Introduction to Computer Science

Container Types

鄒年棣 (Nien-Ti Tsou)

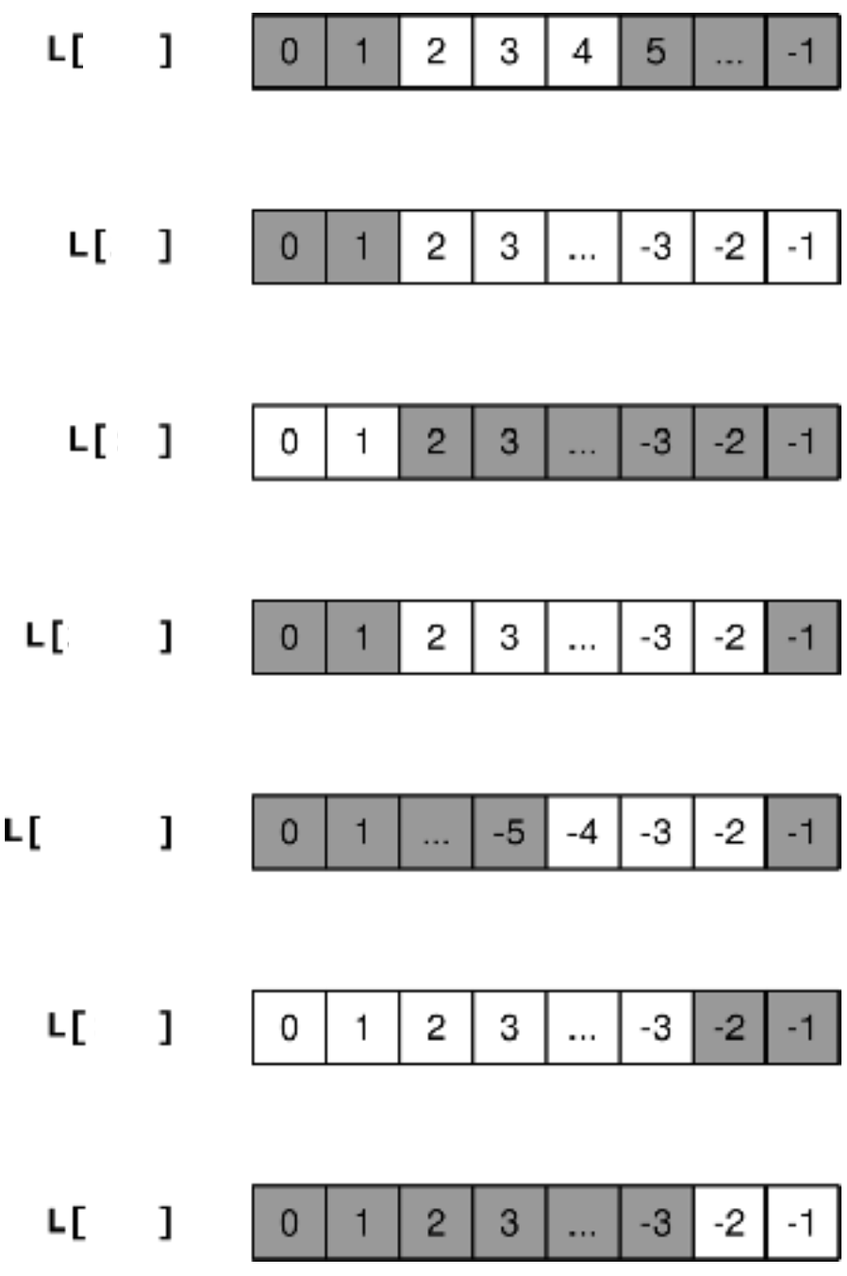
Container types

- Container types are used to group objects together.
- The main difference between the different container types is the way individual elements are accessed and how operations are defined.
- We will introduce
  - Lists
  - Tuples
  - Dictionaries
  - Sets

More about lists

- A list of objects of any kind. The individual objects are enumerated by assigning each element an index. The first element in the list gets index 0. (Consider the usual indexing of coefficients of a polynomial)

```
M = [3,['a', -3.0, 5]]
print(M[1])
print(M[1][2])
```



- Slicing a list `L[i:j]` means create a list by taking all elements from `L` starting at `L[i]` until `L[j-1]`.

