LIST OPERATION IN PYTHON

List is a sequence

- Like a string, a **list is a sequence of values.** In a string, the values are characters; in a list, they can be any type.
- The values in list are called **elements** or sometimes **items.**
- There are several ways to create a new list; the simplest is to enclose the elements in square brackets
- ([and]):

[10, 20, 30, 40]

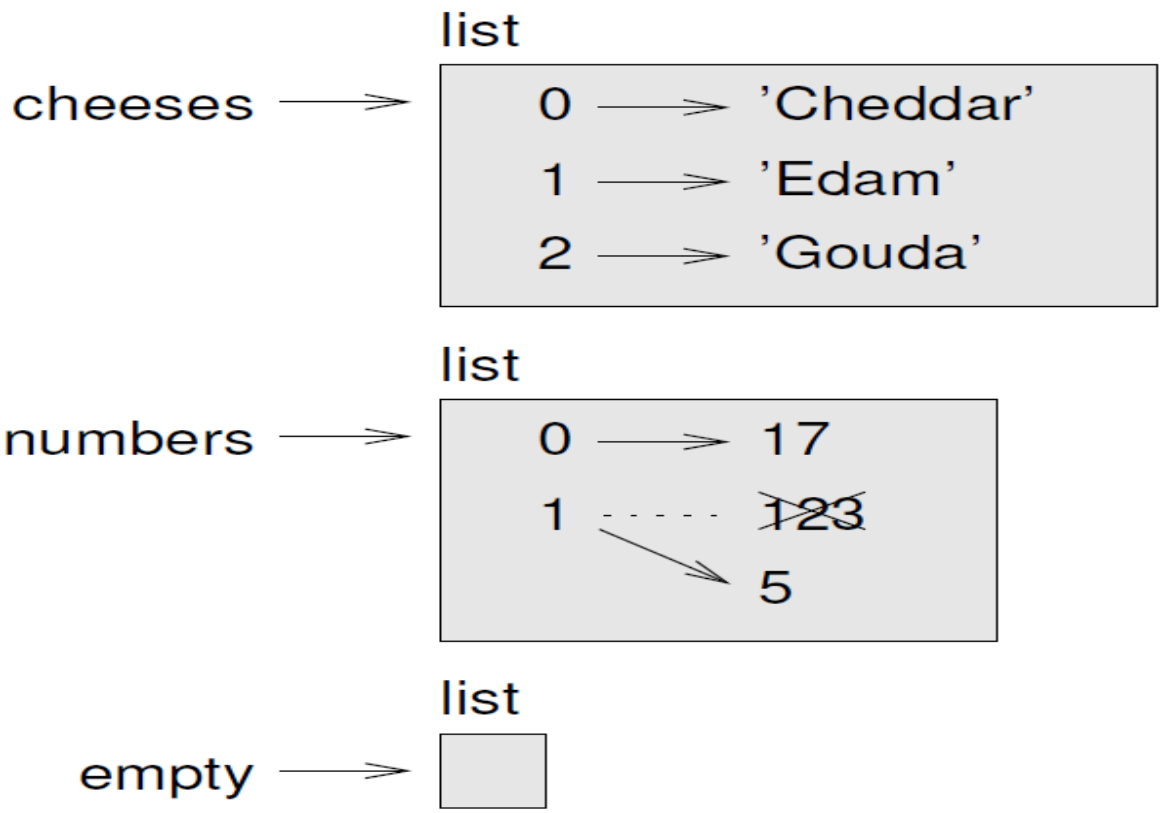
List is a sequence

- A list within another list is **nested.**

['spam', 2.0, 5, [10, 20]]

- A list that contains no elements is called an empty list; you can create one with empty brackets, [].
- As you might expect, you can assign list values to variables:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print (cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```


