

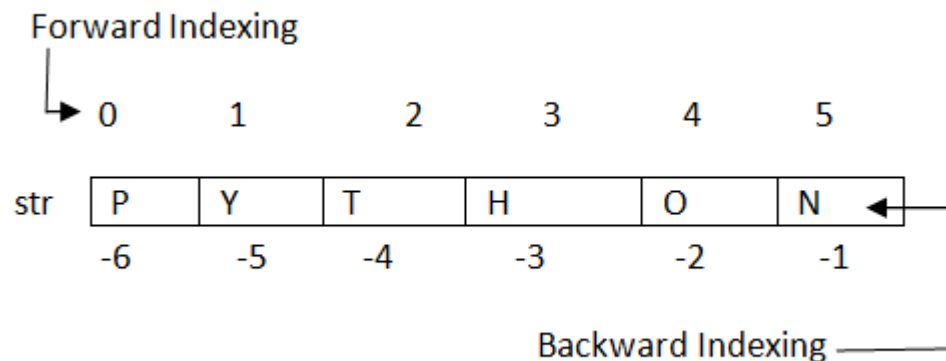
# PYTHON STRINGS

- Strings are the simplest and easy to use in Python.
- String python's are immutable.
- We can simply create Python String by enclosing a text in single as well as double quotes. Python treats both single and double quotes statements same.

## Accessing Strings:

- In Python, Strings are stored as individual characters in a contiguous memory location.
- The benefit of using String is that it can be accessed from both the directions in forward and backward.
- Both forward as well as backward indexing are provided using Strings in Python.
- Forward indexing starts with 0,1,2,3,....
- Backward indexing starts with -1,-2,-3,-4,....

Eg:



# Strings Operators

There are basically 3 types of Operators supported by String:

- Basic Operators.
- Membership Operators.
- Relational Operators.

## Basic Operators:

There are two types of basic operators in String. They are "+" and "\*".

### String Concatenation Operator :(+)

The concatenation operator (+) concatenate two Strings and forms a new String.

eg:

```
>>> "ratan" + "jaishwal"
```

Output:

```
'ratanjaishwal'
```

**NOTE: Both the operands passed for concatenation must be of same type, else it will show an error.**

Eg:

```
'abc' + 3      TypeError: cannot concatenate 'str' and 'int' objects
```

## Replication Operator: (\*)

Replication operator uses two parameter for operation. One is the integer value and the other one is the String.

The Replication operator is used to repeat a string number of times. The string will be repeated the number of times which is given by the integer value.

Eg:

```
>>> 5*"Vimal"
```

Output:

```
'VimalVimalVimalVimalVimal'
```

NOTE: We can use Replication operator in any way i.e., `int * string` or `string * int`. Both the parameters passed cannot be of same type.

```
>>> '$' * 5
```

Output

```
'$$$$$'
```

# Membership Operators

There are two types of Membership operators:

- 1) in:"in" operator return true if a character or the entire substring is present in the specified string, otherwise false.
- 2) not in:"not in" operator return true if a character or entire substring does not exist in the specified string, otherwise false.

Eg:

```
>>> str1="javatpoint"
```

```
>>> str2='sssit'
```

```
>>> str3="seomount"
```

```
>>> str4='java'
```

```
>>> str5="it"
```

```
>>> str6="seo"
```

```
>>> str4 in str1
```

```
True
```

```
>>> str5 in str2
```

```
>>> str5 in str2
```

```
True
```

```
>>> str6 in str3
```

```
True
```

```
>>> str4 not in str1
```

```
False
```

```
>>> str1 not in str4
```

```
True
```

## Relational Operators:

All the comparison operators i.e., (<,>,<=,>=,==,!=,<>) are also applicable to strings.

The Strings are compared based on the ASCII value or Unicode(i.e., dictionary Order).

Eg:

```
>>> "RAJAT"=="RAJAT"
```

True

```
>>> "alisha">='Alisha'
```

True

```
>>> "Z"<>"z"
```

True

## Slice Notation:

String slice can be defined as substring which is the part of string. Therefore further substring can be obtained from a string.

There can be many forms to slice a string. As string can be accessed or indexed from both the direction and hence string can also be sliced from both the direction that is left and right.