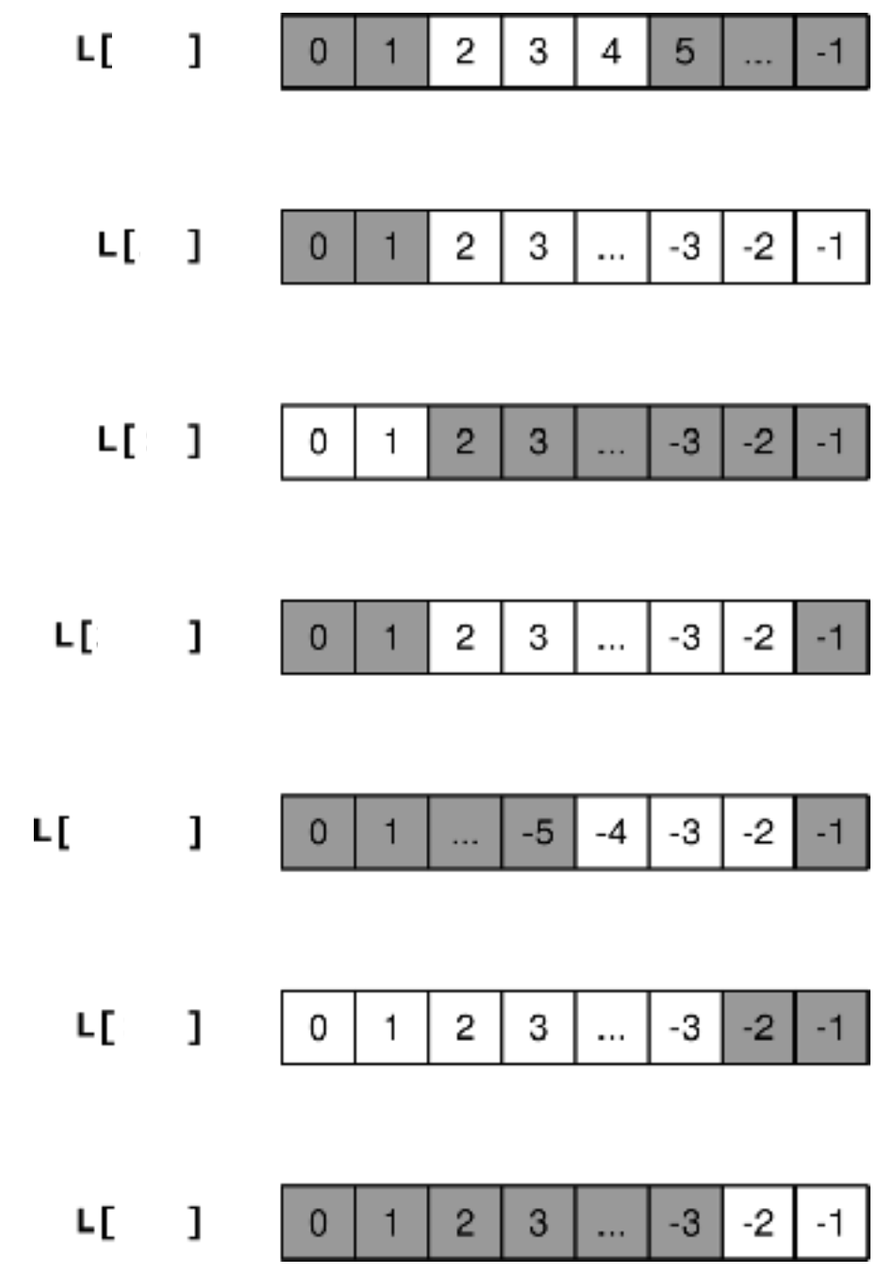
More about lists

- A list of objects of any kind. The individual objects are enumerated by assigning each element an index. The first element in the list gets index 0. (Consider the usual indexing of coefficients of a polynomial)

```
M = [3,['a', -3.0, 5]]
print(M[1])
print(M[1][2])
```

```
['a', -3.0, 5]
```

- Slicing a list `L[i:j]` means create a list by taking all elements from `L` starting at `L[i]` until `L[j-1]`.