list
cheeses
0
'Cheddar'
1
'Edam'
2
'Gouda'
list
numbers
0
17
1
123
5
list
empty

Lists are mutable

- The syntax for accessing the elements of a list is the same as for accessing the characters of a string—the bracket operator.
- The expression inside the brackets specifies the index. Remember that the indices start at 0.
- Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print numbers
    [17, 5]
```

- The one-th element of numbers, which used to be 123, is now 5.

- List indices work the same way as string indices:
  - Any integer expression can be used as an index.
  - If you try to read or write an element that does not exist, you get an IndexError.
  - If an index has a negative value, it counts backward from the end of the list.

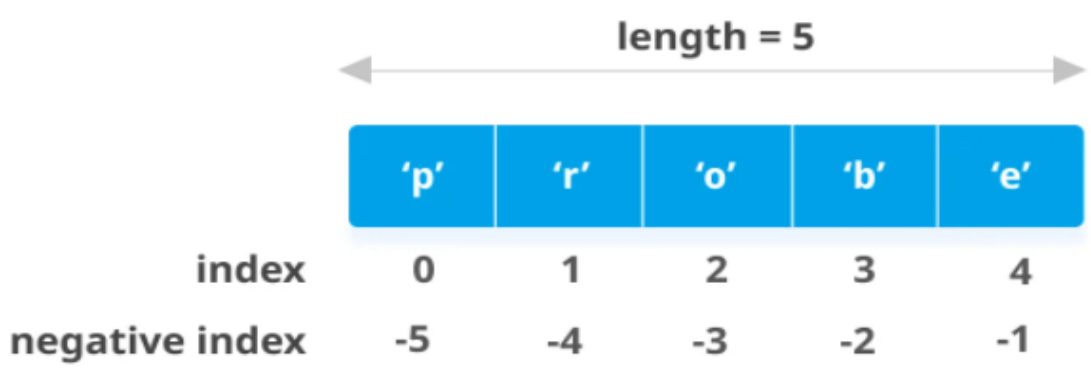
```
# Negative indexing in lists
my_list = ['p','r','o','b','e']

# last item
print(my_list[-1])

# fifth last item
print(my_list[-5])
```

Output

```
e
p
```



Traversing a list

- The **in** operator also works on lists.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
```

- The most common way to traverse the elements of a list is with for loop. The syntax is the same as for strings:

```
>>> for cheese in cheeses:
        print cheese

>>> for i in range(len(numbers)):
        numbers[i] = numbers[i] * 2
```

List operations

- The + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

- Similarly, the * operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

List slices

- The slice operator also works on lists:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

- Since lists are mutable, it is often useful to make a copy before performing operations.

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print t          Here
['a', 'x', 'y', 'd', 'e', 'f']
>>> t[1:2] = ['x', 'y']
>>> t
['a', 'x', 'y', 'y', 'd', 'e', 'f']
```

List methods

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print t
['a', 'b', 'c', 'd']
```

```
x = [1, 2, 3]
x.append([4, 5])
print(x)
[1, 2, 3, [4, 5]]
```

- extend takes a list as an argument and appends all of the elements:

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

```
x = [1, 2, 3]
x.extend([4, 5])
print(x)
[1, 2, 3, 4, 5]
```

- 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.

```
# Demonstration of list insert() method
odd = [1, 9]
odd.insert(1,3)

print(odd)

odd[2:2] = [5, 7]

print(odd)
```

Output

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

- sort arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print (t)
['a', 'b', 'c', 'd', 'e']
```

- List methods are all void; they modify the list and **return None**. If you accidentally write **t = t.sort()**, you will be disappointed with the result.

Deleting elements

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print t
['a', 'c']
>>> print x
b
```

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print t
['a', 'c']
```

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print t
['a', 'c']
```

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print t
['a', 'f']
```

Map, filter and reduce

```
def sum (t):
    total = 0
    for x in t:
        total += x    # total= total +x
    return total
```

- Adding up the elements of a list is such a common operation that Python provides it as a built-in function, sum:

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

- An operation like this that combines a sequence of elements into a single value is sometimes called **reduce.**

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

- res is initialized with an empty list; each time through the loop, we append the next element.
- So res is another kind of accumulator.
- An operation like capitalize_all is sometimes called a **map** because it "maps" a function (in this case the method capitalize) onto each of the elements in a sequence.

```
def only_upper(t):
    res = []
    for s in t:
      if s.isupper():
          res.append(s)
    return res
```

- isupper is a string method that returns True if the string contains only upper case letters.
- An operation like only_upper is called a **filter** because it selects some of the elements and filters out the others.

Python List Methods

Methods	Descriptions
append()	adds an element to the end of the list
extend()	adds all elements of a list to another list
insert()	inserts an item at the defined index
remove()	removes an item from the list
pop()	returns and removes an element at the given index
clear()	removes all items from the list
index()	returns the index of the first matched item
count()	returns the count of the number of items passed as an argument
sort()	sort items in a list in ascending order
reverse()	reverse the order of items in the list
copy()	returns a shallow copy of the list

Lists and strings

- A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string.
- To convert from a string to a list of characters, you can use list:

```
>>> s = 'spam'
>>> t = list(s)
>>> print t
['s', 'p', 'a', 'm']

>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print t
['pining', 'for', 'the', 'fjords']
```

- An optional argument called a **delimiter** specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

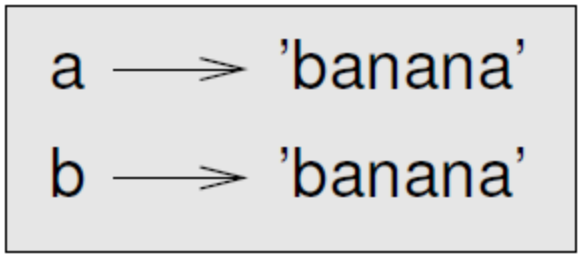
```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

