Chapter- 2:

PYTHON REVISION TOUR:

STRING MANIPULATION:

Introduction:

Python strings are characters enclosed in quotes of any type – single quotation marks, double quotation marks and triple quotation marks. An empty string is a string that has 0 characters (i.e., it is just a pair of quotation marks) and that Python strings are immutable.

Strings are sequence of characters, where each character has a unique position –id/index. The indexes of a string begin from 0 to (length – 1) in forward direction and -1, -2, -3, ..., -length in backward direction.

Traversing A String:

Individual characters of a string are accessible through in unique index of each character. Using the indexes, you can traverse a string character by character. Traversing refers to iterating through the elements of a string, one character at a time. To traverse through a string, you can write a loop like:

```
name = ‘superb’
for ch in name:
        print(ch, ’-‘, end = ’  ‘ )
```

The above code will print

```
s-u-p-e-r-b
```

Program to read a string and display it in reverse order – display one character per line. Do not create a reverse string, just display in reverse order

```
string1 = input(“Enter a string:”)
print (“The”, string1, “in reverse order is”)
length = len(string1)
for a in range (-1, (-length-1),-1) :
        print (string1[a])
```

Sample run of above program is:

```
Enter a string: python
The python in reverse order is:
n
```

o

h

t

y

p

String Operations:

The basic operators + and *, membership operators in and not in and comparisons operators (all relational operators) are used for strings.

String Concatenation Operator +:

The + operator creates a new string by joining the two operand strings, e.g.,

“tea” + “pot”

Will result into

teapot

Consider some more examples:

Expression	**Will result into**
‘l’ +’l’	‘11’
“a” + “0”	‘a0’
‘123’ + ‘abc’	‘123abc’

Caution!

Another important thing that you need to know about + operator is that this operator can work with numbers and strings separately for addition and concatenation respectively, but in the same expression, you cannot combine numbers and strings as operands with a + operator.

For example,

```
2 + 3 = 5            # addition – VALID
‘2’ + ‘3’ = ‘23’     # concatenation – VALID
```

But the expression

```
‘2’ + 3
```

Is invalid. It will produce an error like :

```
>>> ‘2’ + 3
Traceback (most recent call last) :
    File “<pyshell#2>”, line 1, in <module>
```

'2' + 3

Thus, we can summarize + operator as follows:

Table: Working of Python + Operator:

Operands' data type	Operation performed by +	Example
numbers	addition	9 + 9 = 18
string	concatenation	"9"+"9" = "99"

String Replication Operator *:

The * operator when used with numbers (i.e., when both operands are numbers), it performs multiplication and returns the product of the two number operands.

To use a * operator with strings, you need two types of operands – a string and a number, i.e., as number * string or string * number.

Where string operand tells the string to be replicated and number operand tells the number of times, it is to be repeated; Python will create a new string that is a number of repetitions of the string operand.

For example,

3 * "go!"

Will return

'go!go!go!'

Consider some more examples:

Expression	will result into
"abc" * 2	"abcabc"
5*"@"	"@@@@@"
":-" * 4	":-:-:-:-"
"1" * 2	"11"

Caution!

Another important thing that you need to know about * operator is that this operator can work with numbers as both operands for multiplication and with a string and a number for replication respectively, but in the same expression. You cannot have string as both the operands with a* operator.

For example:

2 * 3 = 6	# multiplication – VALID
"2"*3 = "222"	#replication – VALID

But the expression

“2” * “3”

Is invalid. It will produce an error like:

```
>>>”2”*”3”
Traceback(most recent call last) :
  File “<pyshell#0>”, line 1, in <module>
    “2”*”3”
Type Error: can’t multiply sequence by non-int of type ‘str’
```

Thus, we can summarize + operator as follows:

Working of Python * operator:

Operands’ data type	**Operation performed by***	**Example**
Numbers	Multiplication	9 *2 = 18
string, number	Replication	“#” * 3 = “###”
number, string	Replication	3 * “#” = “###”

Quick Interesting Facts:

It’s possible to access individual characters of a string by using array-like indexing syntax. We can access each and every element of string through their index number and the indexing starts from 0

Membership Operators:

There are two membership operators for strings (in fact for all sequence types). These are **in** and **not in**.

in: Returns *True* if a character or a substring exists in the given string; *False* otherwise

not in: Returns *True* if a character or a substring does not exist in the given string; *False* otherwise

Example:

“12” in “xyz”

“12” not in “xyz”

Now, let's have a look at some examples:

"a"	**in** "heya"	will give	**True**
"jap"	**in** "heya"	will give	**False**
"jap"	**in** "japan"	will give	**True**
"jap"	**in** "Japan"	will give	**False** because j letter's cases are different; hence "jap" is not contained in "Japan"
"jap"	**not in** "Japan"	will give	**True** because string "jap" is not contained in string "Japan"
"123"	**not in** "hello"	will give	**True** because string "123" is not contained in string "hello"
"123"	**not in** "12345"	will give	**False** because "123" is contained in string "12345"

Comparison Operators:

Python's standard comparison operators *i.e.,* all relational operators (<, <=, >, >=, ==,!=) apply to strings also. The comparisons using these operators are based on the standard character-by-character comparison rules for Unicode.