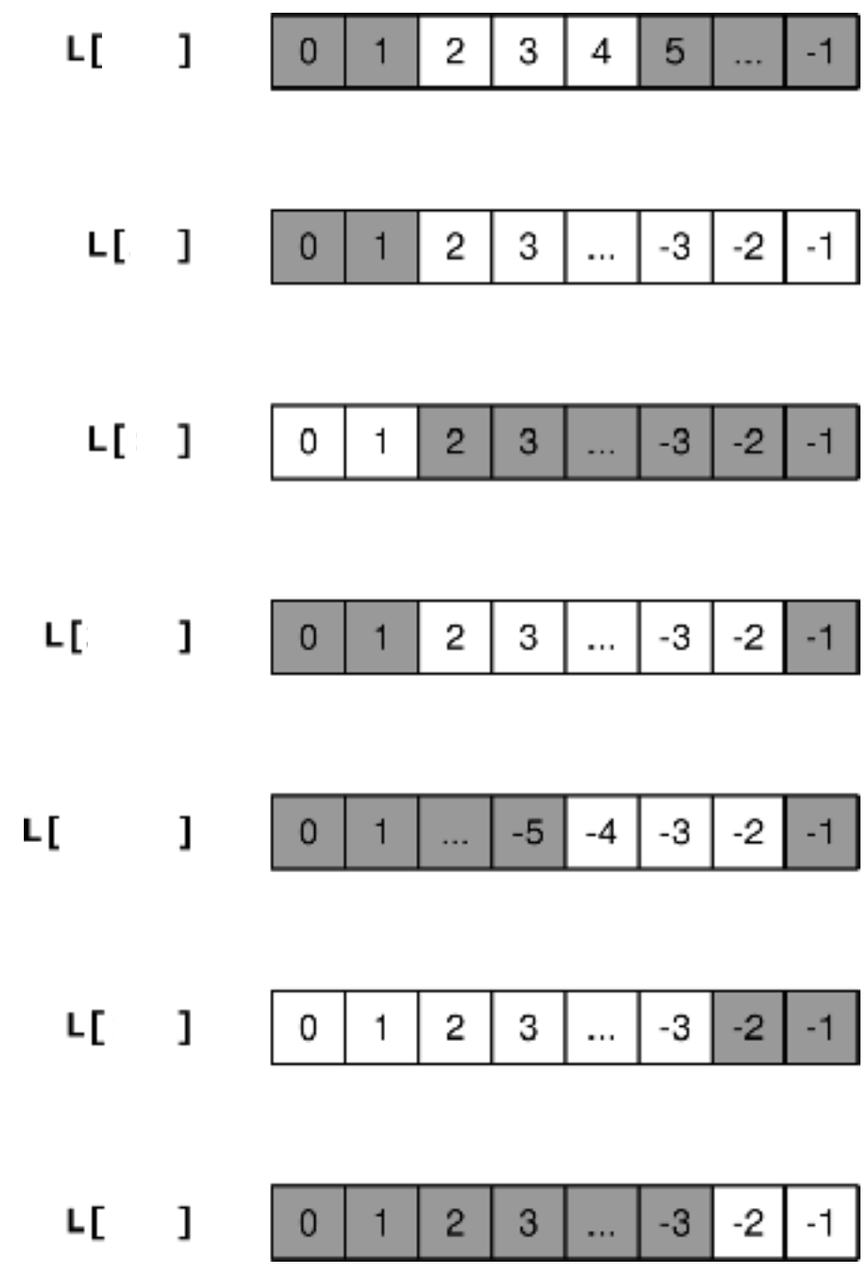
More about lists

- A list of objects of any kind. The individual objects are enumerated by assigning each element an index. The first element in the list gets index 0. (Consider the usual indexing of coefficients of a polynomial)

```
M = [3,['a', -3.0, 5]]
print(M[1])
print(M[1][2])
```

```
['a', -3.0, 5]
5
```

- Slicing a list `L[i:j]` means create a list by taking all elements from `L` starting at `L[i]` until `L[j-1]`.