Syntax:

**<string\_name>[startIndex:endIndex],**

**<string\_name>[:endIndex],**

**<string\_name>[startIndex:]**

Example:

```
>>> str="Nikhil"
```

```
>>> str[0:6]
```

```
'Nikhil'
```

```
>>> str[0:3]
```

```
'Nik'
```

```
>>> str[2:5]
```

```
'khi'
```

```
>>> str[:6]
```

```
'Nikhil'
```


```
>>> str[3:]
```

```
'hil'
```

Note: startIndex in String slice is inclusive whereas endIndex is exclusive.

## String Functions and Methods:

<code>capitalize()</code>	It capitalizes the first character of the String.
<code>count(string,begin,end)</code>	Counts number of times substring occurs in a String between begin and end index.
<code>endswith(suffix ,begin=0,end=n)</code>	Returns a Boolean value if the string terminates with given suffix between begin and end.
<code>find(substring ,beginIndex, endIndex)</code>	It returns the index value of the string where substring is found between begin index and end index.
<code>index(subsring, beginIndex, endIndex)</code>	Same as <code>find()</code> except it raises an exception if string is not found.
<code>isalnum()</code>	It returns True if characters in the string are alphanumeric i.e., alphabets or numbers and there is at least 1 character. Otherwise it returns False.
<code>isalpha()</code>	It returns True when all the characters are alphabets and there is at least one character, otherwise False.
<code>isdigit()</code>	It returns True if all the characters are digit and there is at least one character, otherwise False.

<code>islower()</code>	It returns True if the characters of a string are in lower case, otherwise False.
<code>isupper()</code> 	It returns False if characters of a string are in Upper case, otherwise False.
<code>isspace()</code>	It returns True if the characters of a string are whitespace, otherwise false.
<code>len(string)</code>	<code>len()</code> returns the length of a string.
<code>lower()</code>	Converts all the characters of a string to Lower case.
<code>upper()</code>	Converts all the characters of a string to Upper Case.
<code>startswith(str ,begin=0,end=n)</code>	Returns a Boolean value if the string starts with given str between begin and end.
<code>swapcase()</code>	Inverts case of all characters in a string.
<code>lstrip()</code>	Remove all leading whitespace of a string. It can also be used to remove particular character from leading.
<code>rstrip()</code>	Remove all trailing whitespace of a string. It can also be used to remove particular character from trailing.



## 1) capitalize()

```
>>> 'abc'.capitalize()
```

Output:

```
'Abc '
```

## 2) count(string)

```
msg = "welcome to sssit";  
substr1 = "o";  
print msg.count(substr1, 4, 16)  
substr2 = "t";  
print msg.count(substr2)
```

Output:

```
>>>
```

```
2
```

```
2
```

## 3) isspace()

```
string1="  ";
```

```
print string1.isspace();
```

```
string2="WELCOME TO WORLD OF PYT"
```

```
print string2.isspace();
```

Output:

```
>>>
```

```
True
```

```
False
```

## 4) len(string)

```
string1="  ";
```

```
print len(string1);
```

```
string2="WELCOME TO SSSIT"
```

```
print len(string2);
```

Output:

```
>>>
```

```
4
```

```
16
```

## 5) endswith(string)

```
string1="Welcome to SSSIT";  
substring1="SSSIT";  
substring2="to";  
substring3="of";  
print string1.endswith(substring1);  
print string1.endswith(substring2,2,16);  
print string1.endswith(substring3,2,19);  
print string1.endswith(substring3);
```

Output:

```
>>>  
True  
False  
False  
False  
>>>
```

## 6) find(string)

```
str="Welcome to SSSIT";  
substr1="come";  
substr2="to";  
print str.find(substr1);  
print str.find(substr2);  
print str.find(substr1,3,10);  
print str.find(substr2,19);
```

Output:

```
>>>  
3  
8  
3  
-1  
>>>
```

## 7) index(string)

```
str="Welcome to world of SSSIT";
substr1="come";
substr2="of";
print str.index(substr1);
print str.index(substr2);
print str.index(substr1,3,10);
print str.index(substr2,19);
```

Output:

```
>>>
3
17
3
Traceback (most recent call last):
  File "C:/Python27/fin.py", line 7, in
    print str.index(substr2,19);
ValueError: substring not found
>>>
```

## 8) isalnum()

```
str="Welcome to sssit";
    print str.isalnum();
str1="Python47";
print str1.isalnum();
Output:
>>>
False
True
>>>
```

### 9) isalpha()

```
string1="HelloPython";  
# Even space is not allowed  
print string1.isalpha();  
string2="This is Python2.7.4"  
print string2.isalpha();
```

