List

The list is the most versatile datatype available in Python, which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that the items in a list need not be of the same type.

Lists

S.No.	Methods & Description
1	**list.append(obj)** Appends object obj to list
2	**list.count(obj)** Returns count of how many times obj occurs in list
3	**list.extend(seq)** Appends the contents of seq to list
4	**list.index(obj)** Returns the lowest index in list that obj appears
5	**list.insert(index, obj)** Inserts object obj into list at offset index
6	**list.pop(obj = list[-1])** Removes and returns last object or obj from list
7	**list.remove(obj)** Removes object obj from list
8	**list.reverse()** Reverses objects of list in place
9	**list.sort([func])** Sorts objects of list, use compare func if given

CODE:

```
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')


>>> fruits.count('tangerine')


>>> fruits.index('banana')


>>> fruits.index('banana', 4)


>>> fruits.reverse()
>>> fruits


>>> fruits.append('grape')
>>> fruits


>>> fruits.sort()
>>> fruits

>>>fruits[-1]

>>>fruits.count('grape')

>>>fruits.index('grape')
```

The del statement

There is a way to remove an item from a list given its index instead of its value: the del statement. This differs from the pop() method which returns a value. The del statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

CODE:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a


>>> del a[2:4]
>>> a

>>> del a[:]
>>> a


>>> del a

>>>print(a)
```

Tuple

Advantages of Tuple over List

Since, tuples are quite similar to lists, both of them are used in similar situations as well.

However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:

- We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
- Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

1. Indexing

We can use the index operator [] to access an item in a tuple where the index starts from 0. So, a tuple having 6 elements will have index from 0 to 5. Trying to access an element other that (6, 7,...) will raise an IndexError. The index must be an integer, so we cannot use float or other types. This will result into TypeError.

Code::

```
>>>my_tuple = ('p','e','r','m','i','t')


>>>print(my_tuple[0])


>>>print(my_tuple[5])


# nested tuple
>>>n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

>>>print(n_tuple[0][3])


>>>print(n_tuple[1][1])
```

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')
>>>print(my_tuple[-1])
```

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')

>>>print(my_tuple[1:4])
```



```
>>>print(my_tuple[:-7])


>>>print(my_tuple[7:])


>>>print(my_tuple[:])


>>>my_tuple = (4, 2, 3, [6, 5])

# we cannot change an element
# you will get an error:
# TypeError: 'tuple' object does not support item assignment

>>>my_tuple[1] = 9

# but item of mutable element can be changed
>>>my_tuple[3][0] = 9


# tuples can be reassigned
>>>my_tuple = ('p','r','o','g','r','a','m','i','z')
>>>print(my_tuple)
```

We can use + operator to combine two tuples. This is also called **concatenation**.
We can also **repeat** the elements in a tuple for a given number of times using the * operator.
Both + and * operations result into a new tuple.

```
>>>print((1, 2, 3) + (4, 5, 6))


>>>print(("Repeat",) * 3)
```

Deleting a Tuple

As discussed above, we cannot change the elements in a tuple. That also means we cannot delete or remove items from a tuple.But deleting a tuple entirely is possible using the keyword del.


```
>>>my_tuple = ('p','r','o','g','r','a','m','i','z')


>>>del my_tuple[3]

>>>del my_tuple
>>>my_tuple
```

Function	Description
all()	Return True if all elements of the tuple are true (or if the tuple is empty).
any()	Return True if any element of the tuple is true. If the tuple is empty, return False.
enumerate()	Return an enumerate object. It contains the index and value of all the items of tuple as pairs.
len()	Return the length (the number of items) in the tuple.
max()	Return the largest item in the tuple.
min()	Return the smallest item in the tuple
sorted()	Take elements in the tuple and return a new sorted list (does not sort the tuple itself).
sum()	Retrun the sum of all elements in the tuple.
tuple()	Convert an iterable (list, string, set, dictionary) to a tuple.

Dictionary

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example −

```
>>>dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
>>>print ("dict['Name']: ", dict['Name'])
>>>print ("dict['Age']: ", dict['Age'])
```

If we attempt to access a data item with a key, which is not a part of the dictionary, we get an error as follows −

```
>>>dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};
>>>print "dict['Alice']: ", dict['Alice']
```

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown in a simple example given below.

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School" # Add new entry

print ("dict['Age']: ", dict['Age'])
print ("dict['School']: ", dict['School'])
```

When the above code is executed, it produces the following result −

```
dict['Age']:  8
dict['School']:  DPS School
```

Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.To explicitly remove an entire dictionary, just use the **del** statement. Following is a simple example


```
>>>dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

>>>del dict['Name'] # remove entry with key 'Name'
>>>dict.clear()     # remove all entries in dict
>>>del dict         # delete entire dictionary

>>>print ("dict['Age']: ", dict['Age'])


>>>print ("dict['School']: ", dict['School'])
```

Properties of Dictionary Keys

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.
There are two important points to remember about dictionary keys −
(a) More than one entry per key is not allowed. This means no duplicate key is allowed. When duplicate keys are encountered during assignment, the last assignment wins. For example −

```
>>>dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}
>>>print ("dict['Name']: ", dict['Name'])
```

(b) Keys must be immutable. This means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed. Following is a simple example

```
>>>dict = {['Name']: 'Zara', 'Age': 7}
print ("dict['Name']: ", dict['Name'])
```

Strings

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes.

