List

What is a Python List?

Unlike C, C++ or Java, Python Programming Language doesn't have arrays. To hold a sequence of values, then, it provides the 'list' class. **A Python list can be seen as a collection of values.**
In short, a list is a collection of arbitrary objects, somewhat akin to an array in many other programming languages but more flexible.

How to Create a Python List?

Lists are defined in Python by enclosing a comma-separated sequence of objects in square brackets ([ ]), as shown below:

```
a = ['foo', 'bar', 'baz', 'qux']
>>> print(a)
['foo', 'bar', 'baz', 'qux']
>>> a
['foo', 'bar', 'baz', 'qux']
```

A list is not merely a collection of objects. It is an ordered collection of objects. The order in which you specify the elements when you define a list is an innate characteristic of that list and is maintained for that list's lifetime. (You will see a Python data type that is not ordered in the next tutorial on dictionaries.)
Lists that have the same elements in a different order are not the same:

```
>>> a = ['foo', 'bar', 'baz', 'qux']
>>> b = ['baz', 'qux', 'bar', 'foo']
>>> a == b
False
>>> a is b
False
>>> [1, 2, 3, 4] == [4, 1, 3, 2]
False
```

Lists Can Contain Arbitrary Objects

A **list can contain** any assortment of objects. The elements of a list can all be the **same type** Or the elements can be of **varying types**:

```
>>> a = [2, 4, 6, 8]
>>> a
[2, 4, 6, 8]
>>>
>>> a = [21.42, 'foobar', 3, 4, 'bark', False, 3.14159]
>>> a
[21.42, 'foobar', 3, 4, 'bark', False, 3.14159]
```

Lists can even **contain complex objects, like functions, classes, and modules**:

```
>>> int
<class 'int'>
>>> len
<built-in function len>
>>> def foo():
...    pass
...return
…
>>> foo
<function foo at 0x035B9030>
>>> import math
>>> math
<module 'math' (built-in)>

>>> a = [int, len, foo, math]
>>> a
[<class 'int'>, <built-in function len>, <function foo at
0x02CA2618>,
<module 'math' (built-in)>]
```

A **list can contain any number of objects**, from zero to as many as your computer’s memory will allow:

```
>>> a = []
>>> a
```

```
      [ ]

      >>> a = [ 'foo' ]
      >>> a
      ['foo']
```

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
...18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
...35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50]
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,37,
38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50]
```

(A list with a single object is sometimes referred to as a singleton list.)
List objects needn't be unique. A given object can appear in a list multiple times:

```
      >>>
      >>> a = ['bark', 'meow', 'woof', 'bark', 'cheep', 'bark']
      >>> a
      ['bark', 'meow', 'woof', 'bark', 'cheep', 'bark']
```

A list can also have another list as an item. This is called a nested list.

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

The important characteristics of Python lists are as follows:

- Lists are ordered.
- Lists can contain any arbitrary object.
- List elements can be accessed by index.
- Lists can be nested to arbitrary depth.
- Lists are mutable.
- Lists are dynamic.
- Each of these features is examined in more detail below.

How to access elements from a list?

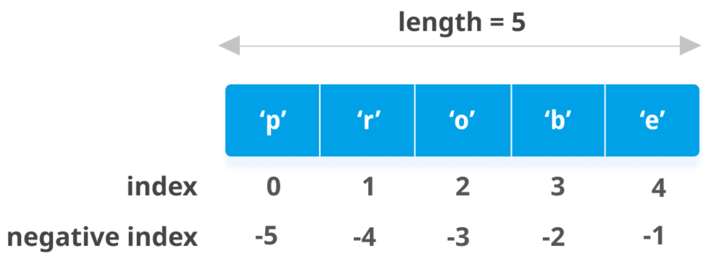
Individual elements in a list can be accessed using an index in square brackets. This is exactly analogous to accessing individual characters in a string. List indexing is zero-based as it is with strings.
Consider the following list:

```
>>>
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
```

The indices for the elements in a are shown below:Here is Python code to access some elements of list **a**:

```
>>>
>>> a[0]
'foo'
>>> a[2]
'baz'
>>> a[5]
'corge'
```

Virtually everything about string indexing works similarly for lists. For example, a negative list index counts from the end of the list:



Negative List Indexing

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

Slicing also works. If ***a*** is a list, the expression **a[m:n:p]** returns the **portion of a from index m to, but not including, index n with a gap of p index**:

```
>>> a[:-1]    # -5 to -1 with a gap of 1 (default)
['p', 'r', 'o', 'b']
>>> a[0:1:2]
['p']
>>> a[0:4:2]
['p', 'o']
>>> a[1:5:2]
['r', 'b']
>>> a[2:5:2]
['o', 'e']
>>>a[:-2:2]
['p', 'o']

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[-5:-2]
['bar', 'baz', 'qux']
>>> a[1:4]
['bar', 'baz', 'qux']
>>> a[-5:-2] == a[1:4]
True
```

Omitting the first index starts the slice at the beginning of the list, and omitting the second index extends the slice to the end of the list:

```
>>> print(a[:4], a[0:4])
['foo', 'bar', 'baz', 'qux'] ['foo', 'bar', 'baz', 'qux']
>>> print(a[2:], a[2:len(a)])
['baz', 'qux', 'quux', 'corge'] ['baz', 'qux', 'quux', 'corge']

>>> a[:4] + a[4:]
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[:4] + a[4:] == a
True
```

Modifying Multiple List Values

What if you want to change several contiguous elements in a list at one time? Python allows this with slice assignment, which has the following syntax:
a[m:n] = <iterable>

Again, for the moment, think of an iterable as a list. This **assignment replaces the specified slice** of a with <iterable>:

```
    >>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
    >>> a[1:4]
    ['bar', 'baz', 'qux']
    >>> a[1:4] = [1.1, 2.2, 3.3, 4.4, 5.5]
    >>> a
    ['foo', 1.1, 2.2, 3.3, 4.4, 5.5, 'quux', 'corge']
    >>> a[1:6]
    [1.1, 2.2, 3.3, 4.4, 5.5]
    >>> a[1:6] = ['Bark!']
    >>> a
['foo', 'Bark!', 'quux', 'corge']
```

The **number of elements inserted need not be equal to the number replaced**. Python just grows or shrinks the list as needed.
You can insert multiple elements in place of a single element—just use a slice that denotes only one element:

```
    >>> a = [1, 2, 3]
    >>> a[1:2] = [2.1, 2.2, 2.3]
    >>> a
    [1, 2.1, 2.2, 2.3, 3]
```

Note that this is not the same as replacing the single element with a list:

```
    >>> a = [1, 2, 3]
    >>> a[1] = [2.1, 2.2, 2.3]
    >>> a
    [1, [2.1, 2.2, 2.3], 3]
```

You can also **insert elements into a list** without removing anything. Simply specify a slice of the form [n:n] (a zero-length slice) at the desired index:

```
>>> a = [1, 2, 7, 8]
>>> a[2:2] = [3, 4, 5, 6]
>>> a
[1, 2, 3, 4, 5, 6, 7, 8]
```

You can **delete multiple elements** out of the middle of a list by assigning the appropriate slice to an empty list. You can also use the del statement with the same slice:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[1:5] = []
>>> a
['foo', 'corge']
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> del a[1:5]
>>> a
['foo', 'corge']
```

Prepending or **Appending Items** to a List

Additional items can be added to the start or end of a list using the + concatenation operator or the += augmented assignment operator:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a += ['grault', 'garply']
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'grault', 'garply']

>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[len(a):len(a)] = [1, 2, 7, 8]
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 1, 2, 3]
```

Note that a list must be concatenated with another list, so if you want to add only one element, you need to specify it as a singleton list:

```
>>>
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a += 20
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    a += 20
TypeError: 'int' object is not iterable