Output:

```
>>>  
True  
False  
>>>
```

### 10) isdigit()

```
string1="HelloPython";  
print string1.isdigit();  
string2="98564738"  
print string2.isdigit();
```

Output:

```
>>>  
False  
True
```

### 11) islower()

```
string1="Hello Python";  
print string1.islower();  
string2="welcome to "  
print string2.islower();
```

Output:

```
>>>  
False  
True  
>>>
```

### 12) isupper()

```
string1="Hello Python";  
print string1.isupper();  
string2="WELCOME TO"  
print string2.isupper();
```

Output:

```
>>>  
False  
True  
>>>
```

### 13) lower()

```
string1="Hello Python";  
print string1.lower();  
string2="WELCOME TO SSSIT"  
print string2.lower();
```

Output:

```
>>>  
hello python  
welcome to sssit  
>>>
```

### 14) upper()

```
string1="Hello Python";  
print string1.upper();  
string2="welcome to SSSIT"  
print string2.upper();  
Output:
```

```
>>>  
HELLO PYTHON  
WELCOME TO SSSIT  
>>>
```

### 15) startswith(string)

```
string1="Hello Python";  
print string1.startswith('Hello');  
string2="welcome to SSSIT"  
print string2.startswith('come',3,7);
```

Output:

```
>>>  
True  
True  
>>>
```

### 16) swapcase()

```
string1="Hello Python";  
print string1.swapcase();  
string2="welcome to SSSIT"  
print string2.swapcase();  
Output:
```

```
>>>  
hELLO pYTHON  
WELCOME TO sssit  
>>>
```

## 17) lstrip()

```
string1="  Hello Python";  
print string1.lstrip();  
string2="@@@@@@@@welcome to SSSIT"  
print string2.lstrip('@');
```

Output:

```
>>>  
Hello Python  
welcome to SSSIT  
>>>
```

## 18) rstrip()

```
string1="    Hello Python    ";  
print string1.rstrip();  
string2="@welcome to SSSIT!!!"  
print string2.rstrip('!');
```

Output:

```
>>>  
        Hello Python  
@welcome to SSSIT  
>>>
```

# Python Native Datatypes

## Python Numbers

While integers can be of any length, a floating point number is accurate only up to 15 decimal places (the 16th place is inaccurate).

Numbers we deal with everyday are decimal (base 10) number system. But computer programmers (generally embedded programmer) need to work with binary (base 2), hexadecimal (base 16) and octal (base 8) number systems. In Python we can represent these numbers by appropriately placing a prefix before that number.

Following table lists these prefix.

Number system prefix for Python numbers

Number System	Prefix
Binary	'0b' or '0B'
Octal	'0o' or '0O'
Hexadecimal	'0x' or '0X'

Here are some examples

```
>>> 0b1101011      # 107
107
>>> 0xFB + 0b10     # 251 + 2
253
>>> 0o15             # 13
13
```

# Python List

## Creating a List

In Python programming, a list is created by placing all the items (elements) inside a square bracket [ ], separated by commas.

It can have any number of items and they may be of different types (integer, float, string etc.).

A list can even have another list as an item. These are called nested list.

Examples:

```
my_list = []           # empty list
```

```
my_list = [1, 2, 3]    # list of integers
```

```
my_list = [1, "Hello", 3.4] # list with mixed datatypes
```

```
my_list = ["mouse", [8, 4, 6]] # nested list
```



## Accessing Elements in a List

There are various ways in which we can access the elements of a list.

### 1) Indexing

- ▮ We can use the index operator `[]` to access an item in a list.
- ▮ Index starts from 0. So, a list having 5 elements will have index from 0 to 4.
- ▮ Trying to access an element other than this will raise an `IndexError`.
- ▮ The index must be an integer. We can't use float or other types, this will result into `TypeError`.
- ▮ Nested lists are accessed using nested indexing.

Example:

```
>>> my_list = ['p', 'r', 'o', 'b', 'e']
```

```
>>> my_list[0]
```

```
'p'
```

```
>>> my_list[2]
```

```
'o'
```

```
>>> my_list[4]
```

```
'e'
```

```
>>> my_list[4.0]
```

```
...
```

```
TypeError: list indices must be integers, not float
```

```
>>> my_list[5]
```

```
...
```

```
IndexError: list index out of range
```

```
>>> n_list = ["Happy", [2,0,1,5]]
```

```
>>> n_list[0][1]      # nested indexing
```

```
'a'
```

```
>>> n_list[1][3]      # nested indexing
```

```
5
```

## 2) Negative indexing

- ▯ Python allows negative indexing for its sequences.
- ▯ The index of -1 refers to the last item, -2 to the second last item and so on.