Thus, you can make out that

"a" == "a"	will give	*True*
"abc" == "abc"	will give	*True*
"a" != "abc"	will give	*True*
"A" != "a"	will give	*True*
"ABC" == "abc"	will give	False (letters' case is different)
"abc" != "Abc"	will give	*True* (letter's case is different)

Common Characters and their Ordinal Values:

Characters	**Ordinal Values**
'0' to '9'	48 to 57
'A' to 'Z'	65 to 90
'a' to 'z'	97 to 122

Thus, upper-case letters are considered smaller than the lower-case letters. For instance,

'a' **<** 'A'	will give	*False* because the Unicode value of lower-case letters is higher than upper case letters; hence 'a' is greater than 'A', not lesser.
'ABC' **>** 'AB'	will give	*True* for obvious reasons.
'abc' **<=** 'ABCD'	will give	*False* because letters of 'abc' have higher ASCII values compared to 'ABCD'.
'abcd' **>** 'abcD'	will give	*Ture* because strings 'abcd' and 'abcD' are same till first three letters but the last letter of 'abcD' has lower ASCII value than last letter of string 'abcd'.

Quick Interesting Facts:

Python can implement the 'else' clause within 'for' and 'while' loop

String Slices:

'Slice', which means- 'a part of'. In the same way, in Python, the term *'string slice'* refers to a part of the string, where strings are sliced using a range of indices.

That is, for a string say **name,** if we give **name[n:m]** where ***n*** and ***m*** are integers and legal indices, Python will return a slice of the string by returning the characters falling between indices ***n*** and ***m*** – starting at *n, n+1, n+2* ... till *m-1.* Let us understand this with the help of examples. Say we have a string namely ***word*** storing a string ***'amazing'*** *i.e.,*

	0	***1***	***2***	***3***	***4***	***5***	***6***
word	a	m	a	z	i	n	g
	-7	***-6***	***-5***	***-4***	***-3***	***-2***	***-1***

Then,

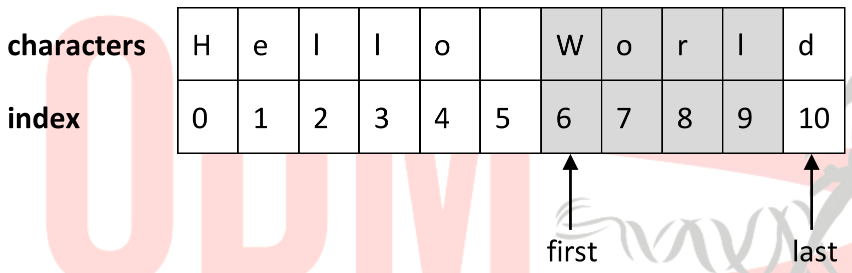
word[0:7]	will give	**'amazing'**	(the letters starting from index 0 going up till 7-1 *i.e.,* 6: from indices 0 to 6, both inclusive)
word[0:3]	will give	**'ama'**	(letters from index 0 to 3 – 1 *i.e.,* 0 to 2)
word[2:5]	will give	**'azi'**	(letters from index 2 to 4 (*i.e.,* 5-1))
word[-7:-3]	will give	**'amaz'**	(letters from indices-7, -6, -5, -4 excluding index -3)

word[-5:-1]	will give	**'azin'**	(letters from indices -5, -4, -3, -2 excluding -1)
word[:7]	will give	**'amazing**	(missing index before colon is taken as 0 (zero))
word[:5]	will give	**'amazi'**	(-do-)
word[3:]	will give	**'zing'**	(missing index after colon is taken as 7 (the length of the string))
word[5:]	will give	**'ng'**	(-do-)

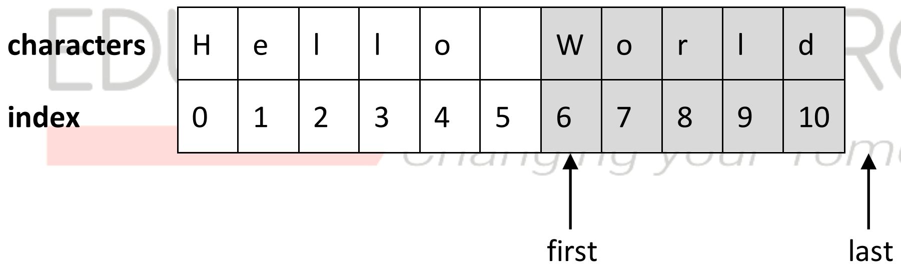
The string slice refers to a part of the string **s[start:end]** that is the elements beginning at **start** and extending upto but not including **end**.

Following figure (Fig. 2.1) shows some string slices:

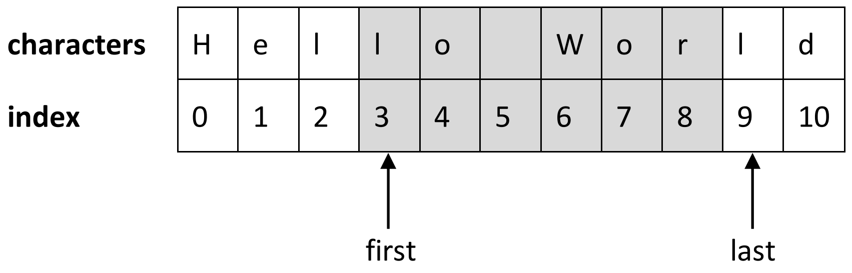
helloString[6:10]



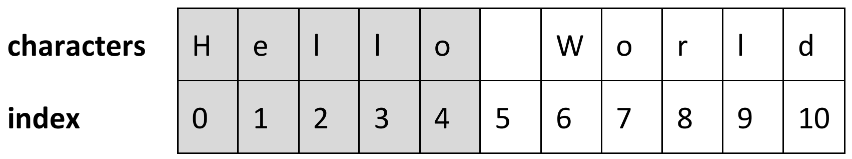
helloString[6:]



helloString[3:-2]



helloString[:5]



Quick Interesting Facts:

To extract substring from the whole string then we use the syntax like

string_name [beginning: end: step]

- beginning represents the starting index of string
- end denotes the end index of string which is not inclusive
- step denotes the distance between the two words.