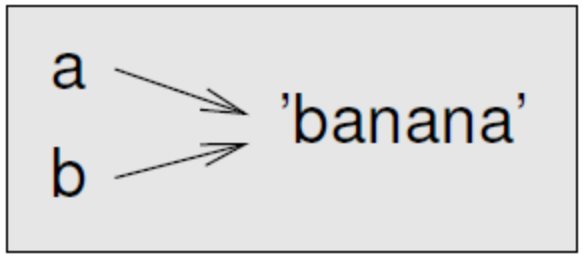
- **join is the inverse of split.** It takes a list of strings and concatenates the elements. join is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

Objects and values

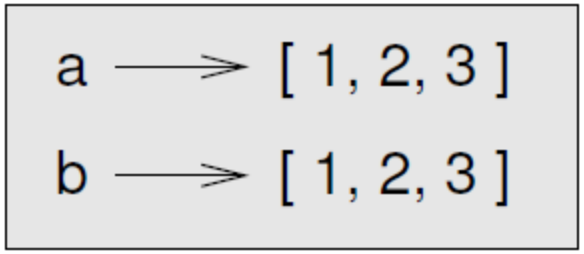
- a = 'banana'
- b = 'banana‘
- We know that a and b both refer to a string, but we don’t know whether they refer to the *same string*.
- There are two possible states:





```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True

>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```



Aliasing

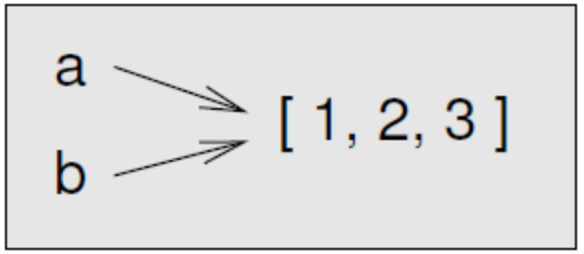
- If a refers to an object and you assign b = a, then both variables refer to the same object:

>>> a = [1, 2, 3]

>>> b = a

>>> b is a

True



- The association of a variable with an object is called a **reference.** In this example, there are two references to the same object.
- An object with more than one reference has more than one name, so we say that the object is **aliased.**
- If the aliased object is mutable, changes made with one alias affect the other:

>>> b[0] = 17

>>> print a

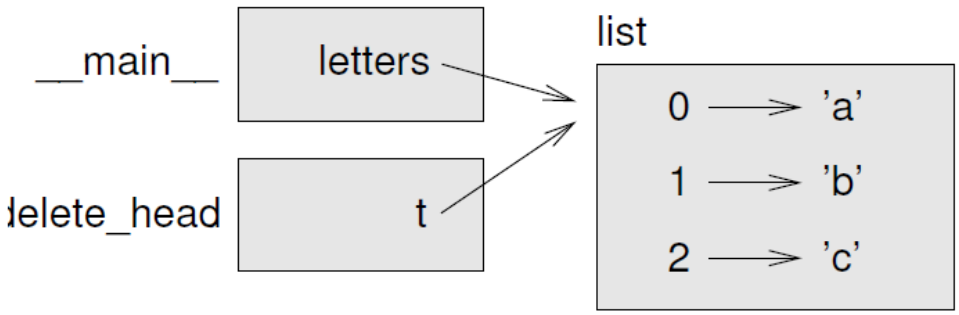
[17, 2, 3]

List arguments

```
def delete_head(t):
    del t[0]
```

- Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print letters
['b', 'c']
```



- It is important to distinguish between operations that modify lists and operations that create new lists.
- For example, the append method modifies a list, but the + operator creates a new list:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print t1
[1, 2, 3]
>>> print t2
None

>>> t3 = t1 + [3]
>>> print t3
[1, 2, 3]
```

This difference is important when you write functions that are supposed to modify lists.

List Comprehension: Elegant way to create Lists

```
pow2 = [2 ** x for x in range(10)]
print(pow2)
```

Output

```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

This code is equivalent to:

```
pow2 = []
for x in range(10):
    pow2.append(2 ** x)
```

Built-in Operations

Sno.	Function	Description	Example
1	cmp(list1, list2)	It compares the elements of both the lists.	This method is not used in the Python 3 and the above versions.
2	len(list)	It is used to calculate the length of the list.	L1 = [1,2,3,4,5,6,7,8] print(len(L1)) 8
3	max(list)	It returns the maximum element of the list.	L1 = [12,34,26,48,72] print(max(L1)) 72
4	min(list)	It returns the minimum element of the list.	L1 = [12,34,26,48,72] print(min(L1)) 12
5	list(seq)	It converts any sequence to the list.	str = "Johnson" s = list(str) print(type(s)) <class list>