Example:

```
>>> my_list = ['p', 'r', 'o', 'b', 'e']  
>>> my_list[-1]  
'e'  
>>> my_list[-5]  
'p'
```

### 3) Slicing

We can access a range of items in a list by using the slicing operator (colon).

```
>>> my_list = ['p','r','o','g','r','a','m','i','z']
>>> my_list[2:5]      # elements 3rd to 5th
['o', 'g', 'r']
```

```
>>> my_list[:-5]      # elements beginning to 4th
['p', 'r', 'o', 'g']
```

```
>>> my_list[5:]       # elements 6th to end
['a', 'm', 'i', 'z']
```

```
>>> my_list[:]        # elements beginning to end
['p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z']
```

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need two index that will slice that portion from the list.

P	R	O	G	R	A	M	I	Z	
0	1	2	3	4	5	6	7	8	9
-9	-8	-7	-6	-5	-4	-3	-2	-1	

## Changing or Adding Elements to a List

List are mutable, meaning, their elements can be changed unlike string or tuple.

We can use assignment operator (=) to change an item or a range of items.

```
>>> odd = [2, 4, 6, 8]      # mistake values
>>> odd[0] = 1             # change the 1st item
>>> odd
[1, 4, 6, 8]

>>> odd[1:4] = [3, 5, 7]   # change 2nd to 4th items
>>> odd                    # changed values
[1, 3, 5, 7
]
```

We can add one item to a list using **append()** method or add several items using **extend()** method.

```
>>> odd
```

```
[1, 3, 5]
```

```
>>> odd.append(7)
```

```
>>> odd
```

```
[1, 3, 5, 7]
```

```
>>> odd.extend([9, 11, 13])
```

```
>>> odd
```

```
[1, 3, 5, 7, 9, 11, 13]
```

We can also use + operator to combine two lists. This is also called concatenation.

The \* operator repeats a list for the given number of times.

```
>>> odd
[1, 3, 5]
>>> odd + [9, 7, 5]
[1, 3, 5, 9, 7, 5]
>>> ["re"] * 3
['re', 're', 're']
```

Furthermore, we can insert one item at a desired location by using the method **insert()** or insert multiple items by squeezing it into an empty slice of a list.

```
>>> odd
[1, 9]

>>> odd.insert(1,3)
>>> odd
[1, 3, 9]

>>> odd[2:2] = [5, 7]   --- just inserting
>>> odd
[1, 3, 5, 7, 9]

>>> odd[3:4]=[1,2,3,4,5]  ---- inserting with replacement
>>> odd
[1, 3, 7, 1, 2, 3, 4, 5, 9]
```

## Deleting or Removing Elements from a List

We can delete one or more items from a list using the keyword `del`. It can even delete the list entirely.

```
>>> my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']
```

```
>>> del my_list[2]      # delete one item
```

```
>>> my_list
['p', 'r', 'b', 'l', 'e', 'm']
```

```
>>> del my_list[1:5]    # delete multiple items
```

```
>>> my_list
['p', 'm']
```

```
>>> del my_list         # delete entire list
```

```
>>> my_list
```

```
...
NameError: name 'my_list' is not defined
```



We can use **remove()** method to remove the given item or **pop()** method to remove an item at the given index.

The pop() method removes and returns the last item if index is not provided.

This helps us implement lists as stacks (first in, last out data structure).

We can also use the **clear()** method to empty a list.

```
>>> my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']
```

```
>>> my_list.remove('p')
```

```
>>> my_list
```

```
['r', 'o', 'b', 'l', 'e', 'm']
```

```
>>> my_list.pop(1)
```

```
'o'
```

```
>>> my_list
```

```
['r', 'b', 'l', 'e', 'm']
```

```
>>> my_list.pop()
```

```
'm'
```

```
>>> my_list
```

```
['r', 'b', 'l', 'e']
```

```
>>> my_list.clear()      --- may not work in lower versions
```

```
>>> my_list              --- Added in Python 3.3
```

```
[]
```

