M.Kumarasamy
College of Engineering
NAAC Accredited Autonomous Institution
Approved by AICTE & Affiliated to Anna University
ISO 9001:2015 Certified Institution
Thalavapalayam, Karur - 639 113, TAMILNADU.

Module III - Strings, Lists and Tuples

Strings: Initializing and Accessing Elements of a String - Slicing - Immutability - Built-in String methods and functions - Concatenating - Appending and Multiplying Strings - String modules
Lists: Creation - Accessing values - Slicing - List methods - Build-in functions - List comprehension
Tuples: Creation - Operations on tuples - Traversing - Indexing and Slicing - Tuple assignment - Build-in Functions –Immutability versus Reassignment - Unpacking tuples.

Python Strings:

- A string is a sequence of characters.
- Python treats anything inside quotes as a string.
- This includes letters, numbers, and symbols.
- Python has no character data type so single character is a string of length 1.

```
s = "ABC"
print(s[1]) # access 2nd char
s1 = s + s[0] # update
print(s1) # print
```

Output

```
B

ABCA
```

Creating a String:

Strings can be created using either **single (‘)** or **double (“)** quotes.

```
s1 = 'ABC'
s2 = "ABC"
print(s1)
print(s2)
```

Output

```
ABC

ABC
```

Multi-line Strings:

If we need a string to span multiple lines, then we can use **triple quotes (”’ or “””)**.

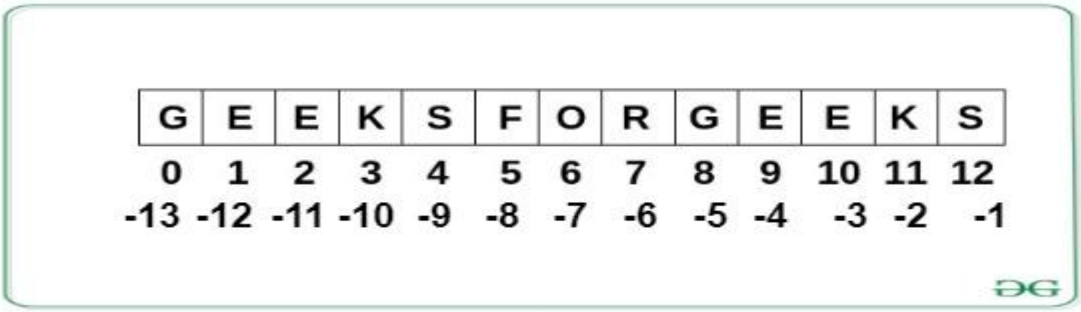
```
s = """I am Learning
Python String """
print(s)
s = '''I'm a
Student'''
print(s)
```

Output

```
I am Learning

Python String

I'm a

Student
```

Accessing characters in Python String:

- Strings in Python are sequences of characters, so we can access individual characters using **indexing.**
- Strings are indexed starting from **0** and **-1** from end**.** This allows us to retrieve specific characters from the string.



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

Python String syntax indexing

```
s = "Hello World"
# Accesses first character: 'H'
print(s[0])
# Accesses 5th character: 'o'
print(s[4])
```

Output

```
H

o
```

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**.

Access string with Negative Indexing:

Python allows negative address references to access characters from back of the String, e.g. -1 refers to the last character, -2 refers to the second last character, and so on.

```
s = " Hello World"

# Accesses 3rd character: 'e'
print(s[-10])

# Accesses 5th character from end: 'W'
print(s[-5])
```

Output

```
e

W
```

String Slicing:

- Slicing is a way to extract 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).

```
s = " Hello World"

# Retrieves characters from index 1 to 3: 'e11'
print(s[1:4])

# Retrieves characters from beginning to index 2: 'Hel'
print(s[:3])

# Retrieves characters from index 3 to the end: ' lo World'
print(s[3:])

# Reverse a string
print(s[::-1])
```

Output

```
e11

Hel

lo World'

dlroW olleH
```

String Immutability:

Strings in Python are immutable.

- This 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 the original.

```
s = " hello World "

# Trying to change the first character raises an error
# s[0] = 'H'  # Uncommenting this line will cause a TypeError

# Instead, create a new string
s = "H" + s[1:]
print(s)
```

Output

```
Hello World
```

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.

```
s = "ABC"
# Deletes entire string
del s
```

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

Updating a String:

To update a part of a string we need to create a new string since strings are immutable.

```
s = "hello World"

# Updating by creating a new string
s1 = "H" + s[1:]

# replacing "World" with "Students"
s2 = s.replace("World", "Students")
print(s1)
print(s2)
```

Output

```
Hello World

hello Students
```

Explanation:

- **For s1,** The original string **s** is sliced from index 1 to end of string and then concatenate “H” to create a new string **s1**.
- **For s2,** we can created a new string s2 and used replace() method to replace ‘World’ with ‘Students’.

String Comparision:

- **Compare** two strings using comparison operators such as ==, !=,<,<=,>, >=
- Python compares strings based on their corresponding ASCII values

Example:

```
str1="green"
str2="black"
print("Is both Equal:", str1==str2)
print("Is str1> str2:", str1>str2)
print("Is str1< str2:", str1<str2)
```

Output:

```
Is both Equal: False
Is str1 > str2: True
Is str1 < str2: False
```

Explanation:

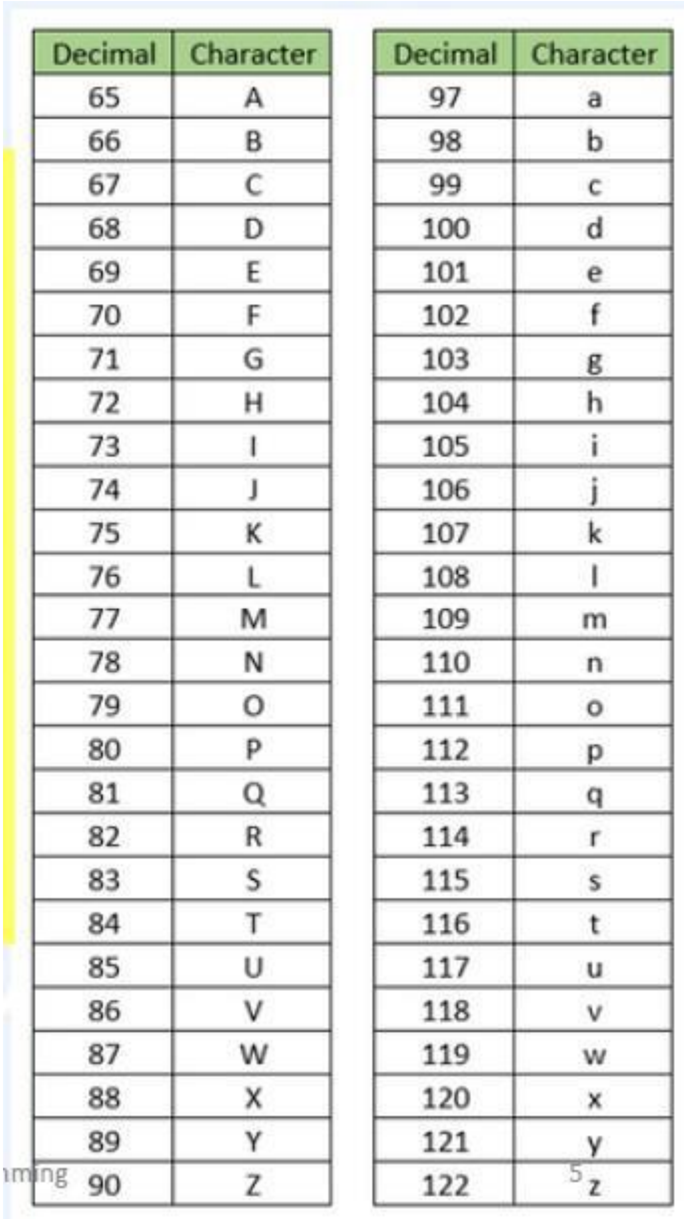
1. **Comparison using == (Equality Check)**
   "green" == "black" → False, because the two strings are different.