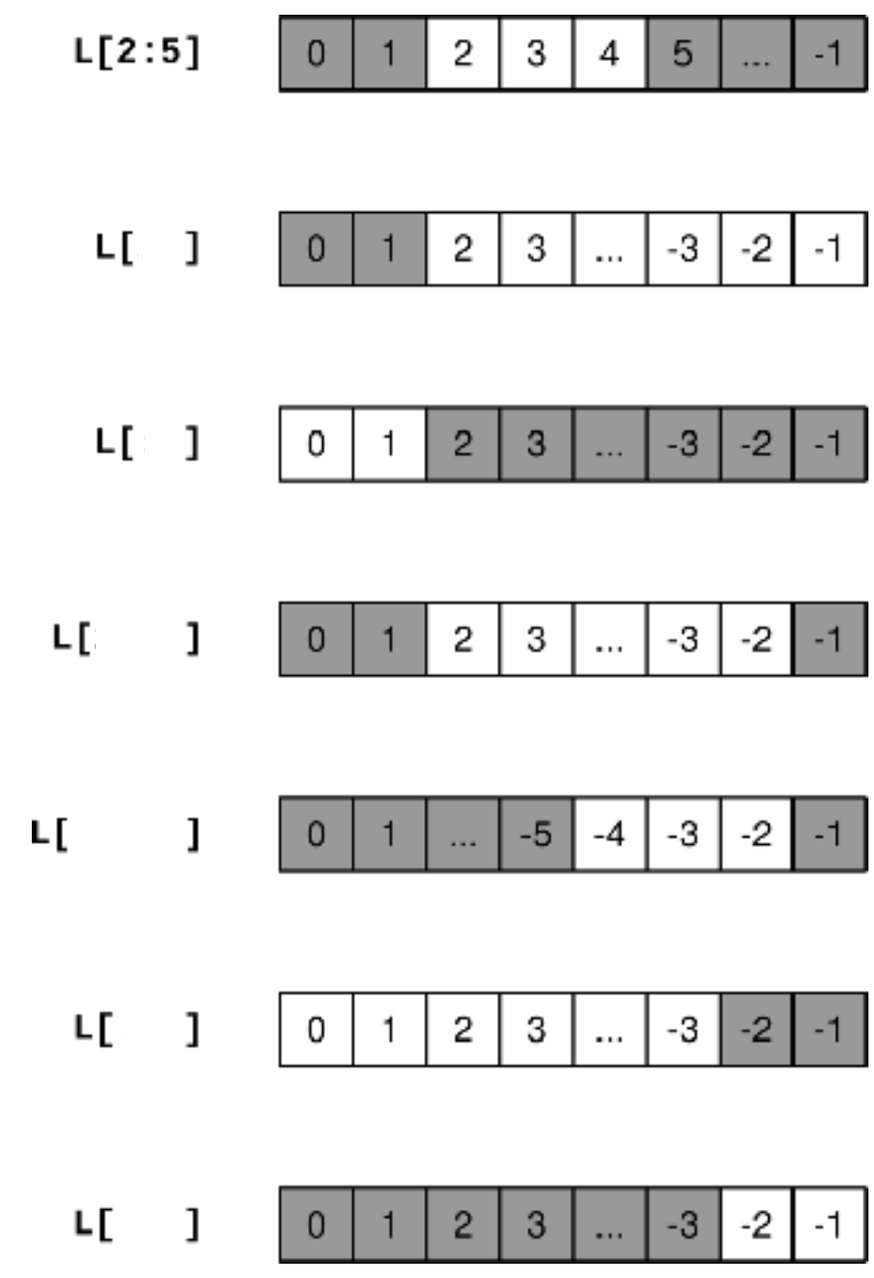
More about lists

- A list of objects of any kind. The individual objects are enumerated by assigning each element an index. The first element in the list gets index 0. (Consider the usual indexing of coefficients of a polynomial)

```
M = [3,['a', -3.0, 5]]
print(M[1])
print(M[1][2])
```

```
['a', -3.0, 5]
5
```

- Slicing a list `L[i:j]` means create a list by taking all elements from `L` starting at `L[i]` until `L[j-1]`.