Finally, we can also delete items in a list by assigning an empty list to a slice of elements.

```
>>> my_list = ['p','r','o','b','l','e','m']
>>> my_list[2:3] = []
>>> my_list
['p', 'r', 'b', 'l', 'e', 'm']
```

```
>>> my_list[2:5] = []
>>> my_list
['p', 'r', 'm']
```

## List Membership Test

We can test if an item exists in a list or not, using the keyword in.

```
>>> my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']
```

```
>>> 'p' in my_list
```

```
True
```

```
>>> 'a' in my_list
```

```
False
```

```
>>> 'c' not in my_list
```

```
True
```

# Functions and Methods of Lists:

There are many Built-in functions and methods for Lists. They are as follows:

**There are following List functions:**

Function	Description
min(list)	Returns the minimum value from the list given.
max(list)	Returns the largest value from the given list.
len(list)	Returns number of elements in a list.
cmp(list1,list2)	Compares the two list.
list(sequence)	Takes sequence types and converts them to lists.

index(object)	Returns the index value of the object.
count(object)	It returns the number of times an object is repeated in list.
pop()/pop(index)	Returns the last object or the specified indexed object. It removes the popped object.
insert(index,object)	Insert an object at the given index.
extend(sequence)	It adds the sequence to existing list.
remove(object)	It removes the object from the given List.
reverse()	Reverse the position of all the elements of a list.
sort()	It is used to sort the elements of the List.

```
>>> len(my_list)
```

```
7
```

```
>>> min(my_list)
```

```
0
```

```
>>> list1=[101,981,'abcd','xyz','m']
```

```
>>> list2=['aman','shekhar',100.45,98.2]
```

```
>>> list3=[101,981,'abcd','xyz','m']
```

```
>>> cmp(list1,list2)
```

```
-1
```

```
>>> cmp(list2,list1)
```

```
1
```

```
>>> cmp(list3,list1)
```

```
0
```

```
>>>
```

```
>>> my_list=[1,2,3,4,5]
```

```
>>> sum(my_list)
```

```
15
```

```
>>> a = [3, 6, 8, 2, 78, 1, 23, 45, 9]
```

```
>>> sorted(a)
```

```
[1, 2, 3, 6, 8, 9, 23, 45, 78]
```

```
>>> a
```

```
[3, 6, 8, 2, 78, 1, 23, 45, 9]
```

```
>>> sorted(a, reverse=True)
```

```
[78, 45, 23, 9, 8, 6, 3, 2, 1]
```

## List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like using loop:

```
squares = []  
for x in range(10):  
    squares.append(x**2)
```

**output**

```
squares [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

**We can obtain the same result with list comprehensions:**

```
squares = [x**2 for x in range(10)]
```

**output**

```
squares [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```





# Python Tuple

- | A tuple is a sequence of immutable objects, therefore tuple cannot be changed.
- | The objects are enclosed within parenthesis and separated by comma.
- | Tuple is similar to list. Only the difference is that list is enclosed between square bracket, tuple between parenthesis and List have mutable objects whereas Tuple have immutable objects.
- | If Parenthesis is not given with a sequence, it is by default treated as Tuple.
- | A tuple can have any number of items and they may be of different types (integer, float, list, string etc.).

```
my_tuple = ()                # empty tuple
my_tuple = (1, 2, 3)         # tuple having integers
my_tuple = (1, "Hello", 3.4) # tuple with mixed datatypes

my_tuple = 3, 4.6, "dog"     # tuple can be created without
                             # parentheses, also called tuple packing
a, b, c = my_tuple           # tuple unpacking is also
possible
```

Creating a tuple with one element is a bit tricky.  
Having one element within parentheses is not enough. We will need a trailing comma to indicate that it is in fact a tuple.

```
>>> my_tuple = ("hello")    # only parentheses is not enough
>>> type(my_tuple)
<class 'str'>
```

```
>>> my_tuple = ("hello",)   # need a comma at the end
>>> type(my_tuple)
<class 'tuple'>
```

```
>>> my_tuple = "hello",     # parentheses is optional
>>> type(my_tuple)
<class 'tuple'>
```

```
>>> tupl1='a', 'mahesh', 10.56
>>> tupl2=tupl1, (10,20,30)   #nested tuple
```

```
>>> tupl1
('a', 'mahesh', 10.56)
>>> tupl2
(('a', 'mahesh', 10.56), (10, 20, 30))
```

# Accessing Tuple

Tuple can be accessed in the same way as List.

