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 that 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 list 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).

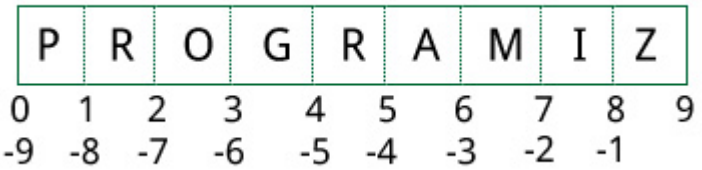
```
>>> my_list = ['p','r','o','g','r','a','m','i','z']
>>> my_list[2:5]    # elements 3rd to 5th
['o', 'g', 'r']

>>> 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.



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 multiplt 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]