Python's Built-in String Manipulation Methods:

String.capitalize( ):

Returns a copy of the string with its first character capitalized.

Example:

```
>>> "true".capitalize ( )

True
```

String.find (sub[, end]):

Returns the lowest index in the string where the substring sub is found within the slice range of start and end. Returns 1 if sub is not found .

Example

```
>>>string= :it goes as – ringa ringa Roses"

>>> sub = ringa

>>> string .find(sub)

13
```

String.isalnum ( ):

Returns **True** if the characters in the string are alphanumeric (alphabets or numbers) and there is at least one character, **False** otherwise.

Example:

```
>>> string = “abc123”
>>> string.isalnum( )
True
```

String.isalpha( ):

Returns True if all characters in the string are alphabetic and there is at least one character, False otherwise.

Example (considering the same string values as used in example of previous function – isalnum)

```
>>> string.isalpha( )
False
>>> string2. Isalpha( )
True
```

String.isdigit( ):

Returns True if all the characters in the string are digits. There must be at least one character, otherwise it returns False.

Example:

(Considering the same string values as used in example of previous function – isalnum)

```
>>> string.isdigit( )
False
```

String.islower( ):

Returns True if all cased characters in the string are lowercase. There must be at least one cased character. It returns False otherwise.

Example:

```
>>> string = ‘hello’
>>> string.islower( )
True
```

String.isspace( ):

Returns True if there are only whitespace characters in the string. There must be at least one character. It returns False otherwise.

Example:

```
>>> string = "   "                  # stores three spaces

>>> string.isspace( )

True
```

String.isupper( ):

Tests whether all cased characters in the string are uppercase and requires that there be at least one cased character. Returns True if so and False otherwise.

Example

```
>>> string = "HELLO"

>>> string.isupper( )

True
```

string.lower( ):

Returns a copy of the string converted to lowercase.

Example:

```
>>> string.lower( )

'hello'
```

string.upper( ):

Returns a copy of the string converted to uppercase.

Example:

```
>>> string.upper( )

'HELLO'
```

LISTS MANIPULATION:

Introduction:

The Python lists are containers that are used to store a list of values of any type. Unlike other variables Python lists are mutable i.e, you can change the elements of a list in place; Python will not create a fresh list when you make changes to an element of a list. List is a type of sequence like strings and types but it differs from them in the way that lists are mutable but strings and tuples are immutable.

Creating and Accessing lists:

A list is a standard data type of Python that can stone a sequence of values belonging to any type. The lists are depicted through square brackets, e.g following are some lists in Python.

```
[ ]                          # list with no member, empty list
[1, 2, 3]                    # list of integers
[1, 2, 5, 3, 7, 9]           # list of numbers (integers and floating point)
['a', 'b', 'c']              # list of characters
['a', 1, 'b', 3.5, 'zero']   # list of mixed value types
['One', 'Two', 'Three']      # list of strings
```

List Operations:

Membership operators: - Both 'in' and 'not in' operators work on Lists just like they work for other sequences. This is, in tells if an element is present in the list or not, and not in does the opposite.

Concatenation and replication operators + and * :- The + operator adds one list to the end of another. The * operator repeats a list. We shall be talking about these two operations in a later section 11.3 - List Operations.

Accessing Individual Elements:- As mentioned, the individual elements of a list are accessed through their indexes. Consider following examples:

```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
>>> vowels [0]
'a'
>>> vowels[4]
'u'
>>> vowels [-1]
```

```
'u'
>>> vowels [-5]
'a'
```

Traversing a list:

The for loop makes it easy to traverse or loop over the items in a list, as per following syntax:

```
for <item> in < List> :
        Process each item here
```

For example, following loop shows each item of a list L in separate lines:

```
L = ['p', 'y', 't', 'h', 'o', 'n']
for a n L :
        print (a)
```

The above loop will produce result as:

```
p
y
t
h
o
n
```

Comparing Lists:

You can compare two lists using standard comparison operation operators of Python, i,e, <, >, ==, !, =, etc. Python internally compares individual elements of lists (and tuples) in lexicographical order. This means that to compare equal, each corresponding element must compare equal and the two sequences must be of the same type i.e, having comparable types of values.

Consider following examples:

```
>>> L1, L2 = [1, 2, 3], [1, 2, 3]
>>> L3 = [1, [2, 3]]
>>> L1 == L2
True
>>> L1 == L3
False
```

For comparison operators <, <, >=, <=, the corresponding elements of two lists must be of comparable types, otherwise Python will give error.

Consider the following considering the above two lists:

```
>>> L1 < L2
False
>>> L1 < L3
Traceback (most recent call last) :
File "<ipython-input - 180-84 fdf598c3f1>", line 1, in <module>
L1 < L3
TypeError : '<' not supported between instances of 'int' and 'list'
```

Comparison	Result	Reason
[1, 2, 8, 9] < [9, 1]	True	Gets the result with the comparison of corresponding first elements of two lists 1 < 9 is True
[1, 2, 8, 9] < [1, 2, 9, 1]	True	Gets the result with the comparison of corresponding third elements of two lists 8 < 9 is True
[1, 2, 8, 9] < [1, 2, 9, 10]	True	Gets the result with the comparison of corresponding third elements of two lists 8 < 9 is True
[1, 2, 8, 9] < [1, 2, 8, 4]	False	Gets the result with the comparison of corresponding fourth elements of two lists 9 < 4 is False

List Operations:

Joining Lists:

Joining two lists is very easy just like you perform addition, literally ;-). The concatenation operator +, when used with two lists, joins two lists. Consider the example given below.

```
>>> 1st1 = [1, 3, 5]
>>> 1st 2 = [6, 7, 8]
>>> 1st 1 + 1st 2
[1, 3, 5, 6, 7, 8]
```

As you can see that the resultant list has firstly elements of first list lst1 and followed by elements of second list lst2. You can also join two or more lists to form a new list, e.g.

```
>>> 1st 1 = [10, 12, 14]
>>> 1st 2 = [20, 22, 24]
>>> 1st 3 = [20, 32, 34]
>>> 1st = 1st 1 + 1st 2 + 1st 3
>>> 1st
```

```
[10, 12, 14, 20, 22, 24, 30, 32, 32]
```

Quick Interesting Facts:

- Lists index their elements just like strings, i.e two-way indexing.
- Lists are stored in memory exactly like strings, except that because some of their objects are larger than others, they store a reference at each index instead of single character as in strings.