## 1. Indexing

```
>>> my_tuple = ['p', 'e', 'r', 'm', 'i', 't']
>>> my_tuple[0]
'p'
>>> my_tuple[5]
't'
>>> my_tuple[6]    # index must be in range
...
IndexError: list index out of range
>>> my_tuple[2.0]  # index must be an integer
...
TypeError: list indices must be integers, not float
>>> n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
>>> n_tuple[0][3]   # nested index
's'
>>> n_tuple[1][1]   # nested index
4
>>> n_tuple[2][0]   # nested index
1
>>> n_tuple = ([8, 4, 6], (1, 2, 3), "mouse")
>>> n_tuple[2][0]
'm'
```

## 2) Negative Indexing

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

```
>>> my_tuple = ['p','e','r','m','i','t']  
>>> my_tuple[-1]  
't'
```

```
>>> my_tuple[-6]  
'p'
```

### 3) Slicing

We can access a range of items in a tuple by using the slicing operator (colon).

```
>>> my_tuple = ('p','r','o','g','r','a','m','i','z')
>>> my_tuple[1:4] # elements 2nd to 4th
('r', 'o', 'g')
```

```
>>> my_tuple[:-7] # elements beginning to 2nd
('p', 'r')
```

```
>>> my_tuple[7:] # elements 8th to end
('i', 'z')
```

```
>>> my_tuple[:] # elements beginning to end
('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

P	R	O	G	R	A	M	I	Z	
0	1	2	3	4	5	6	7	8	9
-9	-8	-7	-6	-5	-4	-3	-2	-1	

## Tuple Operations

### a) Adding Tuple:

Tuple can be added by using the concatenation operator(+) to join two tuples.

```
data1=(1,2,3,4)
data2=('x','y','z')
data3=data1+data2
print data1
print data2
print data3
```

Output:

```
>>>
(1, 2, 3, 4)
('x', 'y', 'z')
(1, 2, 3, 4, 'x', 'y', 'z')
>>>
```

Note: The new sequence formed is a new Tuple.

## **b) Replicating Tuple:**

Replicating means repeating. It can be performed by using '\*' operator by a specific number of time.

Eg:

```
tuple1=(10,20,30);  
tuple2=(40,50,60);  
print tuple1*2  
print tuple2*3
```

Output:

```
>>>  
(10, 20, 30, 10, 20, 30)  
(40, 50, 60, 40, 50, 60, 40, 50, 60)  
>>>
```

### **c) Updating elements in a List:**

Elements of the Tuple cannot be updated.

This is due to the fact that Tuples are immutable.

Whereas the Tuple can be used to form a new Tuple.

Eg:

```
data=(10,20,30)
```

```
data[0]=100
```

```
print data
```

Output:

```
>>>
```

```
Traceback (most recent call last):
```

```
    File "C:/Python27/t.py", line 2, in
```

```
    data[0]=100
```

```
TypeError: 'tuple' object does not support item assignment
```

```
>>>
```



#### **d) Deleting elements from Tuple:**

Deleting individual element from a tuple is not supported. However the whole of the tuple can be deleted using the del statement.

Eg:

```
data=(10,20,'rahul',40.6,'z')
```

```
print data
```

```
del data[3]                                # will show an error
```

```
del data                                   #will delete the tuple data
```

```
print data
```

```
#will show an error since tuple data is already deleted
```

## e) Tuple Membership Test

We can test if an item exists in a tuple or not, using the keyword `in`.

```
>>> my_tuple = ('a', 'p', 'p', 'l', 'e',)  
>>> 'a' in my_tuple  
True
```

```
>>> 'b' in my_tuple  
False
```

```
>>> 'g' not in my_tuple  
True
```

# Functions of Tuple:

There are following in-built Type Functions:

Function	Description
<code>min(tuple)</code>	Returns the minimum value from a tuple.
<code>max(tuple)</code>	Returns the maximum value from the tuple.
<code>len(tuple)</code>	Gives the length of a tuple
<code>cmp(tuple1,tuple2)</code>	Compares the two Tuples.
<code>tuple(sequence)</code>	Converts the sequence into tuple.

## Few more Function Examples:

```
>>> tuple=(1,2,3,4)
>>> sum(tuple)           # use of sum() function
10
```

```
>>> tuple=5,2,7,4
>>> tuple
(5, 2, 7, 4)
>>> sorted(tuple)                # use of sorted() function
[2, 4, 5, 7]
>>> tuple
(5, 2, 7, 4)
```