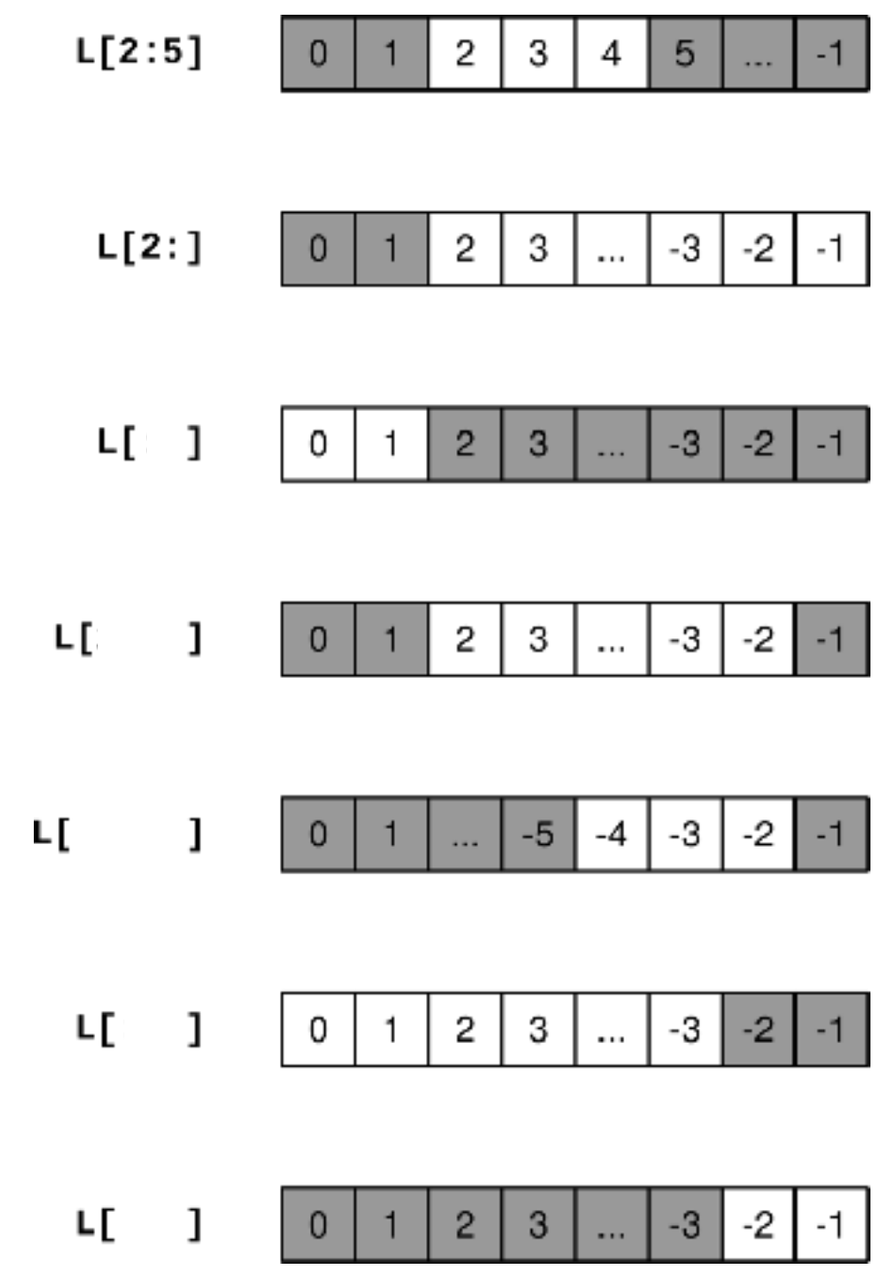
More about lists

- A list of objects of any kind. The individual objects are enumerated by assigning each element an index. The first element in the list gets index 0. (Consider the usual indexing of coefficients of a polynomial)

```
M = [3,['a', -3.0, 5]]
print(M[1])
print(M[1][2])
```

```
['a', -3.0, 5]
5
```

- Slicing a list `L[i:j]` means create a list by taking all elements from `L` starting at `L[i]` until `L[j-1]`.



