Python String

Till now, we have discussed numbers as the standard data-types in Python. In this section of the tutorial, we will discuss the most popular data type in Python, i.e., string.

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.

Syntax:

```
1. str = "Hi Python !"
```

Here, if we check the type of the variable **str** using a Python script

```
1. print(type(str)), then it will print a string (str).
```

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.

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**.

```
1. #Using single quotes
2. str1 = 'Hello Python'
3. print(str1)
4. #Using double quotes
5. str2 = "Hello Python"
6. print(str2)
7. #Using triple quotes
8. str3 = """Triple quotes are generally used for
9.     represent the multiline or
10.    docstring"'
```

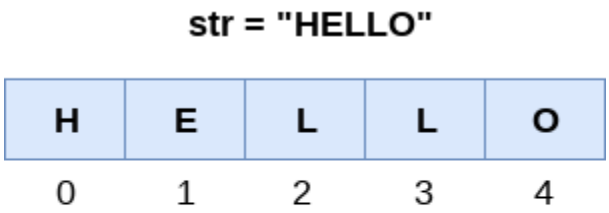
11. **print**(str3)

Output:

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

Strings indexing and 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'

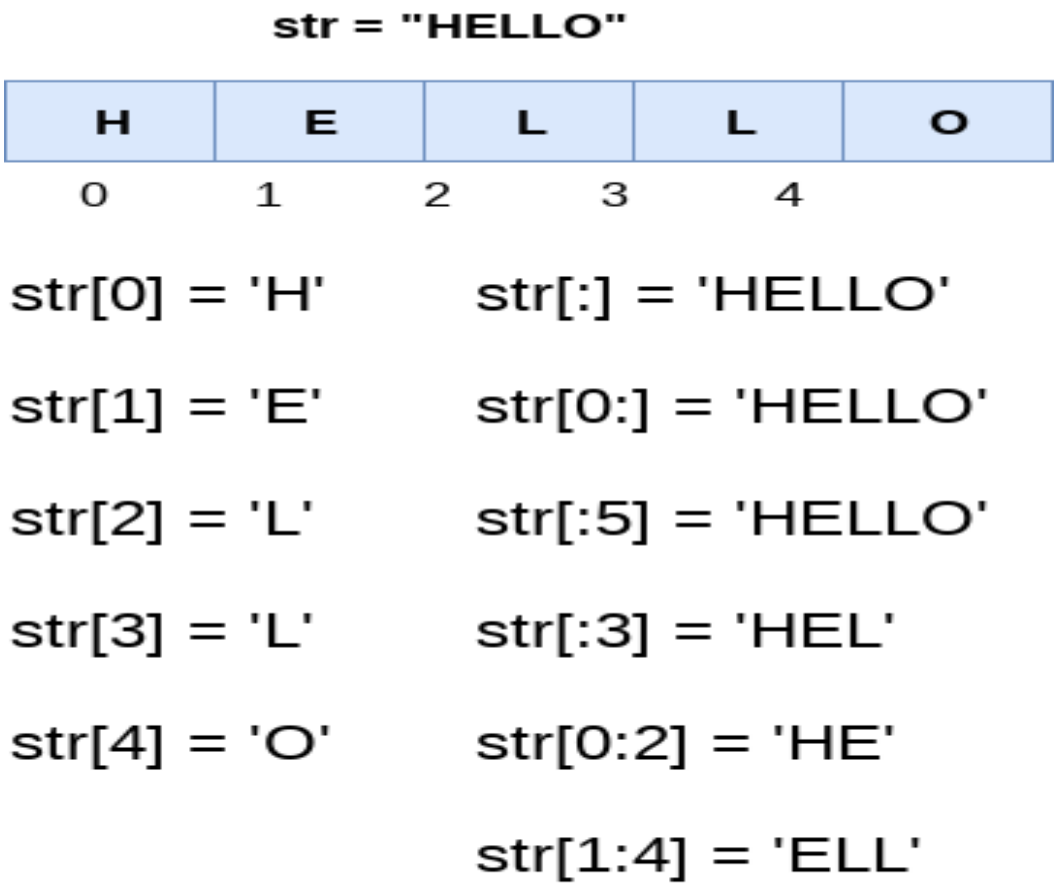
Consider the following example:

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

Output:

H
E
L
L
O
Index Error: 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.



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.

