotes. It can include letters, numbers, symbols or spaces. Since Python has no separate character type, even a single character is treated as a string with length one. Strings are widely used for text handling and manipulation.

Creating a String:

Strings can be created using either **single** ('...') or **double** ("...") quotes. Both behave the same.

Example: Creating two equivalent strings one with single and other with double quotes.

```
s1 = 'GfG' # single quote
s2 = "GfG" # double quote
print(s1)
print(s2)
```

Output

```
GfG
GfG
```

Multi-line Strings:

Use triple quotes ('''...''') or ("""...""") for strings that span multiple lines. Newlines are preserved.

Example: Define and print multi-line strings using both styles.

```
s = """I am Learning
Python String on GeeksforGeeks"""
print(s)

s = '''I'm a
Geek'''
print(s)
```

Output

```
I am Learning
Python String on GeeksforGeeks
I'm a
Geek
```

Accessing characters in String:

Strings are indexed sequences. Positive indices start at **0** from the **left**; negative indices start at **-1** from the **right** as represented in below image:

0	1	2	3	4	5	6	7	8	9	10	11	12
G	E	E	K	S	F	O	R	G	E	E	K	S
-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Indices of string in reverse

Example 1: Access specific characters through positive indexing.

```
s = "GeeksforGeeks"
print(s[0])    # first character
print(s[4])    # 5th character
```

Output

```
G
s
```

Note: Accessing an index out of range will cause an `IndexError`. Only integers are allowed as indices and using a float or other types will result in a `TypeError`.

Example 2: Read characters from the end using [negative indices](#).

```
s = "GeeksforGeeks"  
print(s[-10])    # 3rd character  
print(s[-5])     # 5th character from end
```

Output

```
k  
G
```

String Slicing:

Slicing is a way to extract a portion of a string by specifying the **start** and **end** indexes. The syntax for slicing is **string[start:end]**, where **start** starting index and **end** is stopping index (excluded).

Example: In this example we are slicing through range and reversing a string.

```
s = "GeeksforGeeks"  
print(s[1:4])    # characters from index 1 to 3  
print(s[:3])     # from start to index 2  
print(s[3:])     # from index 3 to end  
print(s[::-1])   # reverse string
```

Output

```
EEK  
Gee  
ksforGeeks  
skeeGrofskeeG
```

String Iteration

Strings are iterable; you can loop through characters one by one.

Example: Here, it print each character on its own line.

```
s = "Python"
for char in s:
    print(char)
```

Output

```
P
y
t
h
o
n
```

Explanation: for loop pulls characters in order and each iteration prints the next character.
String Immutability

Strings are **immutable**, which means that they cannot be changed after they are created. If we need to manipulate strings then we can use methods like **concatenation**, **slicing** or **formatting** to create new strings based on original.

Example: In this example we are changing first character by building a new string.

```
s = "geeksforGeeks"
s = "G" + s[1:]    # create new string
print(s)
```

Output

```
GeeksforGeeks
```

Deleting a String:

In Python, it is not possible to delete individual characters from a string since strings are immutable. However, we can delete an entire string variable using the [del](#) keyword.

Example: Here, we are using del keyword to delete a string.

```
s = "GfG"
del s
```

Note: After deleting the string if we try to access s then it will result in a **NameError** because variable no longer exists.

Updating a String

As strings are immutable, “updates” create new strings using slicing or methods such as `replace()`.

Example: This code fix the first letter and replace a word.

```
s = "hello geeks"
s1 = "H" + s[1:]           # update first character
s2 = s.replace("geeks", "GeeksforGeeks") # replace word
print(s1)
print(s2)
```

Output

```
Hello geeks
hello GeeksforGeeks
```

Explanation:

- **s1:** slice from index 1 onward and prepend "H".
- **s2:** `replace("geeks", "GeeksforGeeks")` returns a new string.

Common String Methods

Python provides various built-in methods to manipulate strings. Below are some of the most useful methods:

1. **len():** The `len()` function returns the total number of characters in a string (including spaces and punctuation).

Example:

```
s = "GeeksforGeeks"
print(len(s))
```

Output

```
13
```

2. **upper() and lower():** [upper\(\)](#) method converts all characters to uppercase whereas, [lower\(\)](#) method converts all characters to lowercase.