>>> a += [20]
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 20]
```

Note: Technically, it isn't quite correct to say a list must be concatenated with another list. More precisely, a list must be concatenated with an object that is iterable. Of course, lists are iterable, so it works to concatenate a list with another list.
Strings are iterable also. But watch what happens when you concatenate a string onto a list:

```
>>>
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux']
>>> a += 'corge'
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'c', 'o', 'r', 'g', 'e']
```

This result is perhaps not quite what you expected. When a string is iterated through, the result is a list of its component characters. In the above example, what gets concatenated onto list a is a list of the characters in the string 'corge'.
If you really want to add just the single string 'corge' to the end of the list, you need to specify it as a singleton list:

```
>>>
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux']
>>> a += ['corge']
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
```

Operator on List

Several Python operators and built-in functions can also be used with lists in ways that are analogous to strings:

The in and not in operators:

```
      >>> a
```

```
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> 'qux' in a
True
>>> 'thud' not in a
True
```

The concatenation (+) and replication (*) operators:

```
>>>
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> a + ['grault', 'garply']
['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'grault', 'garply']
>>> a * 2
['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'foo', 'bar',
'baz', 'qux', 'quux', 'corge']
```

It’s not an accident that strings and lists behave so similarly. They are both special cases of a more general object type called an iterable, which you will encounter in more detail in the upcoming tutorial on definite iteration.

By the way, in each example above, the list is always assigned to a variable before an operation is performed on it. But you can operate on a list literal as well:

```
>>> ['foo', 'bar', 'baz', 'qux', 'quux', 'corge'][2]
'baz'

>>> ['foo', 'bar', 'baz', 'qux', 'quux', 'corge'][::-1]
['corge', 'quux', 'qux', 'baz', 'bar', 'foo']

>>> 'quux' in ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
True

>>> ['foo', 'bar', 'baz'] + ['qux', 'quux', 'corge']
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
```

```
>>> len(['foo', 'bar', 'baz', 'qux', 'quux', 'corge'][::-1])
6
```

A list item can be deleted with the del command:

```
>>>
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> del a[3]
>>> a
['foo', 'bar', 'baz', 'quux', 'corge']
```

Built-in List Functions

There are some built-in functions in Python that you can use on python lists.

```
even=[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

len(): It calculates the length of the list.

```
>>> len(even)
10
```

max(): It returns the item from the list with the highest value.

```
>>> max(even)
20
```

If all the items in your list are strings, it will compare.

```
>>> max(['1','2','3'])
‘3’
```

But it fails when some are numeric, and some are strings in python.

```
>>> max([2,'1','2'])
Error: Traceback (most recent call last)
```

min(): It returns the item from the Python list with the lowest value.

```
>>> min(even)
```

```
2
```

sum(): It returns the sum of all the elements in the list.

```
>>> sum(even)
110
>>> sum([1.1,2.2,3.3])
6.6
```

However, for this, the Python list must hold all numeric (integer/float) values. If some of them are character/string it will fail

```
>>> a=['1','2','3']
>>> sum(a)
Error will occur
```

sorted(): It returns a sorted version of the list, but does not change the original one.

```
>>> a=[3,1,2]
>>> sorted(a)
[1, 2, 3]
>>> a
[3, 1, 2]
```

If the Python list members are strings, it sorts them according to their ASCII values.

```
>>> sorted(['hello','hell','Hello'])
['Hello', 'hell', 'hello']
```

Here, since H has an ASCII value of 72, it appears first.

list(): It converts a different data type into a list.

```
>>> list("abc")
['a', 'b', 'c']
```

It can't convert a single int into a list, though, it only converts iterables.

```
>>> list(2)
```

```
      Error will occur
```

any(): It returns True if even one item in the Python list has a True value.

```
      >>> any(['','','1'])
      True
```

It returns False for an empty iterable.

```
      >>> any([])
      False
```

all(): It returns True if all items in the list have a True value.

```
      >>> all(['','','1'])
      False
```