LIST OPERATIONS:

Slicing the Lists:

List slices, like string slices are the sub part of a list extracted out. You can use indexes of list elements to create list slices as per following format:

seq = L [start : stop]

The above statement will create a list slice namely seq having elements of list L on indexes start, start+1, start+2, ..., stop -1.

```
>>> 1st = [10, 12, 14, 20, 22, 24, 30, 32, 34]
>>> seq = 1st [3 : -3]
>>> seq
[20, 22, 24]
>>> seq [1] = 28
>>> seq
[20, 28, 24]
>>> 1st = [10, 12, 14, 20, 22, 24, 30, 32, 34]
>>> 1st [3 : 30]
[20, 22, 24, 30, 32, 34]
>>> 1st [-15 : 7]
[10, 12, 14, 20, 22, 24, 30]
>>> L1 [2, 3, 4, 5, 6, 7, 8]
>>> L1 [2 : 5]
[4, 5, 6]
>>> L1 [6 : 10]
[8]
```

```
>>> L1 [10 : 20]
[ ]
```

Lists also support slice steps.

seq = L [start : stop : step]

Consider some examples to understand this.

```
>>> 1st
[10, 12, 14, 20, 22, 24, 30, 32, 34]
>>> 1st [0 : 10 : 2]
[10, 14, 22, 30, 34]
>>> 1st [2 : 10 : 3]
[14, 24, 34]
>>> 1st [: : 3]
[10, 20, 30]
```

List Functions and Methods:

1. The index() method:

This function returns the index of first matched item from the list. It is used as per following format:

List. index (<item>)

For example, for a list L1 = [13, 18, 11, 16, 18, 14]

```
>>> L1.index (18)
1
```

However, if the given item is not in the list, it raises exception value Error

2. The append() method:

append () method adds an item to the end of the list. It works as per following syntax:

List.append (<item>)

Takes exactly one element and returns no value

For example, to add a new item "yellow" to a list containing colours, you may write:

```
>>> colours = ['red', 'green', 'blue']
>>> colours . append ('yellow')
>>> colours
['red', 'green', 'blue', 'yellow']
```

The append() does not return the new list, just modifies the original.

3. The extend() method:

The extend () method of is also used for adding multiple elements (given in the form of a list) to a list.

List. extend (<list>)

Takes exactly one element (a list type) and returns no value

That is extend () takes a list as an argument and appends all of the elements of the argument list to the list object on which extend() is applied. Consider following example:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
>>> t2
['d', 'e']
```

The above example left list t2 unmodified. Like append (), extend () also does not returned any value. Consider following example:

```
>>> t3 = t1 . extend (t2)
>>> t3
```

4. The insert() method: -

The insert () method is also an insertion method for lists, like append and extend methods. However, both append () and extend () insert the element (s) at the end of the list. If you want to insert an element somewhere in between or any position of your choice are of no use. For such a requirement insert () is used. The insert () function inserts an item at a given position.

List.insert (<pos>, <item>)

```
>>> t1 = ['a', 'e', 'u']
>>>t1 . insert (2, 'i')          # inset element 'i' at index 2.
>>> t1
['a', 'e', 'i', 'u']
```

5. The pop() method:

You have read about this method earlier. The pop () is used to remove the item from the list. It is used as per following syntax:

List. pop (<index>) # <index is optional argument

```
>>> t1
['k', 'a', 'e', 'i', 'p', 'q', 'u']
>>> ele1 = t1.pop (0)
>>> ele1
'k'
>>> t1
['a', 'e', 'i', 'p', 'q', 'u']
>>> t1
['a', 'e', 'i', 'p', 'q']
```

6. **The remove() method:** index or position in the list? Well, Python though it in advance and made available the remove () method. The remove () method removes the first occurrence of given item from the list. It is used as per following format:

   List. remove (<value>)- Takes one essential argument and does not return anything.

   The remove () will report an error if there is no such item in the list. Consider some examples:

```
>>> t1 = ['a', 'e', 'i', 'p', 'q', 'a', 'q', 'p']
>>> t1.remove ('a')
>>> t1
['e', 'i', 'p', 'a', 'q', 'p']
>>> t1.remove ('p')
>>> t1
['e', 'i', 'q', 'a', 'q', 'p']
>>> t1.remove ('ak')
Traceback (most recent call last) :
File "<physhell #45>", line 1, in <module>
t1.remove ('k')
ValueError : list. remove (x) : x not in list
```

7. **The clear() method**: - This method removes all the items from the list and the list becomes empty list after this function. This function returns nothing. It us used as per following format.

   List. clear ()

   For instance, if you have a list L1 as

```
>>> L1 = [2, 3, 4, 5]
>>> L1.clear ()
>>> L1
[ ]
```

Unlike del <1stname)> statement, clear () removes only the elements and not the list element. After clear (), the list object still exists as an empty list.

8. **The count() method**: This function returns the count of the item that you passed as argument. If the given item is not in the list, it returns zero. It is used as per following format:

   List. count (<item>)

   For instance, for a list L1 = [13, 18, 20, 10, 18, 23]