Consider the following example:

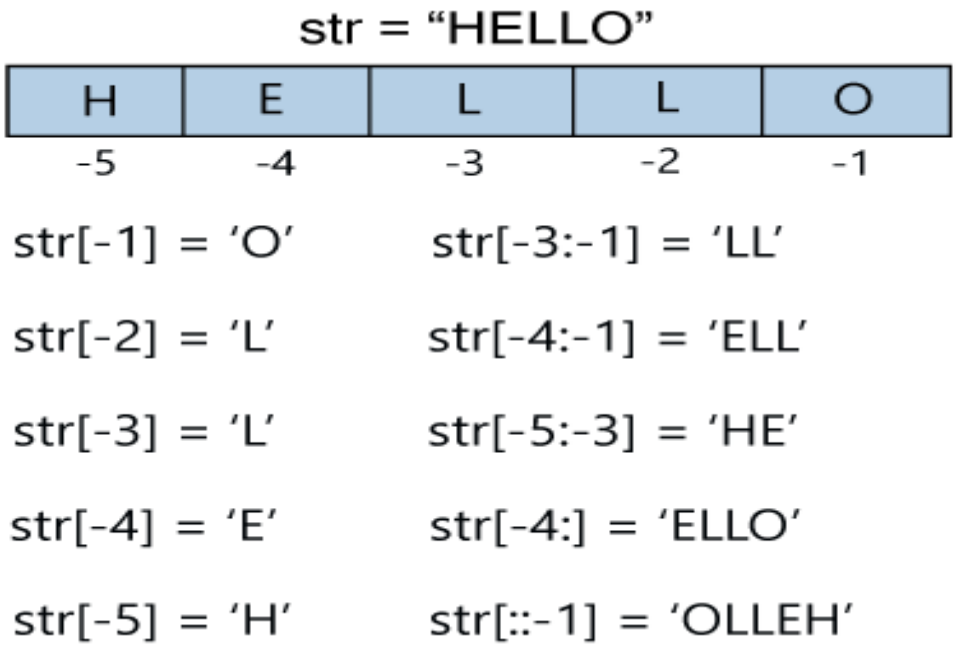
```
1. # Given String
2. str = "JAVATPOINT"
3. # Start Oth index to end
4. print(str[0:])
```

```
# 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.



Consider the following example

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

9. **print**(str[-12])

Output:

T
I
NT
OIN
ATPOI
TNIOPTAVAJ
IndexError: string index out of range

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.

Example 1

1. str = "HELLO"
2. str[0] = "h"
3. **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.

Example 2

1. str = "HELLO"
2. **print**(str)
3. str = "hello"
4. **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.

```
1. str = "JAVATPOINT"
2. del str[1]
```

Output:

TypeError: 'str' object doesn't support item deletion

Now we are deleting entire string.

```
1. str1 = "JAVATPOINT"
2. del str1
3. 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.

Example

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

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

Output:

HelloHelloHello
Hello world
o

ll
False
False
C://python37
The string str : Hello

Python String Formatting

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.

Example

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

```
1. str = "They said, "Hello what's going on?""
2. 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 -

```
1. # using triple quotes
2. print("""They said, "What's there?""")
3. 
4. # escaping single quotes
5. print('They said, "What\'s going on?"')
6. 
7. # escaping double quotes
8. 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.

```
1. print("C:\\Users\\DEVANSH SHARMA\\Python32\\Lib")
2. print("This is the \n multiline quotes")
3. print("This is \x48\x45\x58 representation")
```

Output:

C:\Users\DEVANSH SHARMA\Python32\Lib
This is the
multiline quotes
This is HEX representation

We can ignore the escape sequence from the given string by using the raw string. We can do this by writing **r** or **R** in front of the string. Consider the following example.

```
1. print(r"C:\\Users\\DEVANSH SHARMA\\Python32")
```

Output:

C:\\Users\\DEVANSH SHARMA\\Python32

The format() method

The **format()** method is the most flexible and useful method in formatting strings. The curly braces {} are used as the placeholder in the string and replaced by the **format()** method argument. Let's have a look at the given an example:

```
1. # Using Curly braces
2. print("{} and {} both are the best friend".format("Devansh","Abhishek"))
3.
4. #Positional Argument
5. print("{1} and {0} best players ".format("Virat","Rohit"))
6.
7. #Keyword Argument
8. print("{a},{b},{c}".format(a = "James", b = "Peter", c = "Ricky"))
```

Output:

Devansh and Abhishek both are the best friend

Rohit and Virat best players
James,Peter,Ricky

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.