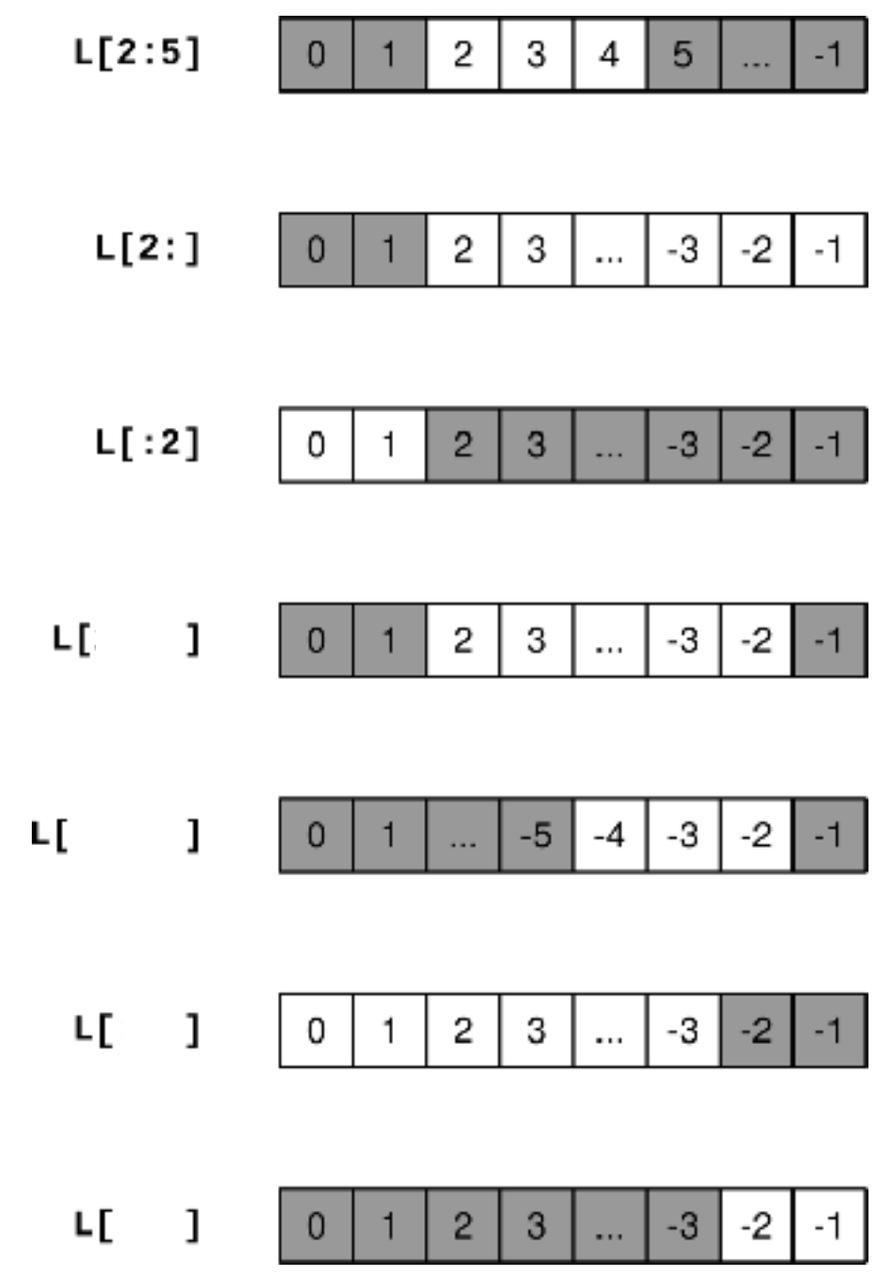
More about lists

- A list of objects of any kind. The individual objects are enumerated by assigning each element an index. The first element in the list gets index 0. (Consider the usual indexing of coefficients of a polynomial)

```
M = [3,['a', -3.0, 5]]
print(M[1])
print(M[1][2])
```

```
['a', -3.0, 5]
5
```

- Slicing a list `L[i:j]` means create a list by taking all elements from `L` starting at `L[i]` until `L[j-1]`.