It returns True for an empty iterable.

```
      >>> all([])
      True
```

Built-in Methods

While a function is what you can apply on a construct and get a result, a method is what you can do to it and change it. To call a method on a construct, you use the dot-operator(.). Python supports some built-in methods to alter a Python list.

append()

append(<obj>) appends object <obj> to the end of list a:

```
      >>> a = ['a', 'b']
      >>> a.append(123)
      >>> a
      ['a', 'b', 123]
```

Remember, list methods modify the target list in place. **They do not return anything**:

```
>>> a = ['a', 'b']
>>> x = a.append(123)
>>> print(x)
None
>>> a
['a', 'b', 123]
```

Remember that **when the + operator is used to concatenate to a list, if the target operand is an iterable, then its elements are broken out and appended to the list** individually:

```
>>> a = ['a', 'b']
>>> a + [1, 2, 3]
['a', 'b', 1, 2, 3]
```

The append() method does not work that way! If an iterable is appended to a list with append(), it is added as a single object:

```
>>> a = ['a', 'b']
>>> a.append([1, 2, 3])
>>> a
['a', 'b', [1, 2, 3]]
```

Thus, with append(), you can append a string as a single entity:

```
>>> a = ['a', 'b']
>>> a.append('foo')
>>> a
['a', 'b', 'foo']
```

extend(<iterable>)

Extends a list with the objects from an iterable.

Yes, this is probably what you think it is. extend() also adds to the end of a list, but the argument is expected to be iterable. The items in <iterable> are added individually:

```
>>> a = ['a', 'b']
>>> a.extend([1, 2, 3])
```

```
    >>> a
    ['a', 'b', 1, 2, 3]
```

In other words, .extend() behaves like the + operator. More precisely, since it modifies the list in place, it behaves like the += operator:

```
    >>> a = ['a', 'b']
    >>> a += [1, 2, 3]
    >>> a
    ['a', 'b', 1, 2, 3]
```

insert(<index>, <obj>)

Inserts an object into a list.

a.insert(<index>, <obj>) inserts object <obj> into list **a** at the specified <index>. Following the method call, a[<index>] is <obj>, and the remaining list elements are pushed to the right:

```
>>>
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a.insert(3, 3.14159)
>>> a[3]
3.14159
>>> a
['foo', 'bar', 'baz', 3.14159, 'qux', 'quux', 'corge']
```

remove(<obj>)

Removes an object from a list.

a.remove(<obj>) removes object <obj> from list a.

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a.remove('baz')
>>> a
['foo', 'bar', 'qux', 'quux', 'corge']
```

If <obj> isn’t in a, an exception is raised:

```
>>> a.remove('Bark!')
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
      a.remove('Bark!')
ValueError: list.remove(x): x not in list
```

If the object is present multiple times, only the 1st occurrence is removed.

```
>>>b=['a','b',[1, 2],'foo',1,2,3,'f','o','o',4,5,6,1,2,3]
>>> b.remove(2)
>>> b
['a', 'b', [1, 2], 'foo', 1, 3, 'f', 'o', 'o', 4, 5, 6, 1, 2, 3]
```

pop(index=-1)

Removes an element from a list. This method **differs from remove() in two ways:**
You specify the **index of the item to remove**, rather than the object itself. **The method returns a value**: the item that was removed.
a.pop() simply removes the last item in the list:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a.pop()
'corge'
>>> a
['foo', 'bar', 'baz', 'qux', 'quux']

>>> a.pop()
'quux'
>>> a
['foo', 'bar', 'baz', 'qux']
```

If the optional <index> parameter is specified, the item at that index is removed and returned. <index> may be negative, as with string and list indexing:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a.pop(1)
'bar'
>>> a
['foo', 'baz', 'qux', 'quux', 'corge']
>>> a.pop(-3)
'qux'
>>> a
['foo', 'baz', 'quux', 'corge']
```

<index> defaults to -1, so a.pop(-1) is equivalent to a.pop().

reverse()

It will will reverse the list in place

For example, if you have a list [1, 3, 4, 5] and you need to reverse it, you can call the reverse method.