Example:

```
s = "Hello World"
print(s.upper())
print(s.lower())
```

Output

```
HELLO WORLD
hello world
```

3. **strip() and replace():** [strip\(\)](#) removes leading and trailing whitespace from the string and [replace\(\)](#) replaces all occurrences of a specified substring with another.

Example:

```
s = "  Gfg  "
print(s.strip())

s = "Python is fun"
print(s.replace("fun", "awesome"))
```

Output

```
Gfg
Python is awesome
```

Concatenating and Repeating Strings

We can concatenate strings using + operator and repeat them using * operator.

1. Strings can be combined by using + **operator**.

Example: Join two words with a space.

```
s1 = "Hello"
s2 = "World"
print(s1 + " " + s2)
```

Output

```
Hello World
```

2. We can repeat a string multiple times using ***** operator.

Example: Repeat a greeting three times.

```
s = "Hello "  
print(s * 3)
```

Output

```
Hello Hello Hello
```

Formatting Strings

Python provides several ways to include variables inside strings.

1. Using f-strings

The simplest and most preferred way to format strings is by using [f-strings](#).

Example: Embed variables directly using {} placeholders.

```
name = "Alice"  
age = 22  
print(f"Name: {name}, Age: {age}")
```

Output

```
Name: Alice, Age: 22
```

2. Using format()

Another way to format strings is by using `format()` method.

Example: Use placeholders {} and pass values positionally.

```
s = "My name is {} and I am {} years old.".format("Alice", 22)
print(s)
```

Output

```
My name is Alice and I am 22 years old.
```

String Membership Testing

`in` keyword checks if a particular substring is present in a string.

Example: Here, we are testing for the presence of substrings.

```
s = "GeeksforGeeks"
print("Geeks" in s)
print("GfG" in s)
```

Output

```
True
False
```

List: Creating Lists; Operations on Lists; Built-in Functions on Lists, Python built-in Methods, Nested Lists.

Python Lists:

In Python, a list is a built-in data structure that can hold an ordered collection of items. Unlike arrays in some languages, Python lists are very flexible:

- Can contain duplicate items
- **Mutable:** items can be modified, replaced, or removed
- **Ordered:** maintains the order in which items are added
- **Index-based:** items are accessed using their position (starting from 0)
- Can store mixed data types (integers, strings, booleans, even other lists)

Creating a List:

Lists can be created in several ways, such as using square brackets, the `list()` constructor or by repeating elements. Let's look at each method one by one with example:

1. Using Square Brackets:

We use square brackets `[]` to create a list directly.

```
a = [1, 2, 3, 4, 5] # List of integers
b = ['apple', 'banana', 'cherry'] # List of strings
c = [1, 'hello', 3.14, True] # Mixed data types

print(a)
print(b)
print(c)
```

Output

```
[1, 2, 3, 4, 5]
['apple', 'banana', 'cherry']
[1, 'hello', 3.14, True]
```

2. Using `list()` Constructor

We can also create a list by passing an iterable (like a [tuple](#), [string](#) or another list) to the `list()` function.

```
a = list((1, 2, 3, 'apple', 4.5))
print(a)

b = list("GFG")
print(b)
```

Output

```
[1, 2, 3, 'apple', 4.5]
['G', 'F', 'G']
```

3. Creating List with Repeated Elements:

We can use the multiplication operator `*` to create a list with repeated items.

```
a = [2] * 5
b = [0] * 7

print(a)
print(b)
```

Output

```
[2, 2, 2, 2, 2]
[0, 0, 0, 0, 0, 0, 0]
```

Accessing List Elements:

Elements in a list are accessed using indexing. Python indexes start at 0, so `a[0]` gives the first element. Negative indexes allow access from the end (e.g., `-1` gives the last element).

```
a = [10, 20, 30, 40, 50]
print(a[0])
print(a[-1])
print(a[1:4])    # elements from index 1 to 3
```

Output

```
10
50
[20, 30, 40]
```

Adding Elements into List:

We can add elements to a list using the following methods:

- **append()**: Adds an element at the end of the list.
- **extend()**: Adds multiple elements to the end of the list.
- **insert()**: Adds an element at a specific position.
- **clear()**: removes all items.

```
a = []

a.append(10)
print("After append(10):", a)

a.insert(0, 5)
print("After insert(0, 5):", a)

a.extend([15, 20, 25])
print("After extend([15, 20, 25]):", a)

a.clear()
print("After clear():", a)
```

Output

```
After append(10): [10]
After insert(0, 5): [5, 10]
After extend([15, 20, 25]): [5, 10, 15, 20, 25]
After clear(): []
```

Updating Elements into List

Since lists are mutable, we can update elements by accessing them via their index.

```
a = [10, 20, 30, 40, 50]
a[1] = 25
print(a)
```

Output

```
[10, 25, 30, 40, 50]
```

Removing Elements from List:

We can remove elements from a list using:

- **remove():** Removes the first occurrence of an element.
- **pop():** Removes the element at a specific index or the last element if no index is specified.
- **del statement:** Deletes an element at a specified index.

```
a = [10, 20, 30, 40, 50]

a.remove(30)
print("After remove(30):", a)

popped_val = a.pop(1)
print("Popped element:", popped_val)
print("After pop(1):", a)

del a[0]
print("After del a[0]:", a)
```

Output

```
After remove(30): [10, 20, 40, 50]
Popped element: 20
After pop(1): [10, 40, 50]
After del a[0]: [40, 50]
```

Iterating Over Lists

We can iterate over lists using [loops](#), which is useful for performing actions on each item.

```
a = ['apple', 'banana', 'cherry']
for item in a:
    print(item)
```

Output

```
apple
banana
cherry
```

Nested Lists

A nested list is a list within another list, which is useful for representing matrices or tables. We can access nested elements by chaining indexes.

```
matrix = [ [1, 2, 3],  
           [4, 5, 6],  
           [7, 8, 9] ]  
print(matrix[1][2])
```

Output

6

List Comprehension:

List comprehension is a concise way to create lists using a single line of code. It is useful for applying an operation or filter to items in an iterable, such as a list or range.

```
squares = [x**2 for x in range(1, 6)]  
print(squares)
```

Output

[1, 4, 9, 16, 25]

Explanation:

- **for x in range(1, 6):** loops through each number from 1 to 5 (excluding 6).
- **x**2:** squares each number x.
- **[]:** collects all the squared numbers into a new list.

How Python Stores List Elements?

In Python, a list doesn't store actual values directly. Instead, it stores references (pointers) to objects in memory. This means numbers, strings and booleans are separate objects in memory and the list just keeps their addresses.

That's why modifying a mutable element (like another list or dictionary) can change the original object, while immutables remain unaffected.

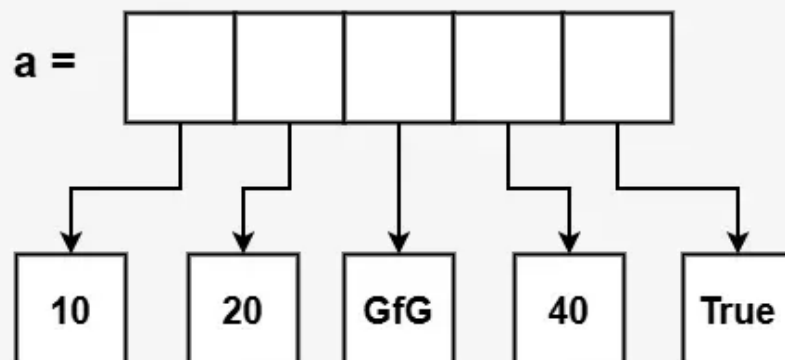
```
a = [10, 20, "GfG", 40, True]
print(a)
print(a[0])
print(a[1])
print(a[2])
```

Output

```
[10, 20, 'GfG', 40, True]
10
20
GfG
```

Explanation:

- The list `a` contains an integer (10, 20 and 40), a string ("GfG") and a boolean (True).
- Elements are accessed using indexing (`a[0]`, `a[1]`, etc.).
- Each element keeps its original type.



Python List

Python List