```
>>> L1.count (18)
2
>>> L1.count (28)
0
```

9. **The reverse() method**: The reverse() reverses the items of the list. This is done "in place", i.e, it does not create a new list. The syntax to use reverse method is:

   List.reverse ()

   - Takes no argument, returns no list; reverses the list 'in place' and does not return anything.

For example,

```
>>> t1
['e', 'i', 'q', 'a', 'q', 'p']
>>> t1.reverse ()
>>> t1
['p', 'q', 'a', 'q', 'i', 'e']
>>> t2 = [3, 4, 5]
>>> t3 = t2. reverse ()
>>> t3
>>> t3
[5, 4, 3]
```

10. The sort() method: The sort () function sorts the items of the list, by default in increasing order. this is done "in place", i.e it does not create a new list. It is used as per following syntax:

List.sort ()

For example,

```
>>> t1 = ['e', 'i', 'q', 'a', 'q', 'p']
>>> t1.sort ()
>>> t1
['a', 'e', 'i', 'p', 'q', 'q']
```

Quick Interesting Facts:

- The + operator adds one List to the end of another. The * operator repeats a List.
- List slice is an extracted part of List; List slice is a List in itself.
- Common List manipulation functions are: len (), index (), and List ().

TUPLE:

Introduction:

Tuple is a type of sequence like strings and lists but it differs from them in the way that lists are mutable but strings and tuples are immutable. This chapter is dedicated to basic tuple manipulation in Python.

Creating and Accessing Tuples:

A tuple is standard data type of Python that can store a sequence of values belonging to any type. The Tuples are depicted through parentheses i.e round brackets e.g following are some types in Python:

```
( )                         # tuple with no member, empty tuple
(1, 2, 3)                   # tuple of integers
(1, 2, 5, 3, 7, 9)          # tuple of numbers (integers and floating point)
('a', 'b', 'c')             # tuple of characters
('a', 1, 'b', 3.5, 'zero')  # tuple of mixed value types
('One', 'Two', 'Three')     # tuple of strings
```

Creating Tuples:-

Creating a tuple is similar to list creation, but here you need to put a number of expressions in parentheses. That is use round brackets to indicate the start and end of the tuple, and separate the items by commas. For example:

```
(2, 4, 6)
('abc", 'def')
(1, 2.0, 3, 4.0)
( )
```

Thus to create a tuple you can write in the form given below:

```
T = ( )
T = (value, ...)
```

Creating Tuples from Existing Sequences:

You can also use the built-in tuple type object (tuple()) to create tuples from sequences as per the syntax given below:

```
T = tuple (<sequence>)
```

where <sequence> can be any kind of sequence object including strings, lists and tuples.
Python creates the individual elements of the tuple from the individual elements of passed sequence. If you pass in another tuple, the tuple function makes a copy.
Consider following examples:

```
>>> t1 = tuple ('hello')
>>> t1
('h', 'e', 'l', 'l', 'o')
>>> L = ['w', 'e', 'r', 't', 'y']
>>> t2 = tuple (L)
>>> t2
('w', 'e', 'r', 't', 'y')
```

You can use this method of creating tuples of single characters or single digits via keyboard input.
Consider the code below:

```
t1 = tuple (input('Enter tuple elements:'.))
Enter tuple elements : 234567
>>> t1
('2', '3', '4', '5', '6', '7')
tuple = eval (input ("Enter tuple to be added:"))
print ("Tuple you entered:", tuple)
```

Accessing Tuples:

Tuples are immutable (non-editable) sequences having a progression of elements. Thus, like lists, you can access its individual elements. Before we talk about that, let us learn about how elements are indexed in tuples.

Accessing Individual Elements:

As mentioned, the individual elements of a tuple are accessed through their indexes given in square brackets. Consider the following examples:

```
>>> vowels = ('a', 'e', 'i', 'o', 'u')
>>> vowels [0]
'a'
>>> vowels [4]
'u'
>>> vowels [-1]
'u'
>>> vowels [-5]
'a'
```

Traversing a Tuple:-

```
for <item> in <Tuple> :
    process each item here
```

For example, following loop shows each item of a tuple T in separate lines:

```
T = ('p', 'y', 't', 'h', 'o', 'n')
for a in T :
    print (T[a])
```

The above loop will produce result as :

```
p
y
t
h
o
n
```

Joining Tuples:-

Joining two tuples is very easy just like you perform addition, literally ;). the + operator, the concatenation operator, when used with two tuples, joins two tuples.

Consider the example given below:

```
>>> tpl1 = (1, 3, 5)
>>> tpl2 = (6, 7, 8)
>>> tpl1 + tpl2
(1, 3, 5, 6, 7, 8)
```

As you can see that the resultant tuple has firstly elements of first tuple lst1 and followed by elements of second tuple lst2. You can also join two or more tuples to form a new tuple, e.g,

```
>>> tpl1 = (10, 12, 14)
>>> tpl2 = (20, 22, 24)
>>> tpl3 = (30, 32, 34)
>>> tpl = tpl1 + tpl2 + tpl3
>>> tpl
(10, 12. 14, 20, 22, 24, 30, 32, 34)
```

Important:

Sometimes you need to concatenate a tuple (say tpl) with another tuple containing only one element. In that case, if you write statement like:

```
>>> tpl + (3)
```

Python will return an error like:

Repeating or Replicating Tuples:

Like strings and lists, you can use * operator to replicate a tuple specified number of times, e.g, if tpl1 is (1, 3, 5), then

```
>>> tpl1 * 3
(1, 3, 5, 1, 3, 5, 1, 3, 5)
```

Slicing the Tuples:

Tuple slices, like list-slices or string slices are the sub parts of the tuple extracted out. You can use indexes of tuple elements to create tuple slices as per following format:

```
seq = T [start : stop]
>>> tpl = (10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> seq = tpl [3 : -3]
>>> seq
(20, 22, 24)
>>> tpl = (10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> tpl = [3 : 30]
(20, 22, 24, 30, 32, 34)
>>> tpl [-15 : 7]
(10, 12, 14, 20, 22, 24, 30)
```

Tuples also support slice steps too. That is, if you want to extract, not consecutive but every other element of the tuple, there is a way out - the slice steps. The slice steps are used as per following format:

seq = T[start : stop : step]

Consider some examples to understand this.