1. Integer = 10;
2. Float = 1.290
3. String = "Devansh"
4. **print**("Hi I am Integer ... My value is %d\nHi I am float ... My value is %f\nHi I am s tring ... 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 Devansh

Python String functions

Python provides various in-built functions that are used for string handling. Many String fun

Method	**Description**
capitalize()	It capitalizes the first character of the String. This function is deprecated in python3
casefold()	It returns a version of s suitable for case-less comparisons.
center(width ,fillchar)	It returns a space padded string with the original string centred with equal number of left and right spaces.

count(string,begin,end)	It counts the number of occurrences of a substring in a String between begin and end index.
decode(encoding = 'UTF8', errors = 'strict')	Decodes the string using codec registered for encoding.
encode()	Encode S using the codec registered for encoding. Default encoding is 'utf-8'.
endswith(suffix ,begin=0,end=len(string))	It returns a Boolean value if the string terminates with given suffix between begin and end.
expandtabs(tabsize = 8)	It defines tabs in string to multiple spaces. The default space value is 8.
find(substring ,beginIndex, endIndex)	It returns the index value of the string where substring is found between begin index and end index.
format(value)	It returns a formatted version of S, using the passed value.
index(subsring, beginIndex, endIndex)	It throws an exception if string is not found. It works same as find() method.
isalnum()	It returns true if the characters in the string are alphanumeric i.e., alphabets or numbers and there is at least 1 character. Otherwise, it returns false.
isalpha()	It returns true if all the characters are alphabets and there is at least one character, otherwise False.
isdecimal()	It returns true if all the characters of the string are decimals.

isdigit()	It returns true if all the characters are digits and there is at least one character, otherwise False.
isidentifier()	It returns true if the string is the valid identifier.
islower()	It returns true if the characters of a string are in lower case, otherwise false.
isnumeric()	It returns true if the string contains only numeric characters.
isprintable()	It returns true if all the characters of s are printable or s is empty, false otherwise.
isupper()	It returns false if characters of a string are in Upper case, otherwise False.
isspace()	It returns true if the characters of a string are white-space, otherwise false.
istitle()	It returns true if the string is titled properly and false otherwise. A title string is the one in which the first character is upper-case whereas the other characters are lower-case.
isupper()	It returns true if all the characters of the string(if exists) is true otherwise it returns false.
join(seq)	It merges the strings representation of the given sequence.
len(string)	It returns the length of a string.

ljust(width[,fillchar])	It returns the space padded strings with the original string left justified to the given width.
lower()	It converts all the characters of a string to Lower case.
lstrip()	It removes all leading whitespaces of a string and can also be used to remove particular character from leading.
partition()	It searches for the separator sep in S, and returns the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings.
maketrans()	It returns a translation table to be used in translate function.
replace(old,new[,count])	It replaces the old sequence of characters with the new sequence. The max characters are replaced if max is given.
rfind(str,beg=0,end=len(str))	It is similar to find but it traverses the string in backward direction.
rindex(str,beg=0,end=len(str))	It is same as index but it traverses the string in backward direction.
rjust(width,[,fillchar])	Returns a space padded string having original string right justified to the number of characters specified.
rstrip()	It removes all trailing whitespace of a string and can also be used to remove particular character from trailing.

rsplit(sep=None, maxsplit = -1)	It is same as split() but it processes the string from the backward direction. It returns the list of words in the string. If Separator is not specified then the string splits according to the white-space.
split(str,num=string.count(str))	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.
splitlines(num=string.count('\n'))	It returns the list of strings at each line with newline removed.
startswith(str,beg=0,end=len(str))	It returns a Boolean value if the string starts with given str between begin and end.
strip([chars])	It is used to perform lstrip() and rstrip() on the string.
swapcase()	It inverts case of all characters in a string.
title()	It is used to convert the string into the title-case i.e., The string **meEruT** will be converted to Meerut.
translate(table,deletechars = '')	It translates the string according to the translation table passed in the function .
upper()	It converts all the characters of a string to Upper Case.
zfill(width)	Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).

Python List Vs Tuple

This tutorial will study the major differences between lists and tuples and how to handle these two data structures.

Lists and tuples are types of data structures that hold one or more than one objects or items in a predefined order. We can contain objects of any data type in a list or tuple, including the null data type defined by the None Keyword.

What is a List?

In other programming languages, list objects are declared similarly to arrays. Lists don't have to be homogeneous all the time, so they can simultaneously store items of different data types. This makes lists the most useful tool. The list is a kind of container data Structure of Python that is used to hold numerous pieces of data simultaneously. Lists are helpful when we need to iterate over some elements and keep hold of the items.