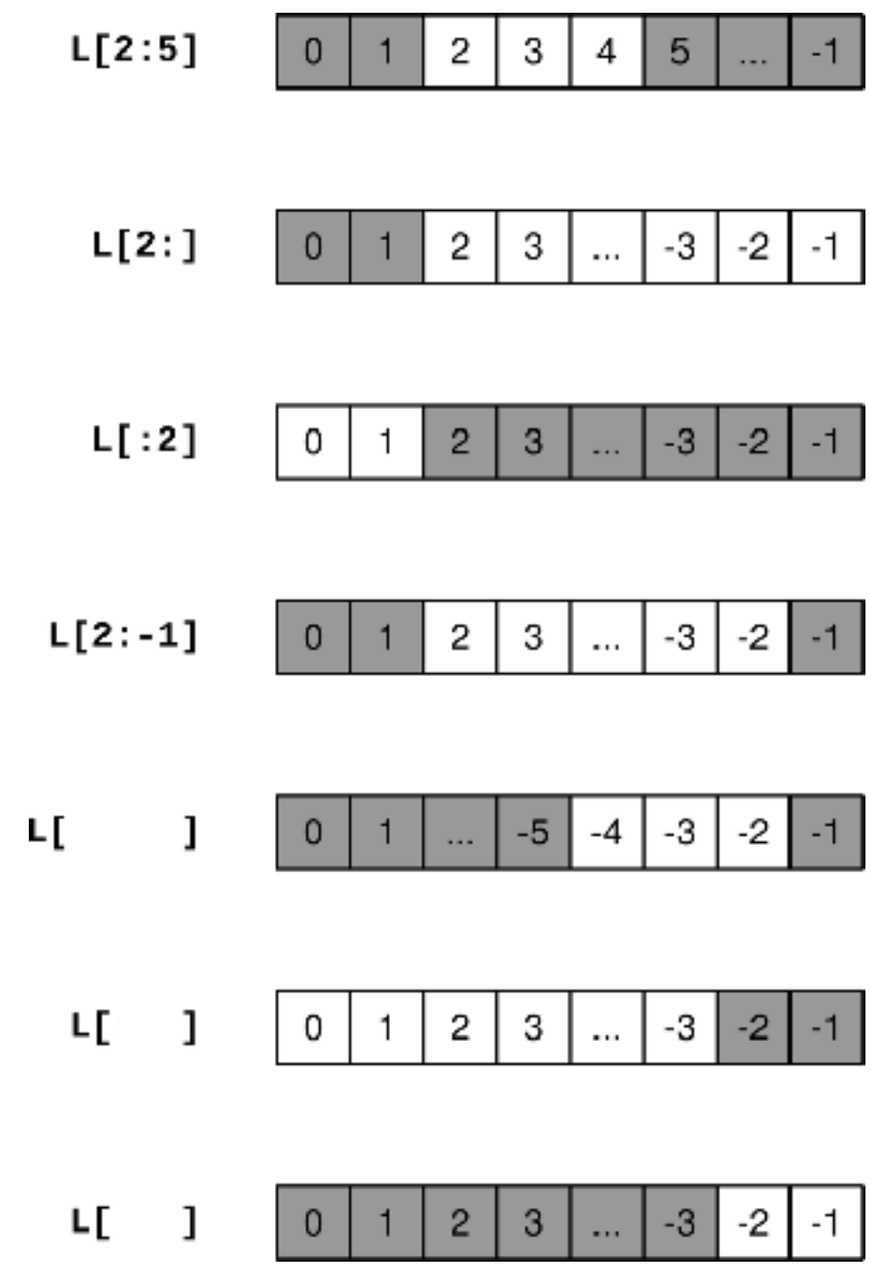
More about lists

- A list of objects of any kind. The individual objects are enumerated by assigning each element an index. The first element in the list gets index 0. (Consider the usual indexing of coefficients of a polynomial)

```
M = [3,['a', -3.0, 5]]
print(M[1])
print(M[1][2])
```

```
['a', -3.0, 5]
5
```

- Slicing a list `L[i:j]` means create a list by taking all elements from `L` starting at `L[i]` until `L[j-1]`.



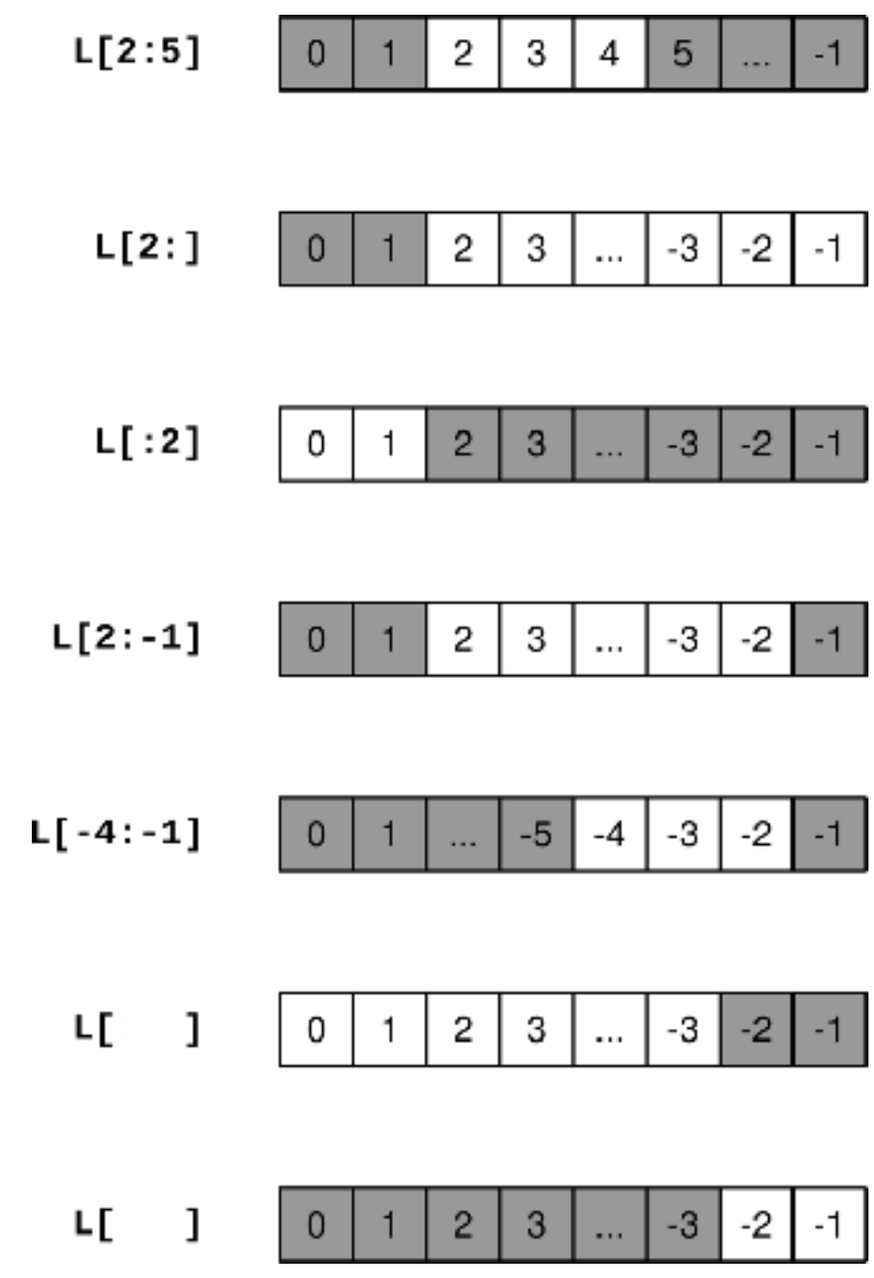
More about lists

- A list of objects of any kind. The individual objects are enumerated by assigning each element an index. The first element in the list gets index 0. (Consider the usual indexing of coefficients of a polynomial)

```
M = [3,['a', -3.0, 5]]
print(M[1])
print(M[1][2])
```

```
['a', -3.0, 5]
5
```

- Slicing a list `L[i:j]` means create a list by taking all elements from `L` starting at `L[i]` until `L[j-1]`.