2. **Comparison using > (Greater Than)**

- String comparison in Python is lexicographical (dictionary order), meaning characters are compared based on their Unicode (ASCII) values.
- The first character of "green" is 'g' (Unicode: 103), and the first character of "black" is 'b' (Unicode: 98).
- Since 'g' (103) is greater than 'b' (98), "green" > "black" is True.

3. **Comparison using < (Less Than)**

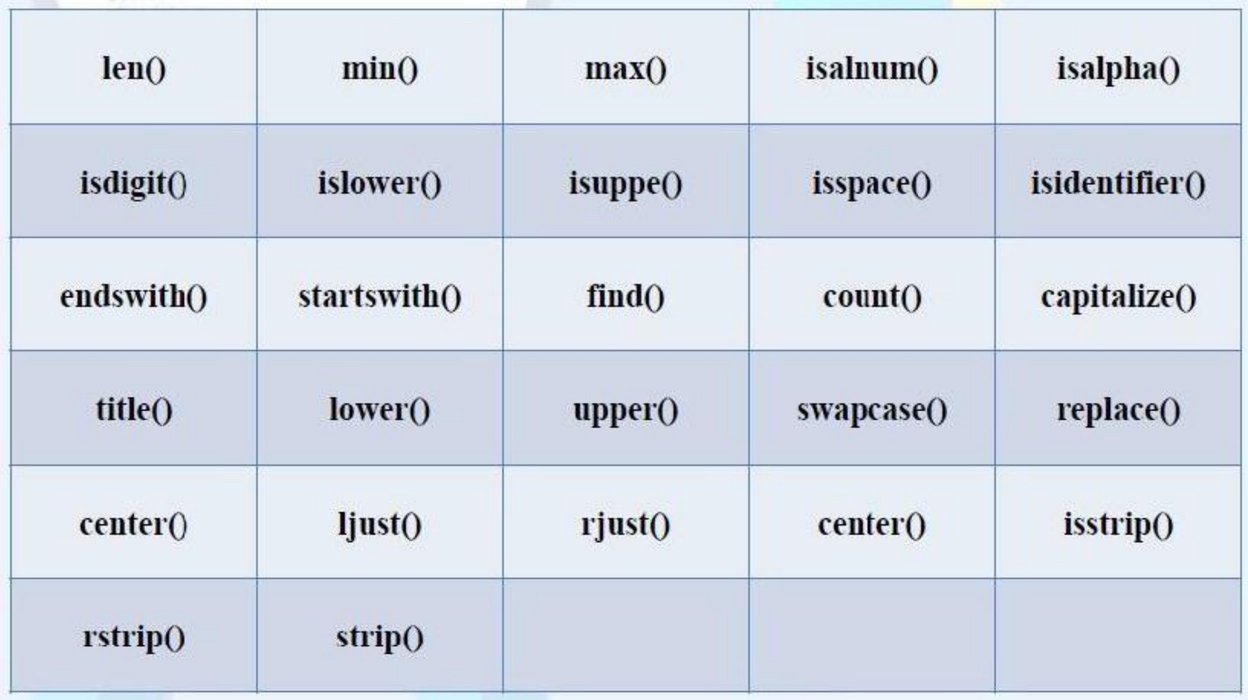
- Since "green" > "black" is True, "green" < "black" must be False.



Decimal	Character	Decimal	Character
65	A	97	a
66	B	98	b
67	C	99	c
68	D	100	d
69	E	101	e
70	F	102	f
71	G	103	g
72	H	104	h
73	I	105	i
74	J	106	j
75	K	107	k
76	L	108	l
77	M	109	m
78	N	110	n
79	O	111	o
80	P	112	p
81	Q	113	q
82	R	114	r
83	S	115	s
84	T	116	t
85	U	117	u
86	V	118	v
87	W	119	w
88	X	120	x
89	Y	121	y
90	Z	122	z

Common String Methods:

Python provides a various built-in methods to manipulate strings.



len()	min()	max()	isalnum()	isalpha()
isdigit()	islower()	isuppe()	isspace()	isidentifier()
endswith()	startswith()	find()	count()	capitalize()
title()	lower()	upper()	swapcase()	replace()
center()	ljust()	rjust()	center()	isstrip()
rstrip()	strip()			

1. len():

The len() function returns the total number of characters in a string.

Example:

```
s = "Hello Students"
print(len(s))
```

Output

```
14
```

2. max() and min():

max() - Largest value in a string based on ASCII values

min() - Smallest value in a string based on ASCII values

Example:

```
name=input("Enter Your name:")
print("Welcome",name)
print("Length of your name:",len(name))
print("Maximum value of chararacter in your name", max(name))
print("Minimum value of character in your name",min(name))
```

Output

```
Enter Your name: Alice
Welcome Alice
Length of your name: 5
Maximum value of character in your name: l
Minimum value of character in your name: A
```

3. upper() and lower():

upper() method converts all characters to uppercase.
lower() method converts all
characters to lowercase.

Example:

```
s = "Hello World"
print(s.upper())   # output: HELLO WORLD
print(s.lower())   # output: hello world
```

Output

```
HELLO WORLD

hello world
```

4. strip(), lstrip, rstrip:

strip() removes leading and trailing whitespace from the string.
lstrip() removes leading (left-side) spaces or specified characters from a string.
Rstrip() removes trailing (right-side) spaces or specified characters from a string.

Example:

```
s = "   ABC   "
# Removes spaces from both ends
print(s.strip())
print(s.lstrip())
print(s.rstrip())
```

Output:

```
ABC
ABC
   ABC
```

5. replace():

replace(old, new) replaces all occurrences of a specified substring with another.