What is a Tuple?

A tuple is another data structure to store the collection of items of many data types, but unlike mutable lists, tuples are immutable. A tuple, in other words, is a collection of items separated by commas. Because of its static structure, the tuple is more efficient than the list.

Differences between Lists and Tuples

In most cases, lists and tuples are equivalent. However, there are some important differences to be explored in this article.

List and Tuple Syntax Differences

The syntax of a list differs from that of a tuple. Items of a tuple are enclosed by parentheses or curved brackets (), whereas items of a list are enclosed by square brackets [].

Example Code

```
1.  # Python code to show the difference between creating a list and a tuple
2.
3.  list_ = [4, 5, 7, 1, 7]
4.  tuple_ = (4, 1, 8, 3, 9)
5.
6.  print("List is: ", list_)
```

```
7. print("Tuple is: ", tuple_)
```

Output:

List is: [4, 5, 7, 1, 7]
Tuple is: (4, 1, 8, 3, 9)

We declared a variable named list_, which contains a certain number of integers ranging from 1 to 10. The list is enclosed in square brackets []. We also created a variable called tuple_, which holds a certain number of integers. The tuple is enclosed in curly brackets (). The type() method in Python returns the data type of the data structure or object passed to it.

Example Code

```
1. # Code to print the data type of the data structure using the type() function
2. print( type(list_) )
3. print( type(tuple_) )
```

Output:

<class 'list'>
<class 'tuple'>

Mutable List vs. Immutable Tuple

An important difference between a list and a tuple is that lists are mutable, whereas tuples are immutable. What exactly does this imply? It means a list's items can be changed or modified, whereas a tuple's items cannot be changed or modified.

We can't employ a list as a key of a dictionary because it is mutable. This is because a key of a Python dictionary is an immutable object. As a result, tuples can be used as keys to a dictionary if required.

Let's consider the example highlighting the difference between lists and tuples in immutability and mutability.

Example Code

```
1. # Updating the element of list and tuple at a particular index
2.
3. # creating a list and a tuple
4. list_ = ["Python", "Lists", "Tuples", "Differences"]
5. tuple_ = ("Python", "Lists", "Tuples", "Differences")
6.
7. # modifying the last string in both data structures
8. list_[3] = "Mutable"
```

```
print( list_ )
try:
    tuple_[3] = "Immutable"
    print( tuple_ )
except TypeError:
    print( "Tuples cannot be modified because they are immutable" )
```

Output:

['Python', 'Lists', 'Tuples', 'Mutable']
Tuples cannot be modified because they are immutable

We altered the string of list_ at index 3 in the above code, which the Python interpreter updated at index 3 in the output. Also, we tried to modify the last index of the tuple in a try block, but since it raised an error, we got output from the except block. This is because tuples are immutable, and the Python interpreter raised TypeError on modifying the tuple.

Size Difference

Since tuples are immutable, Python allocates bigger chunks of memory with minimal overhead. Python, on the contrary, allots smaller memory chunks for lists. The tuple would therefore have less memory than the list. If we have a huge number of items, this makes tuples a little more memory-efficient than lists.

For example, consider creating a list and a tuple with the identical items and comparing their sizes:

Example Code

```
# Code to show the difference in the size of a list and a tuple

#creating a list and a tuple
list_ = ["Python", "Lists", "Tuples", "Differences"]
tuple_ = ("Python", "Lists", "Tuples", "Differences")
# printing sizes
print("Size of tuple: ", tuple_.__sizeof__())
print("Size of list: ", list_.__sizeof__())
```

Output:

Size of tuple: 28
Size of list: 52

Available Functions

Tuples have fewer built-in functions than lists. We may leverage the in-built function dir([object] to access all the corresponding methods for the list and tuple.

Example Code

```
1. # printing directory of list
2. dir(list_)
```

Output:

['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

Example Code

```
1. # Printing directory of a tuple
2. print( dir(tuple_), end = '', '' )
```

Output:

['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'count', 'index']

As we can observe, a list has many more methods than a tuple. With intrinsic functions, we can perform insert and pop operations and remove and sort items from the list not provided in the tuple.

Tuples and Lists: Key Similarities

- They both hold collections of items and are heterogeneous data types, meaning they can contain multiple data types simultaneously.
- They're both ordered, which implies the items or objects are maintained in the same order as they were placed until changed manually.
- Because they're both sequential data structures, we can iterate through the objects they hold; hence, they are iterables.
- An integer index, enclosed in square brackets [index], can be used to access objects of both data types.