```
>>> # initialise a list of numbers that
 >>> some_numbers = [1, 3, 4, 5]

 >>> # Try to reverse the list now
 >>> some_numbers.reverse()

 >>> # print the list to check if it is really reversed.
 >>> print(some_numbers)
 [5, 4, 3, 1]
```

One and Multi Dimensional List/Array

Python provides many ways to create 2-dimensional lists/arrays. However one must know the differences between these ways because they can create complications in code that can be very difficult to trace out. Lets start by looking at common ways of creating 1d array of size N initialized with 0s.

```
# First method to create a 1 D array
    >>> N = 5
    >>> arr = [0]*N
    >>> print(arr)
```
Output:
```
    [0, 0, 0, 0, 0]
```
Or we can use
```
# Second method to create a 1 D array
    >>> N = 5
    >>> arr = [0 for i in range(N)]
    >>> print(arr)
```

We can access array element or modify them
```
    >>> arr[3]=12
    >>> arr
```
Output: [0, 0, 0, 12, 0]
```
    >>> arr[1]=11
    >>> print(arr)
```
Output: [0, 11, 0, 12, 0]

Extending the above we can define 2-dimensional arrays in the following ways.

```
# Using above first method to create a 2D array
    >>> rows, cols = (5, 5)
    >>> arr = [[0]*cols]*rows  #arr = [[0]*rows]*cols
    >>> print(arr)
```

Output: `[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]`

```
        >>> arr[0][1]=11
```
Output: [[0, 11, 0, 0, 0], [0, 11, 0, 0, 0], [0, 11, 0, 0, 0], [0, 11, 0, 0, 0], [0, 11, 0, 0, 0]]

```
 # Using above second method to create a 2D array
      rows, cols = (5, 5)
      arr = [[0 for i in range(cols)] for j in range(rows)]
      print(arr)
```

Output:

```
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0],
[0, 0, 0, 0, 0]]
```

```
     >>> arr[0][1]=11

     >>> print(arr)
```

Output:

```
[[0, 11, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0],
[0, 0, 0, 0, 0]]
```

```
     rows, cols = (5, 5)
      arr=[]
      for i in range(cols):
           col = []
           for j in range(rows):
              col.append(0)
           arr.append(col)
     print(arr)
```

Output:

```
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0,
0, 0], [0, 0, 0, 0, 0]]
```

```
     >>> arr[1][2]=22

     >>> print(arr)
```

Output:

```
[[0, 0, 22, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0],
[0, 0, 0, 0, 0]]
```

Reading values in list

Using append( )

```
lst=[]
m=int(input('Enter value of m :'))
for i in range(m):
     val=int(input('Enter value  : '))
     lst.append(val)
print('Entered list is ',lst)
```

Using insert()

```
lst=[]
m=int(input("How many number you want to read : "))
for i in range(m):
    val=float(input("Enter value : "))
    lst.insert(i,val)
print("Values in the list is ",lst)
for i in range(m):
    print("Value[",i,"]= ",lst[i])
```

Removing duplicate values from list

```
#removing duplicate
lst=[]
remdup=[]
m=int(input("How many number you want to read : "))
for i in range(m):         # read values in list
    val=float(input("Enter value : "))
    lst.insert(i,val)
for val in lst:          # Remove Duplicates
    if val not in remdup:
       remdup.append(val)

print("After removing duplicates list is : ",remdup)
```

Operation 2D Array

```
row=int(input("Number of row : "))
col=int(input("Number of Col : "))

#initialize array
myarr=[[0 for i in range(col)] for j in range(row)]

#print initial array
print(myarr)

# Read array element
for i1 in range(row):
    for j1 in range(col):
        myarr[i1][j1]=float(input("Enter Value : "))

print("after Reading Value :")
for i in range(row):
    for j in range(col):
        print(myarr[i][j], end=" ")
    print("\n")
```