Example:

```
s = "Python is fun"
# Replaces 'fun' with 'awesome'
print(s.replace("fun", "awesome"))
```

Output

```
ABC

Python is awesome
```

6. title()

Converts the first character of each word to uppercase.

Example:

```
text = "hello world"
print(text.title())
```

Output:

```
Hello World
```

7. capitalize()

Capitalizes the first character of the string.

Example:

```
text = "python programming"
print(text.capitalize())
```

Output:

```
Python programming
```

8. split():

Splits a string into a list using a delimiter (default: space).

Example:

```
text = "Python is fun"
print(text.split())
```

Output:

```
 ['Python', 'is', 'fun']
```

9. join():

Joins elements of a list into a string.

Example:

```
words = ["Python", "is", "fun"]
print(" ".join(words))
```

Output:

```
Python is fun
```

10. count()

Counts occurrences of a substring.

Example:

```
text = "banana"
print(text.count("a"))
```

Output:

```
3
```

11. find()

Finds the first occurrence index of a substring (-1 if not found).

Example:

```
text = "hello world"
print(text.find("world"))
```

Output:

```
6
```

12. startswith()

Checks if a string starts with a given substring.

Example:

```
text = "Python programming"
print(text.startswith("Python"))
```

Output:

```
True
```

13. endswith():

Checks if a string ends with a given substring.

Example:

```
text = "hello.txt"
print(text.endswith(".txt"))
```

Output:

```
True
```

14. isalpha()

Returns True if all characters are letters (no spaces or numbers).

Example:

```
text = "Python"
print(text.isalpha())
```

Output:

```
True
```

15. isdigit()

Returns True if all characters are digits.

Example:

```
num = "12345"
print(num.isdigit())
```

Output:

```
True
```

16. isalnum()

Returns True if all characters are letters or digits.

Example:

```
text = "Python123"
print(text.isalnum())
```

Output:

```
True
```

17. isspace()

Returns True if the string contains only whitespace.

Example:

```
text = "   "
print(text.isspace())
```

Output:

```
True
```

18. islower():

Returns True if all alphabetic characters in the string are lowercase; otherwise, returns False.

Syntax:

```
string.islower()
```

Example:

```
text1 = "hello"
text2 = "Hello123"
print(text1.islower())  # True
print(text2.islower())  # False (contains uppercase 'H')
```

Output:

```
True
False
```

19. isupper():

Returns True if all alphabetic characters in the string are uppercase; otherwise, returns False.

Syntax:

```
string.isupper()
```

Example:

```
text1 = "HELLO"
text2 = "HELLO123"
text3 = "Hello"
print(text1.isupper())  # True
print(text2.isupper())  # True (numbers don't affect the check)
print(text3.isupper())  # False (contains lowercase 'e')
```

Output:

```
True
True
False
```

20. swapcase():

Swaps uppercase and lowercase characters.

Example:

```
text = "Hello World"
print(text.swapcase())
```

Output:

```
hELLO wORLD
```

center(), ljust(), rjust()

These methods are used for aligning strings by adding spaces (or a specified character) to a given width.

21. center(width, fillchar)

Centers the string within a given width by padding with spaces (or a specified character).

Syntax:

```
string.center(width, fillchar)
```

Example:

```
text = "Python"
print(text.center(10))          # Default padding with spaces
print(text.center(10, "-"))     # Padding with "-"
```

Output:

```
  Python
--Python—
```

22. ljust(width, fillchar)

Description: Left-aligns the string within a given width by padding with spaces (or a specified character).

Syntax:

```
string.ljust(width, fillchar)
```

Example:

```
text = "Python"
print(text.ljust(10))          # Default padding with spaces
print(text.ljust(10, "*"))     # Padding with "*"
```

Output:

```
Python
Python****
```

23. rjust(width, fillchar)

Right-aligns the string within a given width by padding with spaces (or a specified character).

Syntax:

```
string.rjust(width, fillchar)
```

Example:

```
text = "Python"
print(text.rjust(10))         # Default padding with spaces
print(text.rjust(10, "."))    # Padding with "."
```

Output:

```
    Python
....Python
```

Concatenating and Repeating Strings:

- We can concatenate strings using + operator and repeat them using * operator.
- Strings can be combined by using **+ operator**.

```
s1 = "Hello"
s2 = "World"
s3 = s1 + " " + s2
print(s3)
```

Output

```
Hello World
```

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

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

Output

```
Hello Hello Hello
```

Formatting Strings:

Python provides several ways to include variables inside strings.

1. String Formatting Operator:

String formatting operator % is unique to strings.

Example:

```
print(“My name is %s and i secured %d marks in python” % (“xyz”,92))
```

Output:

```
My name is xyz and I secured 92 marks in Python
```

2. Using f-strings:

The simplest and most preferred way to format strings is by using f-strings**.**

Example:

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

Output

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

3. Using format():

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

Example:

```
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.
```

Using in for String Membership Testing:

The **in** keyword checks if a particular substring is present in a string.

```
s = "HelloWorld"
print("Hello" in s)
print("Students" in s)
```

Output

```
True

False
```

Special Character and Escape Sequence:

- Statement can be written in single quotes or double quotes but it will raise the SyntaxError as it contains both single and double-quotes.
- Backslash(/) symbol denotes the escape sequence
- Backslash can be followed by a special character and it interpreted differently
- Single quotes inside the string must be escaped.
- We can apply the same as in the double quotes

Example:

‘’’Code will cause a SyntaxError because the string contains both double quotes (") and single quotes (') that conflicts with Python's string syntax’’’

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

# 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?\"")
```

Escape Sequence	Description	Example Output
\'	Single quote	'Hello'
\"	Double quote	"Python"
\\	Backslash	\
\n	Newline (line break)	Hello World
\t	Tab space	Hello World
\r	Carriage return (Moves cursor to the beginning of the line)	World (Overwrites previous text)
\b	Backspace (Deletes one character)	Hell World (Removes 'o')
\f	Form feed (Page break)	Hello (New page in some printers)
\v	Vertical tab	Hello (Adds vertical spacing)
\a	Alert (Beep sound)	(Plays system beep sound)
\0	Null character	(Marks end of string in some languages)
\ooo	Octal value (Character code)	\101 → A
\xhh	Hexadecimal value (Character code)	\x41 → A

Example:

```
# Using various escape sequences
print("He said, \"Python is awesome!\"")  # Double quote inside a string
print('It\'s a beautiful day!')          # Single quote inside a string
print("This is a backslash: \\")          # Printing a backslash
print("Hello\nWorld")                     # Newline
print("Hello\tWorld")                    # Tab space
print("Python\bProgramming")              # Backspace (removes 'n')

# Using hexadecimal and octal values
print("\x48\x65\x6C\x6C\x6F")  # Hex representation of "Hello"
print("\101\102\103")         # Octal representation of "ABC"
```

Output:

```
He said, "Python is awesome!"
It's a beautiful day!
This is a backslash: \
Hello
World
Hello    World
PythoProgramming
Hello
ABC
```

Checking for a Palindrome:

A palindrome is a word, number, or phrase that reads the same forward and backward (e.g., "madam", "121", "racecar").

Method 1: Using String Slicing ([::-1])

```
def is_palindrome(s):
    return s == s[::-1]

word = input("Enter a word: ")
if is_palindrome(word):
    print(f"'{word}' is a palindrome!")
else:
    print(f"'{word}' is not a palindrome.")
```

Output:

```
Enter a word: madam
'madam' is a palindrome!
```

Method 2: Using a Loop:

```
def is_palindrome(s):
    s = s.lower()  # Convert to lowercase for case-insensitivity
    left, right = 0, len(s) - 1
    while left < right:
        if s[left] != s[right]:
            return False
        left += 1
        right -= 1
    return True

word = input("Enter a word: ")
if is_palindrome(word):
    print(f"'{word}' is a palindrome!")
else:
    print(f"'{word}' is not a palindrome.")
```

Output:

```
Enter a word: madam
'madam' is a palindrome!
```

Method 3: Using reversed()

```
def is_palindrome(s):
    return s == "".join(reversed(s))

word = input("Enter a word: ")
print(f"'{word}' is a palindrome!" if is_palindrome(word) else f"'{word}' is not a palindrome.")
```

Output:

```
Enter a word: madam
'madam' is a palindrome!
```

String Modules in Python

- Python provides the string module, which contains useful constants, functions, and classes for handling and manipulating strings efficiently.
- The **Python string module** provides a wide range of **functions** and **constants** related to string manipulation.
- Whether we're building a text-processing application, working with patterns, or cleaning up data, the string module simplifies many common string operations.
- The string module is part of Python’s standard library and is specifically designed for common string operations.
- It provides constants representing commonly used sets of characters (like **lowercase** and **uppercase letters**, **digits**, and **punctuation**) as well as utility functions like **capwords()** for manipulating strings.

Note: The Python String module is a part of the standard Python library, so we do not need to explicitly install it.

The module is useful for:

- Validating characters in a string.
- Formatting output.
- Handling templates in text processing.

We can import the string module by:

```
import string
```

Constants in the Python String Module:

The string module provides several constants that represent predefined sets of characters. These constants are useful for validating, cleaning, or manipulating strings.

- ascii_letters
- ascii_lowercase
- ascii_uppercase
- digits
- hexdigits
- octdigits
- punctuation
- printable
- whitespace

1. ascii_letters

The ascii_letters is a concatenation of all ASCII lowercase and uppercase letters**:**

```
import string
print(string.ascii_letters)
```

Output

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
```

2. ascii_lowercase

The ascii_lowercase contains all **ASCII lowercase letters**:

```
import string
print(string.ascii_lowercase)
```

Output

```
abcdefghijklmnopqrstuvwxyz
```

3. ascii_uppercase

The ascii_uppercase contains all **ASCII uppercase letters**:

```
import string
print(string.ascii_uppercase)
```

Output

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

4. digits

The **digits** contain all decimal digits from **0 to 9**:

```
import string
print(string.digits)
```

Output

```
0123456789
```

5. hexdigits

The hexdigits contains all characters used in **hexadecimal numbers (0-9 and A-F)**:

```
import string
print(string.hexdigits)
```

Output

```
0123456789abcdefABCDEF
```

6. octdigits

The octdigits contains all characters used in **octal numbers (0-7)**:

```
import string
print(string.octdigits)
```

Output

```
01234567
```

7. punctuation

The punctuation contains all characters that are considered **punctuation marks**:

```
import string
print(string.punctuation)
```

Output

```
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

8. printable

The printable contains all characters that are printable, including **digits, letters, punctuation, and whitespace**:

```
import string
print(string.printable)
```

Output

```
0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()*+,
-./:;<=>?@[\]^_`{|}~
```

9. whitespace

The **whitespace** contains all characters that are considered whitespace (**spaces, tabs, newlines, etc.**):

```
import string
print("There are some whitespaces including tab and newline.")
print(string.whitespace)
```

Output

```
There are some whitespaces including tab and newline.


```

Functions in the Python String Module:

Apart from constants, the string module provides several useful functions for string manipulation.

- **capwords()**
- **Formatter()**
- **Template()**

1. capwords()

The **capwords()** function capitalizes the first letter of every word in a string and lowers **the rest of** the letters. It also replaces multiple spaces between words with a single space**.**

```
import string
text = "hello, geek! this is your geeksforgeeks!"
print(string.capwords(text))
```

Output

```
Hello, Geek! This Is Your Geeksforgeeks!
```

2. Formatter()

The **Formatter()** provides an extensible way to create custom string formatting. It works behind the scenes in **Python’s str.format()** method but can be customized to fit our needs.

```
from string import Formatter
fmt = Formatter()
print(fmt.format("Hello, {}!", "geek"))
```

Output

```
Hello, geek!
```

3. Template()

The **Template class** allows us to create string templates where placeholders can be replaced by values. It's useful for situations where we need simple string substitution.

```
from string import Template
t = Template("Hello, $name!")
print(t.substitute(name="geek"))
```

Output

```
Hello, geek!
```

The **Template class** also offers **safe substitution (safe_substitute)** which **will not raise an exception if a placeholder is missing**:

```
from string import Template
t = Template("Hello, $name")
print(t.safe_substitute(name="Python"))
print(t.safe_substitute())
```

Output

```
Hello, Python

Hello, $name
```

Python List:

- A list in Python is used to store the sequence of various types of data.
- Python lists are mutable type its mean we can modify its element after it created.
- However, Python consists of six data- types that are capable to store the sequences, but the most common and reliable type is the list.
- A list can be defined as a collection of values or items of different types.
- The items in the list are separated with the comma (,) and enclosed with the square brackets [].

A list can be defined as below

```
L1 = ["John", 102, "USA"]
L2 = [1, 2, 3, 4, 5, 6]
```

If we try to print the type of L1, L2, and L3 using type() function then it will come out to be a list.

Example:

```
print(type(L1))
print(type(L2))
```

Output:

```
<class 'list'>
<class 'list'>
```

Characteristics of Lists

The list has the following characteristics:

- The lists are ordered.
- The element of the list can be accessed by index.
- The lists are mutable types.
- A list can store the number of various elements.

Example:

```
a = [1,2,"Peter",4.50,"Ricky",5,6]
b = [1,2,5,"Peter",4.50,"Ricky",6]
a == b
```

Output:

```
False
```

Both lists have consisted of the same elements, but the second list changed the index position of the 5th element that violates the order of lists. When compare both lists it returns the false.

Lists maintain the order of the element for the lifetime. That's why it is the ordered collection of objects.