```
>>> tpl
(10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> tpl [0 : 2]
(10, 14, 22, 30, 32)
>>> tpl [2 : 10 : 3]
(14, 24, 34)
>>> tpl [: : 3]
(10, 20, 30)
```

Comparing Tuples:

You can compare two tuples without having to write code with loop for it. For comparing two tuples, you can use comparison operators, i.e, <, >, =, != etc. For comparison purposes, Python internally compares individual elements of two tuples, applying all the comparison rules that you have read earlier. Consider following code:

```
>>> a = (2, 3)
>>> b = (2, 3)
>>> a = = b
True
>>> c = ('2', '3')
>>> a = = c
False
>>> a > b
False
>>> d = (2.0, 3.0)
>>> d > a
False
>>> d = = a
True
>>> e = (2, 3, 4)
>>> a < e
True
```

You can also discuss non-equality comparisons of two sequences. Elements in tuples are also compared on similar lines.

Unpacking Tuples:-

Creating a tuple from a set of values is called packing and its reverse, i.e, creating individual values from a tuple's elements is called unpacking.

Unpacking is done as per syntax :

```
<variable1>, <variable2>, <variable3>, ... = t
```

Where the number of variables in the left side of assignment must match the number of elements in the tuple.

For example, if we have a tuple as :

```
t = (1, 2, 'A', 'B')
```

The length of above tuple t is 4 as there are four elements in it. Now to unpack it, we can write:

```
w, x, y, z = t
```

Python will now assign each of the elements of tuple t to the variables on the left side of assignment operator. That is, you can now individually print the values of these variables, somewhat like:

```
print (w)
print (x)
print (y)
```

```
    print (z)
```

The above code will yield the result as :

```
    1
    2
    'A'
    'B'
```

As per Guido van Rossom, the creator of Python language -
"Tuple unpacking requires that the list of variables on the left has the same number of elements as the length of the tuple"

Deleting Tuples:

The **del** statement of Python is used to delete elements and objects but as you know that tuples are immutable, which also means that individual elements of tuple cannot be deleted, i.e., if you give a code like:

```
    >>> del t1 [2]
```

It shows error message

But you can delete a complete tuple with **del** statement as:

del <tuple _ name>

For Example,

```
    >>> t1 = (5, 7, 3, 9, 12)
    >>> t1
    (5, 7, 3, 9, 12)
    >>> del t1
    >>> print (t1)
    Traceback (most recent call last):
        File "<ipython-input-89-04f730070f35>, "line 1, in <module>
            print (t1)
    Name Error: name 't1' is not defined
```

Tuple Functions and Methods:

Python also offers many built-in functions and methods for tuple manipulation. Let us now have a look at some useful built-in tuple manipulation methods.

1. The len () method:

This method returns length of the tuple, i.e, the count of elements in the tuple.

Syntax :

len (<tuple>)

- Takes tuple name as argument and returns an integer.

Example:

```
    >>> employee = ('John', 10000, 24, 'Sales')
    >>> len (employee)
    4
```

2. The max () method:

This method returns the element from the tuple having maximum value.

Syntax:

max (<tuple>)

- Takes tuple name as argument and returns an object (the element with maximum value)

Example:

```
>>> tp1 = (10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> max (tpl)
34
>>> tpl2 = ("Karan", "Zubin", "Zara", "Ana")
>>> max(tpl2)
'Zubin'
```

3. The min () method:

This method returns the element from the tuple having minimum value.

Syntax :

min (<tuple>)

- Takes tuple name as argument and returns an object (the element with minimum value)

Example:-

```
>>> tpl = (10, 12, 14, 20, 22, 24, 30, 32, 34)
10
>>> tpl2 = ("Karan", "Zubin", "Zara", "Ana")
>>> min (tpl2)
'Ana'
```

Like max(), for min () to work, the elements of tuple should be of same type.

4. The index () method:

The index () works with tuples in the same way it works with lists. That is, it returns the index of an existing element of a tuple. It is used as :

<tuplename> . index (<item>)

Example:-

```
>>> t1 = (3, 4, 5, 6, 0)
>>> t1.index (5)
2
```

But if the given item does not exist in tuple, it raises **Value Error** exception.

5. The count () function:

The count () method returns the count of a member element/ object in a given sequence (list/ tuple). You can use the count () function as per following syntax:

<sequence name>.count(<object>)

Example:-

```
>>> t1 = (2, 4, 2, 5, 7, 4, 8, 9, 11, 7, 2)
>>> t1.count (2)
3
>>> t1.count (7)
2
>>> t1.count (11)
1
```

For an element not in tuple, it returns 0 (zero)

6. The tuple () method:

This method is actually constructor method that can be used to create tuples from different types of values.

Syntax:

tuple (<sequence>)

Takes an optional argument of sequence type; Returns a tuple. With no argument, it returns empty tuple

Example:

Creating empty tuple