```
>>> sorted(tuple,reverse=True)
[7, 5, 4, 2]
```

```
>>> tuple=1,2,2,3,4,5
>>> tuple.count(2)           # use of count() function
2
>>> tuple.count(4)
1
```

```
>>> tuple.index(5)           # use of index() function
5                             # Element 5 is at index 5
```

# Advantages of Tuple over List

1. Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
2. Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.
3. If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

# Python Dictionary

- Dictionary is an unordered set of key and value pair.
- It is an container that contains data, enclosed within curly braces.
- The pair i.e., key and value is known as item.
- The key and the value is separated by a colon(:). This pair is known as item. Items are separated from each other by a comma(,). Different items are enclosed within a curly brace and this forms Dictionary.
- Dictionary is mutable i.e., value can be updated.
- Key must be unique and immutable. Value is accessed by key. Value can be updated while key cannot be changed.
- Dictionary is known as **Associative array** since the Key works as Index and they are decided by the user.

**eg:**

```
data={100:'Ravi' ,101:'Vijay' ,102:'Rahul'}
```

```
print data
```

**Output:**

```
>>>
```

```
{100: 'Ravi', 101: 'Vijay', 102: 'Rahul'}
```

```
>>>
```

```
plant={}
```

```
plant[1]='Ravi'
```

```
plant[2]='Manoj'
```

```
plant['name']='Hari'
```

```
plant[4]='Om'
```

```
print plant[2]
```

```
print plant['name']
```

```
print plant[1]
```

```
print plant
```

Output:

```
>>>
```

```
Manoj
```

```
Hari
```

```
Ravi
```

```
{1: 'Ravi', 2: 'Manoj', 4: 'Om', 'name': 'Hari'}
```

```
>>>
```

# Accessing Values

Since Index is not defined, a Dictionaries value can be accessed by their keys.

Syntax:

[key]

Eg:

```
data1={'Id':100, 'Name':'Suresh',  
'Profession':'Developer'}  
data2={'Id':101, 'Name':'Ramesh',  
'Profession':'Trainer'}  
print "Id of 1st employer is",data1['Id']  
print "Id of 2nd employer is",data2['Id']  
print "Name of 1st employer:",data1['Name']  
print "Profession of 2nd employer:",data2['Profession']
```

Output:

```
>>>
```

```
Id of 1st employer is 100  
Id of 2nd employer is 101  
Name of 1st employer is Suresh  
Profession of 2nd employer is Trainer
```



# Updation

The item i.e., key-value pair can be updated. Updating means new item can be added. The values can be modified.

Eg:

```
data1={'Id':100, 'Name':'Suresh',  
      'Profession':'Developer'}  
data2={'Id':101, 'Name':'Ramesh',  
      'Profession':'Trainer'}  
data1['Profession']='Manager'  
data2['Salary']=20000  
data1['Salary']=15000  
print data1  
print data2
```

Output:

```
>>>  
{'Salary': 15000, 'Profession': 'Manager', 'Id': 100,  
 'Name': 'Suresh'}  
{'Salary': 20000, 'Profession': 'Trainer', 'Id': 101,  
 'Name': 'Ramesh'}  
>>>
```

# Deletion

del statement is used for performing deletion operation.

An item can be deleted from a dictionary using the key.

Syntax:

```
del [key]
```

Whole of the dictionary can also be deleted using the del statement.

Eg:

```
data={100:'Ram', 101:'Suraj', 102:'Alok'}
```

```
del data[102]
```

```
print data
```

```
del data
```

```
print data    #will show an error since dictionary is  
deleted.
```

Output:

```
>>>
```

```
{100: 'Ram', 101: 'Suraj'}
```

```
Traceback (most recent call last):File
```

```
"C:/Python27/dict.py" line 5, in
```

# Dictionary Functions:

Functions	Description
<code>len(dictionary)</code>	Gives number of items in a dictionary.
<code>cmp(dictionary1,dictionary2)</code>	Compares the two dictionaries.
<code>str(dictionary)</code>	Gives the string representation of a string.

1) `len(dictionary)`:

Eg:

```
data={100:'Ram', 101:'Suraj', 102:'Alok'}
```

```
print data
```

```
print len(data)
```

Output:

```
>>>
```

```
{100: 'Ram', 101: 'Suraj', 102: 'Alok'}
```

```
3
```

```
>>>
```

## 2) cmp(dictionary1,dictionary2):

The comparison is done on the basis of key and value.

If, dictionary1 == dictionary2, returns 0.

dictionary1 < dictionary2, returns -1.

dictionary1 > dictionary2, returns 1.

Eg:

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}
```

```
data2={103:'abc', 104:'xyz', 105:'mno'}
```

```
data3={'Id':10, 'First':'Aman', 'Second':'Sharma'}
```

```
data4={100:'Ram', 101:'Suraj', 102:'Alok'}
```

```
print cmp(data1,data2)
```

```
print cmp(data1,data4)
```

```
print cmp(data3,data2)
```

Output:

```
>>>
```

```
-1
```

```
0
```

```
1
```

## 3) str(dictionary):

Eg:

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}
```

```
print str(data1)
```

Output:

```
>>>
```

```
{100: 'Ram', 101: 'Suraj', 102: 'Alok'}
```

```
>>>
```

# Dictionary Methods:

Methods	Description
<code>keys()</code>	Return all the keys element of a dictionary.
<code>values()</code>	Return all the values element of a dictionary.
<code>items()</code>	Return all the items(key-value pair) of a dictionary.
<code>update(dictionary2)</code>	It is used to add items of dictionary2 to first dictionary.
<code>clear()</code>	It is used to remove all items of a dictionary. It returns an empty dictionary.
<code>fromkeys(sequence,value1)/ fromkeys(sequence)</code>	It is used to create a new dictionary from the sequence where sequence elements forms the key and all keys share the values ?value1?. In case value1 is not give, it set the values of keys to be none.
<code>copy()</code>	It returns an ordered copy of the data.
<code>has_key(key)</code>	It returns a boolean value. True in case if key is present in the dictionary ,else false.
<code>get(key)</code>	Returns the value of the given key. If key is not present it returns none.

## **1) keys():**

Eg:

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}  
print data1.keys()
```

Output:

```
>>>  
[100, 101, 102]  
>>>
```

## **2) values():**

Eg:

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}  
print data1.values()
```

Output:

```
>>>  
['Ram', 'Suraj', 'Alok']  
>>>
```

### 3) items():

Eg:

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}  
print data1.items()
```

Output:

```
>>>  
[(100, 'Ram'), (101, 'Suraj'), (102, 'Alok')]  
>>>
```

### 4) update(dictionary2):

Eg:

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}  
data2={103:'Sanjay'}  
data1.update(data2)  
print data1  
print data2
```

Output:

```
>>>  
{100: 'Ram', 101: 'Suraj', 102: 'Alok', 103: 'Sanjay'}  
{103: 'Sanjay'}  
>>>
```

## 5) clear():

Eg:

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}  
print data1  
data1.clear()  
print data1
```

Output:

```
>>>  
{100: 'Ram', 101: 'Suraj', 102: 'Alok'}  
{}  
>>>
```

## 6) fromkeys(sequence)/ fromkeys(seq,value):

Eg:

```
sequence=('Id' , 'Number' , 'Email')  
data={}  
data1={}  
data=data.fromkeys(sequence)  
print data  
data1=data1.fromkeys(sequence,100)  
print data1
```

Output:

```
>>>  
{'Email': None, 'Id': None, 'Number': None}  
{'Email': 100, 'Id': 100, 'Number': 100}
```



### 7) copy():

```
data={'Id':100 , 'Name':'Aakash' , 'Age':23}
```

```
data1=data.copy()
```

```
print data1
```

Output:

```
>>>
```

```
{'Age': 23, 'Id': 100, 'Name': 'Aakash'}
```

### 8) has\_key(key):

```
data={'Id':100 , 'Name':'Aakash' , 'Age':23}
```

```
print data.has_key('Age')
```

```
print data.has_key('Email')
```

Output:

```
>>>
```

```
True
```

```
False
```

### 9) get(key):

```
data={'Id':100 , 'Name':'Aakash' , 'Age':23}
```

```
print data.get('Age')
```

```
print data.get('Email')
```

Output:

```
>>>
```

```
23
```

```
None
```

# Dictionary Membership Test

We can test if a key is in a dictionary or not using the keyword `in`. Notice that membership test is for keys only, not for values.

```
>>> squares  
{1: 1, 3: 9, 5: 25, 7: 49, 9: 81}  
>>> 1 in squares  
True
```

```
>>> 2 not in squares  
True
```

```
>>> # membership tests for key only not value  
>>> 49 in squares  
False
```