```
a = [1, 2,"Peter", 4.50,"Ricky",5, 6]
b = [1, 2,"Peter", 4.50,"Ricky",5, 6]
a == b
```

Output:

```
True
```

Example:

```
emp = ["John", 102, "USA"]
Dep1 = ["CS",10]
Dep2 = ["IT",11]
HOD_CS = [10,"Mr. Holding"]
HOD_IT = [11, "Mr. Bewon"]

print("printing employee data...")
print("Name : %s, ID: %d, Country: %s"%(emp[0],emp[1],emp[2]))

print("printing departments...")
print("Department 1:\nName: %s, ID: %d\nDepartment 2:\nName: %s, ID: %s"%(Dep1[0],
Dep1[1],Dep2[0],Dep2[1]))


print("HOD Details ..")
print("CS HOD Name: %s, Id: %d"%(HOD_CS[1],HOD_CS[0]))
print("IT HOD Name: %s, Id: %d"%(HOD_IT[1],HOD_IT[0]))

print(type(emp),type(Dep1),type(Dep2),type(HOD_CS),type(HOD_IT))
```

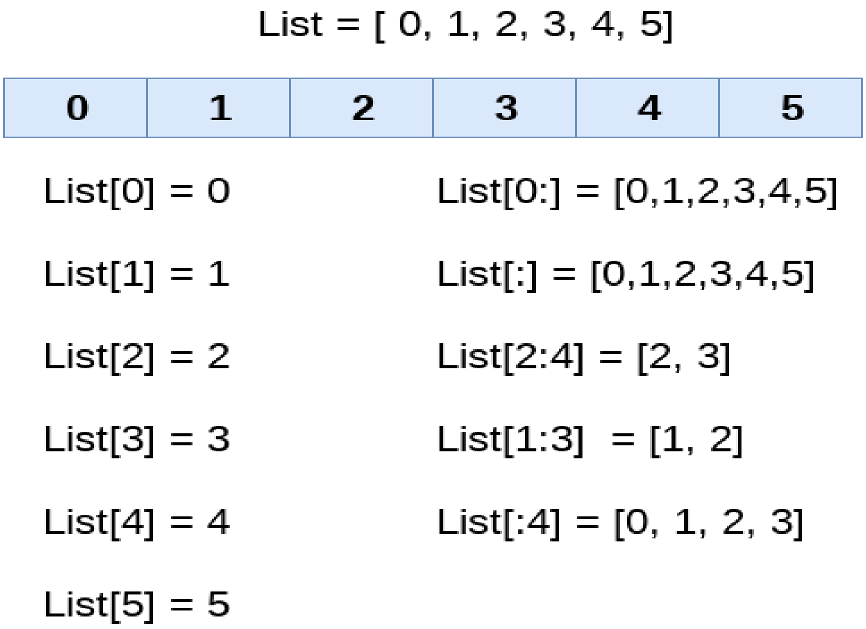
Output:

```
printing employee data...
Name : John, ID: 102, Country:
USA printing departments...
Department 1:
Name: CS, ID: 10
Department 2:
Name: IT, ID: 11
HOD Details ....
CS HOD Name: Mr. Holding,
Id: 10 IT HOD Name: Mr.
Bewon, Id: 11
```

In the above example, we have created the lists which consist of the employee and department details and printed the corresponding details. Observe the above code to understand the concept of the list better.

List indexing and splitting

- The indexing is processed in the same way as it happens with the strings.
- The elements of the list can be accessed by using the slice operator [].
- The index starts from 0 and goes to length - 1.
- The first element of the list is stored at the 0th index, the second element of the list is stored at the 1st index, and so on.



List = [0, 1, 2, 3, 4, 5]

0	1	2	3	4	5

List[0] = 0
List[1] = 1
List[2] = 2
List[3] = 3
List[4] = 4
List[5] = 5

List[0:] = [0,1,2,3,4,5]
List[:] = [0,1,2,3,4,5]
List[2:4] = [2, 3]
List[1:3] = [1, 2]
List[:4] = [0, 1, 2, 3]

We can get the sub-list of the list using the following syntax.

list_varible(start:stop:step)

- The **start** denotes the starting index position of the list.
- The **stop** denotes the last index position of the list.
- The **step** is used to skip the nth element within a **start:stop**

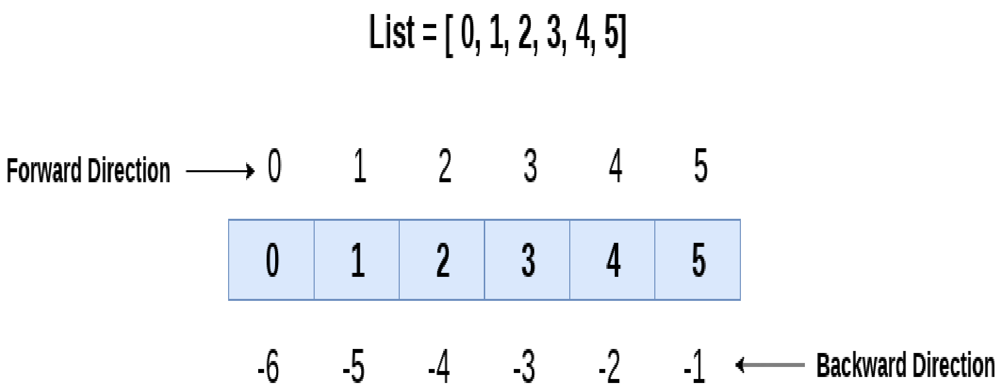
Example:

```
list = [1,2,3,4,5,6,7]
print(list[0])
print(list[1])
print(list[2])
print(list[3])
# Slicing the elements
print(list[0:6])
# By default the index value is 0 so its starts from the 0th element and go for index -1.
print(list[:])
print(list[2:5])
print(list[1:6:2])
```

Output:

```
1
2
3
4
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6, 7]
[3, 4, 5]
[2, 4, 6]
```

- Unlike other languages, Python provides the flexibility to use the negative indexing also.
- The negative indices are counted from the right.
- The last element (rightmost) of the list has the index -1; its adjacent left element is present at the index -2 and so on until the left-most elements are encountered.



Example:

```
list = [1,2,3,4,5]
print(list[-1])
print(list[-3:])
print(list[:-1])
print(list[-3:-1])
```

Output:

```
5
[3, 4, 5]
[1, 2, 3, 4]
[3, 4]
```

As we discussed above, we can get an element by using negative indexing. In the above code, the first print statement returned the rightmost element of the list. The second print statement returned the sub-list, and so on.

Updating List values

- Lists are the most versatile data structures in Python since they are mutable, and their values can be updated by using the slice and assignment operator.
- Python also provides append() and insert() methods, which can be used to add values to the list.

Example to update the values inside the list:

```
list = [1, 2, 3, 4, 5, 6]
print(list)
# It will assign value to the value to the second index
list[2] = 10
print(list)
# Adding multiple-element
list[1:3] = [89, 78]
print(list)
# It will add value at the end of the list
list[-1] = 25
print(list)
```

Output:

```
[1, 2, 3, 4, 5, 6]
[1, 2, 10, 4, 5, 6]
[1, 89, 78, 4, 5, 6]
[1, 89, 78, 4, 5, 25]
```

Appending (or adding) elements in the list:

- Python provides built-in methods to append or add elements to the list.
- We can also append a list into another list.
- These methods are given below.
  - **append(element) -** It appends the value at the end of the list.
  - **insert(index, element) -** It inserts the value at the specified index position.
  - **extends(iterable) -** It extends the list by adding the iterable object.

Example:

1. append(element)

This function is used to add the element at the end of the list.

Example -

```
names = ["Joseph", "Peter", "Cook", "Tim"]
print('Current names List is:', names)
new_name = input("Please enter a name:\n")
names.append(new_name)  # Using the append() function
print('Updated name List is:', names)
```

Output:

```
Current names List is: ['Joseph', 'Peter', 'Cook', 'Tim'] Please enter a name:
Devansh
Updated name List is: ['Joseph', 'Peter', 'Cook', 'Tim', 'Devansh']
```

2. insert(index, element)

- The insert() function adds the elements at the given an index position.
- It is beneficial when we want to insert element at a specific position.

Example:

```
list1 = [10, 20, 30, 40, 50]
print('Current Numbers List: ', list1)
list1.insert(3, 77)
print("The new list is: ",list1)
n = int(input("enter a number to add to list:\n"))
index = int(input('enter the index to add the number:\n'))
list1.insert(index, n)
print('Updated Numbers List:', list1)
```

Output:

```
Current Numbers List:  [10, 20, 30, 40, 50]
The new list is:  [10, 20, 30, 77, 40, 50]
enter a number to add to list:
45
enter the index to add the number:
1
Updated Numbers List: [10, 45, 20, 30, 77, 40, 50]
```

3. extend(iterable)

- The extends() function is used to add the iterable elements to the list.
- It accepts iterable object as an argument.

Example -

```
list1 = [10,20,30]
list1.extend(["52.10", "43.12" ])  # extending list elements
print(list1)
list1.extend((40, 30))  # extending tuple elements
print(list1)
list1.extend("Apple")  # extending string elements
print(list1)
```

Output:

```
[10, 20, 30, '52.10', '43.12']
[10, 20, 30, '52.10', '43.12', 40, 30]
[10, 20, 30, '52.10', '43.12', 40, 30, 'A', 'p', 'p', 'l', 'e']
```

Removing an element from a list:

- Python provides the following methods to remove one or multiple elements.
- We can delete the elements using the **del** keyword by specifying the index position.
  - remove()
  - pop()
  - clear()
  - del

1. The remove() method

- The remove() method is used to remove the specified value from the list.
- It accepts the item value as an argument.

Example -

```
list1 = ['Joseph', 'Bob', 'Charlie', 'Bob', 'Dave']
print("The list is: ", list1)
list1.remove('Joseph')
print("After removing element: ",list1)
```

Output:

```
The list is:  ['Joseph', 'Bob', 'Charlie', 'Bob', 'Dave']
After removing element:  ['Bob', 'Charlie', 'Bob', 'Dave']
```

If the list contains more than one item of the same name, it removes that item's first occurrence.

Example -

```
list1 = ['Joseph', 'Bob', 'Charlie', 'Bob', 'Dave']
print("The list is: ", list1)
list1.remove('Bob')
print("After removing element: ",list1)
```

Output:

```
The list is:  ['Joseph', 'Bob', 'Charlie', 'Bob', 'Dave']
After removing element:  ['Joseph', 'Charlie', 'Bob', 'Dave']
```

2. The pop() method

- The **pop()** method removes the item at the specified index position.
- If we have not specified the index position, then it removes the last item from the list.

Example -

```
list1 = ['Joseph', 'Bob', 'Charlie', 'Bob', 'Dave']
print("The list is: ", list1)
list1.pop(3)
print("After removing element: ",list1)
# index position is omitted
list1.pop()
print("After removing element: ",list1)
```

Output:

```
The list is:  ['Joseph', 'Bob', 'Charlie', 'Bob', 'Dave']
After removing element:  ['Joseph', 'Bob', 'Charlie', 'Dave']
After removing element:  ['Joseph', 'Bob', 'Charlie']
```

We can also specify the negative index position. The index -1 represents the last item of the list.

Example -

```
list1 = ['Joseph', 'Bob', 'Charlie', 'Bob', 'Dave']
print("The list is: ", list1)
# Negative Indexing
list1.pop(-2)
print("After removing element: ",list1)
```

Output:

```
The list is:  ['Joseph', 'Bob', 'Charlie', 'Bob', 'Dave']
After removing element:  ['Joseph', 'Bob', 'Charlie', 'Dave']
```

3. The clear() method

The clear() method removes all items from the list. It returns the empty list. Let's understand the following example.

Example -

```
list1 = [10, 20, 30, 40, 50, 60]
print(list1)
# It will return the empty list
list1.clear()
print(list1)
```

Output:

```
[10, 20, 30, 40, 50, 60]
[]
```

4. The del statement

We can remove the list item using the del keyword. It deletes the specified index item. Let's understand the following example.

Example -

```
list1 = [10, 20, 30, 40, 50, 60]
print(list1)
del list1[5]
print(list1)
del list1[-1]
print(list1)
```

Output:

```
[10, 20, 30, 40, 50, 60]
[10, 20, 30, 40, 50]
[10, 20, 30, 40]
```

It can delete the entire list.

```
del list1
print(list1)
```

We can also delete the multiple items from the list using the **del** with the slice operator.

Example -

```
list1 = [10, 20, 30, 40, 50, 60]
print(list1)
del list1[1:3]
print(list1)
del list1[-4:-1]
print(list1)
del list1[:]
print(list1)
```

Output:

```
[10, 20, 30, 40, 50, 60]
[10, 40, 50, 60]
[60]
[]
```

Python List Operations:

The concatenation (+) and repetition (*) operators work in the same way as they were working with the strings.

Consider a Lists l1 = [1, 2, 3, 4], **and** l2 = [5, 6, 7, 8] to perform operation.

Operator	Description	Example
Repetition	The repetition operator enables the list elements to be repeated multiple times.	L1*2 = [1, 2, 3, 4,1, 2, 3, 4]
Concatenation	It concatenates the list mentioned on either side of the operator.	l1+l2 = [1, 2, 3, 4,5, 6, 7, 8]
Membership	It returns true if a particular item exists in a particular list otherwise false.	print(2 in l1) => prints True.
Iteration	The for loop is used to iterate over the list elements.	for i in l1: print(i) **Output** 1 2 3 4
Length	It is used to get the length of the list	len(l1) = 4

Iterating a List:

- A list can be iterated by using a for - in loop.

Example:

```
list = ["John", "David", "James", "Jonathan"]

for i in list:
 # The i variable will iterate over the elements of the List and contains each element in each it eration.
     print(i)
```

Output:

```
John
David
James
Jonathan
```

Adding elements to the list:

- Python provides append() function which is used to add an element to the list.
- However, the append() function can only add value to the end of the list.

Example:

```
#Declaring the empty list
l =[]
#Number of elements will be entered by the user
n = int(input("Enter the number of elements in the list:"))

# for loop to take the input
for i in range(0,n):
# The input is taken from the user and added to the list as the item
      l.append(input("Enter the item:"))

print("printing the list items..")
# traversal loop to print the list items
for i in l:
       print(i, end = "  ")
```

Output:

```
Enter the number of elements in the list:5
Enter the item:25
Enter  the  item:46
Enter  the  item:12
Enter  the  item:75
Enter the item:42
printing the list items
25   46   12   75   42
```

Removing elements from the list:

Python provides the **remove()** function which is used to remove the element from the list.

Example -

```
list = [0,1,2,3,4]
print("printing original list: ");
for i in list:
    print(i,end=" ")
list.remove(2)
print("\nprinting the list after the removal of first element...")
for i in list:
    print(i,end=" ")
```

Output:

```
printing original list:
0 1 2 3 4
printing the list after the removal of first element...
0 1 3 4
```

Python List Built-in functions:

S.No.	Function	Description	Example
1	cmp(list1, list2)	It compares the elements of both the lists.	This method is not used in the Python 3 and the above versions.

2	len(list)	It is used to calculate the length of the list.	L1 = [1,2,3,4,5,6,7,8]
			print(len(L1))
			Output: 8

3	max(list)	It returns the maximum element of the list.	L1 = [12,34,26,48,72]
			print(max(L1))
			Output: 72

4	min(list)	It returns the minimum element of the list.	L1 = [12,34,26,48,72]
			print(min(L1))
			Output: 12

5	list(seq)	It converts any sequence to the list.	str = "Johnson"
			s = list(str)
			print(type(s))
			Output: <class list>

Example: 1- Write the program to remove the duplicate element of the list.

```
list1 = [1,2,2,3,55,98,65,65,13,29]
# Declare an empty list that will store unique values
list2 = []
for i in list1:
    if i not in list2:
        list2.append(i)
print(list2)
```

Output:

```
[1, 2, 3, 55, 98, 65, 13, 29]
```

Example:2- Write a program to find the sum of the element in the list.

```
list1 = [3,4,5,9,10,12,24]
sum = 0
for i in list1:
    sum = sum+i
print("The sum is:",sum)
```

Output:

```
The sum is: 67
```

Example: 3- Write the program to find the lists consist of at least one common element.

```
list1 = [1,2,3,4,5,6]
list2 = [7,8,9,2,10]
for x in list1:
    for y in list2:
        if x == y:
            print("The common element is:",x)
```

Output:

```
The common element is: 2
```

Python List Comprehension:

- List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.
- List Comprehension is defined as an elegant way to define, create a list in Python and consists of brackets that contain an expression followed by **for** clause.
- It is efficient in both computationally and in terms of coding space and time.

Signature:

The list comprehension starts with **'['** and **']'**.

[expression **for** item **in** list **if** conditional]

Example:

```
letters = []
for letter in 'Python':
        letters.append(letter)
print(letters)
```

Output:

```
['P', 'y', 't', 'h', 'o', 'n']
```

Example:

```
letters = [ letter for letter in 'Python' ]
print( letters)
```

Output:

```
['P', 'y', 't', 'h', 'o', 'n']
```

Python Tuple:

- Python Tuple is used to store the sequence of immutable Python objects.
- The tuple is similar to lists since the value of the items stored in the list can be changed, whereas the tuple is immutable, and the value of the items stored in the tuple cannot be changed.

Creating a tuple:
A tuple can be written as the collection of comma-separated (,) values enclosed with the small () brackets.

Example:

```
T1 = (101, "Peter", 22)
T2 = ("Apple", "Banana", "Orange")
T3 = 10,20,30,40,50


print(type(T1))
print(type(T2))
print(type(T3))
```

Output:

```
<class 'tuple'>
<class 'tuple'>
<class 'tuple'>
```

Note: The tuple which is created without using parentheses is also known as tuple packing.

An empty tuple can be created as follows.

```
T4 = ()
```

- Creating a tuple with single element is slightly different.
- We will need to put comma after the element to declare the tuple.

Example:

```
tup1 = ("JavaTpoint")
print(type(tup1))
#Creating a tuple with single element
tup2 = ("JavaTpoint",)
print(type(tup2))
```

Output:

```
<class 'str'>
<class 'tuple'>
```

Accessing elements of a tuple:

- A tuple is indexed in the same way as the lists.
- The items in the tuple can be accessed by using their specific index value.

Example - 1

```
tuple1 = (10, 20, 30, 40, 50, 60)
print(tuple1)
count = 0
for i in tuple1:
    print("tuple1[%d] = %d"%(count, i))
    count = count+1
```

Output:

```
(10, 20, 30, 40, 50, 60)
tuple1[0] = 10
tuple1[1] = 20
tuple1[2] = 30
tuple1[3] = 40
tuple1[4] = 50
tuple1[5] = 60
```

Example - 2

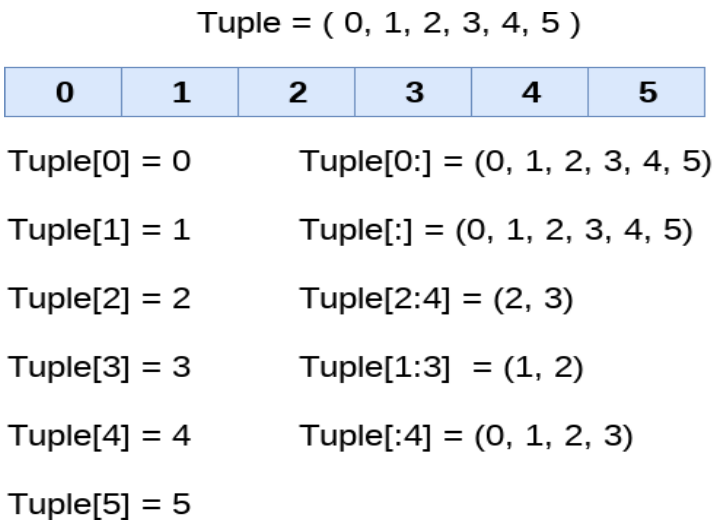
```
tuple1 = tuple(input("Enter the tuple elements ..."))
print(tuple1)
count = 0
for i in tuple1:
    print("tuple1[%d] = %s"%(count, i))
    count = count+1
```

Output:

```
Enter the tuple elements ...123456
('1', '2', '3', '4', '5', '6')
tuple1[0] = 1
tuple1[1] = 2
tuple1[2] = 3
tuple1[3] = 4
tuple1[4] = 5
tuple1[5] = 6
```

Tuple indexing and slicing:

- The indexing and slicing in the tuple are similar to lists.
- The indexing in the tuple starts from 0 and goes to length(tuple) - 1.
- The items in the tuple can be accessed by using the index [] operator.
- Python also allows us to use the colon operator to access multiple items in the tuple.



Tuple = (0, 1, 2, 3, 4, 5)

0	1	2	3	4	5

Tuple[0] = 0 Tuple[0:] = (0, 1, 2, 3, 4, 5)

Tuple[1] = 1 Tuple[:] = (0, 1, 2, 3, 4, 5)

Tuple[2] = 2 Tuple[2:4] = (2, 3)

Tuple[3] = 3 Tuple[1:3] = (1, 2)

Tuple[4] = 4 Tuple[:4] = (0, 1, 2, 3)

Tuple[5] = 5

Example:

```
 tup = (1,2,3,4,5,6,7)
 print(tup[0])
 print(tup[1])
 print(tup[2])
 # It will give the IndexError
 print(tup[8])
```

Output:

```
1
2
3
tuple index out of range
```

In the above code, the tuple has 7 elements which denote 0 to 6.
We tried to access an element outside of tuple that raised an **IndexError**.

```
tuple = (1,2,3,4,5,6,7)
#element 1 to end
print(tuple[1:])
#element 0 to 3 element
print(tuple[:4])
#element 1 to 4 element
print(tuple[1:5])
# element 0 to 6 and take step of 2
print(tuple[0:6:2])
```

Output:

```
(2, 3, 4, 5, 6, 7)
(1, 2, 3, 4)
(1, 2, 3, 4)
(1, 3, 5)
```

Negative Indexing:

- The tuple element can also access by using negative indexing.
- The index of -1 denotes the rightmost element and -2 to the second last item and so on.
- The elements from left to right are traversed using the negative indexing.

Example:

```
tuple1 = (1, 2, 3, 4, 5)
print(tuple1[-1])
print(tuple1[-4])
print(tuple1[-3:-1])
print(tuple1[:-1])
print(tuple1[-2:])
```

Output:

```
5
2
(3, 4)
(1, 2, 3, 4)
(4, 5)
```

Deleting Tuple:

- Unlike lists, the tuple items cannot be deleted by using the **del** keyword as tuples are immutable.
- To delete an entire tuple, we can use the **del** keyword with the tuple name.

Example:

```
tuple1 = (1, 2, 3, 4, 5, 6)
print(tuple1)
del tuple1[0]
print(tuple1)
del tuple1
print(tuple1)
```

Output:

```
(1, 2, 3, 4, 5, 6)
Traceback (most recent call last):
  File "tuple.py", line 4, in <module>
    print(tuple1)
NameError: name 'tuple1' is not defined
```

Basic Tuple operations:

The operators like concatenation (+), repetition (*), Membership (in) works in the same way as they work with the list.

Let's say Tuple t = (1, 2, 3, 4, 5) and Tuple t1 = (6, 7, 8, 9) are declared.

Operator	Description	Example
Repetition	The repetition operator enables the tuple elements to be repeated multiple times.	T1*2 = (1, 2, 3, 4,5, 1, 2, 3, 4, 5)
Concatenation	It concatenates the tuple mentioned on either side of the operator.	T1+T2 = (1, 2, 3, 4,5, 6, 7, 8, 9)
Membership	It returns true if a particular item exists in the tuple otherwise false	print (2 in T1) **Output:** True.
Iteration	The for loop is used to iterate over the tuple elements.	for i in T1: print(i) **Output** 1 2 3 4 5
Length	It is used to get the length of the tuple.	len(T1) = 5

Python Tuple inbuilt functions:

S.No.	Function	Description	Example
1	cmp(tuple1, tuple2)	It compares two tuples and returns true if tuple1 is greater than tuple2 otherwise false.	The cmp() function was used in Python 2 to compare two values, including tuples. In Python 3, cmp() was removed.

2	len(tuple)	It calculates the length of the tuple.	my_tuple = (10, 20, 30, 40)
			print(len(my_tuple)) # **Output: 4**
			
			empty = ()
			print(len(empty)) # **Output: 0**

3	max(tuple)	It returns the maximum element of the tuple	nums = (3, 7, 2, 9, 5) print(max(nums)) # **Output: 9**
4	min(tuple)	It returns the minimum element of the tuple.	numbers = (8, 3, 5, 1, 9) print(min(numbers)) # **Output: 1**

5	tuple(seq)	It converts the specified sequence to the tuple.	my_list = [1, 2, 3]
			my_tuple = tuple(my_list)
			print(my_tuple)
			Output: (1, 2, 3)

Uses of tuple:

Using tuple instead of list in the following scenario:

- Using tuple instead of list gives us a clear idea that tuple data is constant and must not be changed.
- Tuple can simulate a dictionary without keys. Consider the following nested structure, which can be used as a dictionary.
  [(101, "John", 22), (102, "Mike", 28), (103, "Dustin", 30)]

List vs. Tuple:

S. No.	List	Tuple
1	The literal syntax of list is shown by the [].	The literal syntax of the tuple is shown by the ().
2	The List is mutable.	The tuple is immutable.
3	The List has a variable length.	The tuple has the fixed length.
4	The list provides more functionality than a tuple.	The tuple provides less functionality than the list.
5	The list is used in the scenario in which we need to store the simple collections with no constraints where the value of the items can be changed.	The tuple is used in the cases where we need to store the read-only collections i.e., the value of the items cannot be changed. It can be used as the key inside the dictionary.
6	The lists are less memory efficient than a tuple.	The tuples are more memory efficient because of its immutability.

Tuple Assignment:

- Tuple assignment is a way to assign multiple variables at once using a tuple.

Example:

```
a, b = (1, 2)
print(a)  # 1
print(b)  # 2
```

This is shorthand for:

```
t = (1, 2)
a = t[0]
b = t[1]
```

- We can also omit the parentheses:

Example:

```
a, b = 1, 2
```

Immutability vs Reassignment:

- Tuples are immutable: once created, we cannot change the contents of a tuple.

Example:

```
t = (1, 2, 3)
t[0] = 10  # This will raise a TypeError
```

- Reassignment is allowed: We can reassign the variable to point to a new tuple.

Example:

```
t = (1, 2, 3)
t = (4, 5, 6)  #  This is valid
```

- **Immutability refers to the contents of the tuple, not the variable name.**

Tuple packing:

- The tuple which is created without using parentheses is also known as tuple packing.

Example:

```
t = 10, 20, 30
print(type(t))  #<class 'tuple'>
```

Unpacking Tuples

- Tuple unpacking lets us assign elements of a tuple to multiple variables in one line:

Example:

```
t = (10, 20, 30)
x, y, z = t
print(x, y, z)  # 10 20 30
```

- We can also use a * to collect excess values:

Example:

```
a, *b = (1, 2, 3, 4)
print(a)  # 1
print(b)  # [2, 3, 4]
```