```
>>> tuple ()
()
```

Creating a tuple from a list

```
>>> t = tuple ([1, 2, 3])
>>> t
(1, 2, 3)
```

Dictionary:

Dictionary – Key: Value Pairs:

Among the built-in Python data types is a very versatile type called a dictionary. Dictionaries are simply another type of collection in Python, but with a twist. Rather than having an index associated with each data-item (just like in lists or strings), Dictionaries in Python have a "key" and a "value of that key". That is, Python dictionaries are collection of some key-value pairs.

Creating a Dictionary:

To create a dictionary, you need to include the key, value pairs in curly braces as per following

Syntax:

<dictionary-name>={ <key>: <value>, <key> : <value>....}

Teachers = {'Dimple" : "Computer Science", "Karen" : "Sociology", "Harpreet" : Mathematics", "Sabah" : Legal Studies"}

Key-Value pair	Key	Value
“Dimple” : “Computer Science”	“Dimple”	“Computer Science”
“Karen” : “Sociology”		“Karen” “Sociology”
“Harpreet” : “Mathematics”	“Harpreet”	“Mathematics”
“Sabah” : “Legal Studies”	“Sabah”	“Legal Studies”

Consider some more dictionary declarations:

$dict1 = \{\ \}$ # It is an empty dictionary with no elements

DaysInMonths = { “January” : 31, “February” : 28, “March” : 31, “April” : 30, “May” : 31,
“June” : 30, “July” : 31, “August” : 31, “September” : 30,
“October” : 31, “November” : 30, “December” : 31 }

BirdCount = {“Finch” : 10, “Myna” : 13, “Parakeet” : 16,
“Hornbil1” : 15, “Peacock” : 15 }

Now you can easily identify the keys and corresponding values from above given dictionaries.

One things that you must know is that keys of a dictionary must be of immutable types, such as :

- A Python string
- A number
- A tuple (containing only immutable entries).

If you try to give a mutable type as key, Python will give you an error as : “unhashable type”. See below :

$>>> dict3 = \{[2,3]:"abc"\}$

:

TypeError : unhashable type : ‘list’

Accessing Elements of a Dictionary:

While accessing elements from a dictionary, you need the key. In dictionaries, the elements are accessed through the keys defined in the key : value pairs, as per the syntax shown below :

$< dictionary - name > [< key >]$

$>>> teachers["Karen"]$

and python will return sociology

Consider following example :

```
>>> d = {"Vowel1" : "a", "Vowel2" : "e", "vowel3" : "i", "Vowel4" : "o", "Vowel5" : 'u"}
>>> d
```

```
{'Vowel5' : 'u', 'Vowel4' : 'o', 'Vowel3' : 'i', 'Vowel2' : 'e', 'Vowel1' : 'a'}
>>>print (d)
{'Vowel5' : 'u', 'Vowel4' : 'o', 'Vowel3' : 'i', 'Vowel2' : 'e', 'Vowel1' : 'a'}
>>> d["Vowel1"]
'a'
>>> d["Vowel4"]
'o'
```

Traversing a Dictionary:-

Traversal of a collection means accessing and processing each element of it. Thus, traversing a dictionary also means the same and same and same is the tool for it, i.e, the python loops.

The for loop makes it easy to traverse or loop over the items in a dictionary, as per following syntax:

```
for <item> in <Dictionary> :
        process each item here
```

```
d1 = {t : "number", "a" : "string", (1,2) : "tuple"}
```

To traverse the above dictionary, you can write for loop as:

```
for key in d1 :
      print (key, ":", d1 [key])
```

The above loop will produce the output as shown here

```
a : string
(1, 2) : tuple
5 : number
```

```
PhoneDict = {"Madhav" : 1234567, "Steven" : 7654321, "Dilpreet" : 6734521, "Rabiya" : 4563217,
"Murughan" : 3241567, "Sampree" : 4673215}
  for name in PhoneDict:
        print (name, ":", PhoneDict[name])
```

The output produced by above program will be

```
Rabiya :      4563217
Murughan :    3241567
Madhav :      1234567
Dilpreet :     6734521
Steven :       7654321
Sampree :      4673215
```

Quick Interesting Facts:
A dictionary is an **unordered** and **mutable** Python **container** that stores mappings of unique **keys to values**. Dictionaries are written with curly brackets ({}), including **key-value** pairs separated by commas (,). A colon (:) separates each **key** from its **value**.

Characteristics of a Dictionary:

Dictionaries like lists are mutable and that is the only similarity they have with lists. Otherwise, dictionaries are different type of data structures with following characteristics:

1. Unordered Set:
2. Not a sequence:
3. Indexed by Keys, Not Numbers:
4. Keys must be Unique:
5. Mutable:

Adding Elements to Dictionary:

You can add new elements (key : value pair) to a dictionary using assignment as per following syntax. BUT the key being added must not exist in dictionary and must be unique. If the key already exists, then this statement will change the value of existing key and no new entry will be added to dictionary.

<dictionary> [<key>] = <value>

Consider the following example :

```
>>> Employee = {'name' : 'John', 'salary' : 10000, 'age' : 24}
>>> Employee ['dept'] = 'sales'
>>> Employee
{'salary' : 10000, 'dept' : 'sales', 'age' : 24, 'name' : 'John'}
```

Updating Existing Elements in a Dictionary:

Updating an element is similar to what we did just now. That is, you can change value of an existing key using assignment as per following syntax:

dictionary > [<key>] = <value>

Consider following example:

```
>>> Employee = {'name' : 'John', 'salary' : 10000, 'age' : 24}
>>> Employee ['salary'] = 20000
>>> Employee
```

```
{'salary' : 20000, 'age' : 24, 'name' : 'John'}
```

But make sure that the key must exist in the dictionary otherwise new entry will be added to the dictionary. Using this technique of adding key:value pairs, you can create dictionaries interactively at runtime by accepting input from user. Following program illustrates this.

Deleting elements from a Dictionary:

There are two methods for deleting elements from a dictionary.

(i) del command:

del<dictionary>[<key>]

Consider the following example:

```
>>>empl3
{'salary' : 10000, 'age' : 24, 'name' : 'John'}
>>>del empl3 ['age']
>>> empl3
  {'salary' : 10000, 'name' : 'John'}
```

But with del statement, the key that you are giving to delete must exist in the dictionary, otherwise Python raises exception (KeyError). See below:

```
>>> del empl3['new']
:
 KeyError : 'new'
```

(ii) pop ( ):

```
<dictionary>.pop(<key>)
```

The pop ( ) method will not only delete the key:value pair for mentioned key but also return the corresponding value. Consider following code example

```
>>> employee
{'salary' : 10000, 'age' : 24, 'name' : 'John'}
>>> employee. pop('age')
24
>>> employee
{'salary' : 10000, 'name' : 'John'}
```

If you try to delete a key which does not exist, the Python returns error. See below :

```
>>> employee. pop('new')
:
```

```
KeyError : 'new'
```

However, pop ( ) method allows you to specify what to display when the given key does not exist, rather than the default error message. This can be done as per following syntax:

```
<dictionary> . pop(<key>, <in-case-of-error-show-me)
```

For example :

```
>>> employee. pop ('new', "Not Found")
'Not Fund'
```

See, now python returned the specified message in place of error-message.

Checking for Existence of a Key:

Usual membership operators in and not in work with dictionaries as well. But they can check for the existence of keys only.

<key> in <dictionary>

<key> not in <dictionary>

```
>>> empl = {'salary' : 10000, 'age' :24, 'name' :'John'}
>>> 'age' in empl
True
 >>> 'John' in empl
 False
  >>> 'John' not in empl
   True
    >>> 'age' not in empl
   False
 >>> dict1
 {'age' : 25, 'name' : 'xyz', 'salary' : 23480.5}
>>>'name' in dict1
True
 >>> 'xyz' in dict1
False
```

However, if you need to search for a value in a dictionary, then you can use the in operator with >dictionary _ name>.values ( ), i.e

```
 >>>'xyz' in dictl. values ( )
True
```

Dictionary Functions and Methods

1. The len ( ) method

The method returns length of the dictionary, i.e, the count of elements (key : value pairs) in the dictionary. The syntax to use this method is given below:

len (<dictionary>)

Takes dictionary name as argument and returns an integer.

2. The clear ( ) method

This method removes all items from the dictionary and the dictionary becomes empty dictionary post this method. the syntax to use this method is given below:

<dictionary>.clear()

Takes no argument, returns no value

3. The get ( ) method

With this method, you can get the item with the given key, similar to dictionary [key]. If the key is not present, Python by default gives error, but you can specify your own message through default argument as per following syntax:

<dictionary> . get (key . [default])

Takes key as essential argument and returns corresponding value if the key exists or when key does not exist, it returns Python's Error if default argument is missing otherwise returns default argument.

Example:

```
>>> empl1
{'salary' : 10000, 'dept' : 'sales', 'age' : 24, 'name' : 'John'}
>>> empl1.get ('dept')
'sales'
```

See it returned value for given key if key exists in the dictionary. If you only specify the key as argument, which does not exist in the dictionary, then Python will return error:

```
>>> empl1.get ('desig')
:
NameError: name'desig' is not defined
```

In place of error, you can also specify your own custom error message as second argument:

```
>>> empl1. get ('desig', "Error ! ! key not found")
'Error ! ! key not found'
```

4. The items ( ) method

This method returns all of the items in the dictionary as a sequence of (key, value) tuples. Note that these are returned in no particular order.

<dictionary> . items ( )

Takes no argument; Returns a sequence of (key, value) pairs.

Example:

```
employee = {'name' : 'John', 'salary' : 10000, 'age' : 24}
myList = employee . items (  )
for x in myList:
        print (x)
```

The above code gives output as:

```
(  'salary', 10000)
 ('age',  24)
 (  'name' , 'John')
```

As the items () function returns a sequence of (key value) pairs, you can write a loop having two variables to access key value pairs.

5. The keys () method:

You have already worked with this method. This method returns all of the keys in the dictionary as a sequence of keys (in form of a list). Note that these are returned in no particular order.

<dictionary>. keys ()

Takes no argument; Returns a list sequence of keys.

Example:

```
>>> employee= {'salary' : 10000, 'dept' : 'sales', 'age' : 24, 'name' : 'John'}
>>> employee.keys ( )
   ['salary', 'dept', 'age', 'name'
```

6. The values () method:

This method returns all the values from the dictionary as a sequence (a list). Note that these are returned in no particular order. The syntax to use this method is given below:

<dictionary>.values ()

Takes no argument; Returns a list sequence of values

Example:

```
>>> employee
```

```
{'salary' : 10000, 'dept' : 'sales', 'age' : 24, 'name' : 'John'}
  >>>employee.values ()
 {10000, 'sales', 24, 'John']
```

7. The update () method:

This method merges key : value pairs from the new dictionary into the original dictionary, adding or replacing as needed. The items in the new dictionary are added to the old one and override any items already there with the same keys. The syntax to use this method is given below:

<dictionary>. update (<other-dictionary>):

Example:

```
>>> employee1={'name' : 'John', 'salary' : 10000, 'age' : 24}
>>> employee2= {'name' : 'Diya', 'salary' : 54000, 'dept' : 'sales'}
 >>> employee1. update (employee2)
>>> employee 1
{'salary' : 54000, 'dept' : 'sales', 'name' : 'Diya', 'age' : 24}
 >>> employee2
{'salary' : 54000, 'dept' : 'sales', 'name' : 'Diya'}
```

This method is equivalent to the following Python statement:

```
for key in newDict. keys () :
        dictionary [k] = newDict[k]
```

****