String Special Operators

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then −

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give - HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	print r'\n' prints \n and print R'\n'prints \n
%	Format - Performs String formatting	See at next section

String Formatting Operator

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the pack of having functions from C's printf() family. Following is a simple example −

```
>>>print ("My name is %s and weight is %d kg!" % ('Zara', 21))
```

S.No.	Format Symbol & Conversion
1	**%c** character
2	**%s** string conversion via str() prior to formatting
3	**%i** signed decimal integer
4	**%d** signed decimal integer
5	**%u** unsigned decimal integer
6	**%o** octal integer
7	**%x** hexadecimal integer (lowercase letters)
8	**%X** hexadecimal integer (UPPERcase letters)
9	**%e** exponential notation (with lowercase 'e')
10	**%E** exponential notation (with UPPERcase 'E')
11	**%f** floating point real number
12	**%g** the shorter of %f and %e
13	**%G** the shorter of %f and %E

Other supported symbols and functionality are listed in the following table −

S.No.	Symbol & Functionality

1	***** argument specifies width or precision
2	**-** left justification
3	**+** display the sign
4	**<sp>** leave a blank space before a positive number
5	**#** add the octal leading zero ('0') or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.
6	**0** pad from left with zeros (instead of spaces)
7	**%** '%%' leaves you with a single literal '%'
8	**(var)** mapping variable (dictionary arguments)
9	**m.n.** m is the minimum total width and n is the number of digits to display after the decimal point (if appl.)

Escape Characters

Following table is a list of escape or non-printable characters that can be represented with backslash notation.An escape character gets interpreted; in a single quoted as well as double quoted strings.

Backslash notation	Hexadecimal character	Description
\a	0x07	Bell or alert
\b	0x08	Backspace
\cx		Control-x
\C-x		Control-x

\e	0x1b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0.7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

String methods

S.No.	Methods & Description
1	**capitalize()** Capitalizes first letter of string
2	**center(width, fillchar)** Returns a string padded with *fillchar* with the original string centered to a total of *width* columns.
3	**count(str, beg = 0,end = len(string))** Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
4	**decode(encoding = 'UTF-8',errors = 'strict')** Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.

5	**encode(encoding = 'UTF-8',errors = 'strict')** Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.
6	**endswith(suffix, beg = 0, end = len(string))** Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
7	**expandtabs(tabsize = 8)** Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
8	**find(str, beg = 0 end = len(string))** Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
9	**index(str, beg = 0, end = len(string))** Same as find(), but raises an exception if str not found.
10	**isalnum()** Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
11	**isalpha()** Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
12	**isdigit()** Returns true if string contains only digits and false otherwise.
13	**islower()** Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
14	**isnumeric()** Returns true if a unicode string contains only numeric characters and false otherwise.
15	**isspace()** Returns true if string contains only whitespace characters and false otherwise.
16	**istitle()** Returns true if string is properly "titlecased" and false otherwise.
17	**isupper()**

	Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
18	**join(seq)** Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
19	**len(string)** Returns the length of the string
20	**ljust(width[, fillchar])** Returns a space-padded string with the original string left-justified to a total of width columns.
21	**lower()** Converts all uppercase letters in string to lowercase.
22	**lstrip()** Removes all leading whitespace in string.
23	**maketrans()** Returns a translation table to be used in translate function.
24	**max(str)** Returns the max alphabetical character from the string str.
25	**min(str)** Returns the min alphabetical character from the string str.
26	**replace(old, new [, max])** Replaces all occurrences of old in string with new or at most max occurrences if max given.
27	**rfind(str, beg = 0,end = len(string))** Same as find(), but search backwards in string.
28	**rindex(str, beg = 0, end = len(string))** Same as index(), but search backwards in string.
29	**rjust(width,[, fillchar])** Returns a space-padded string with the original string right-justified to a total of width columns.
30	**rstrip()**

	Removes all trailing whitespace of string.
31	**split(str="", num=string.count(str))** Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
32	**splitlines(num=string.count('\n'))** Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.
33	**startswith(str, beg=0,end=len(string))** Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.
34	**strip([chars])** Performs both lstrip() and rstrip() on string
35	**swapcase()** Inverts case for all letters in string.
36	**title()** Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.
37	**translate(table, deletechars="")** Translates string according to translation table str(256 chars), removing those in the del string.
38	**upper()**

	Converts lowercase letters in string to uppercase.
39	**zfill (width)** Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).
40	**isdecimal()** Returns true if a unicode string contains only decimal characters and false otherwise.