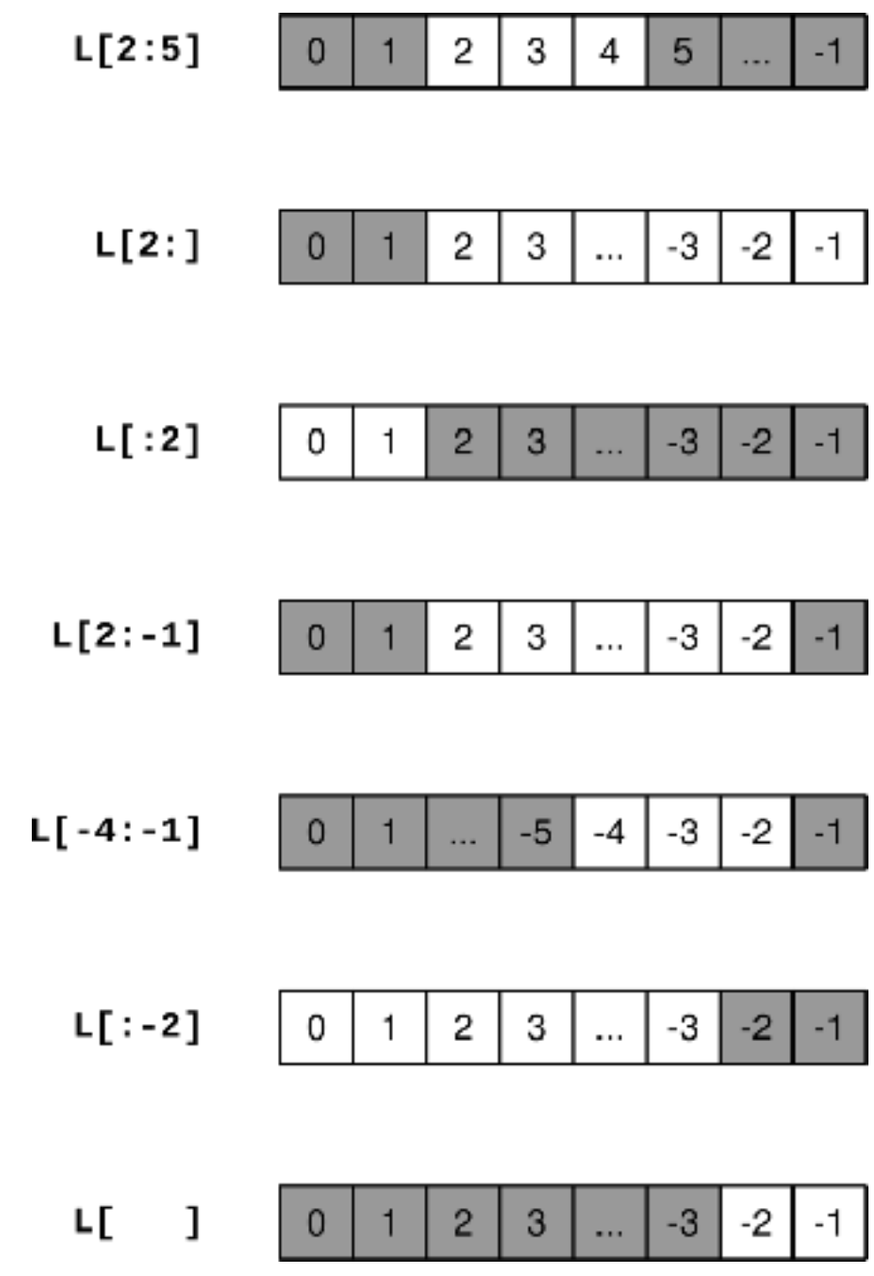
More about lists

- A list of objects of any kind. The individual objects are enumerated by assigning each element an index. The first element in the list gets index 0. (Consider the usual indexing of coefficients of a polynomial)

```
M = [3,['a', -3.0, 5]]
print(M[1])
print(M[1][2])
```

```
['a', -3.0, 5]
5
```

- Slicing a list `L[i:j]` means create a list by taking all elements from `L` starting at `L[i]` until `L[j-1]`.



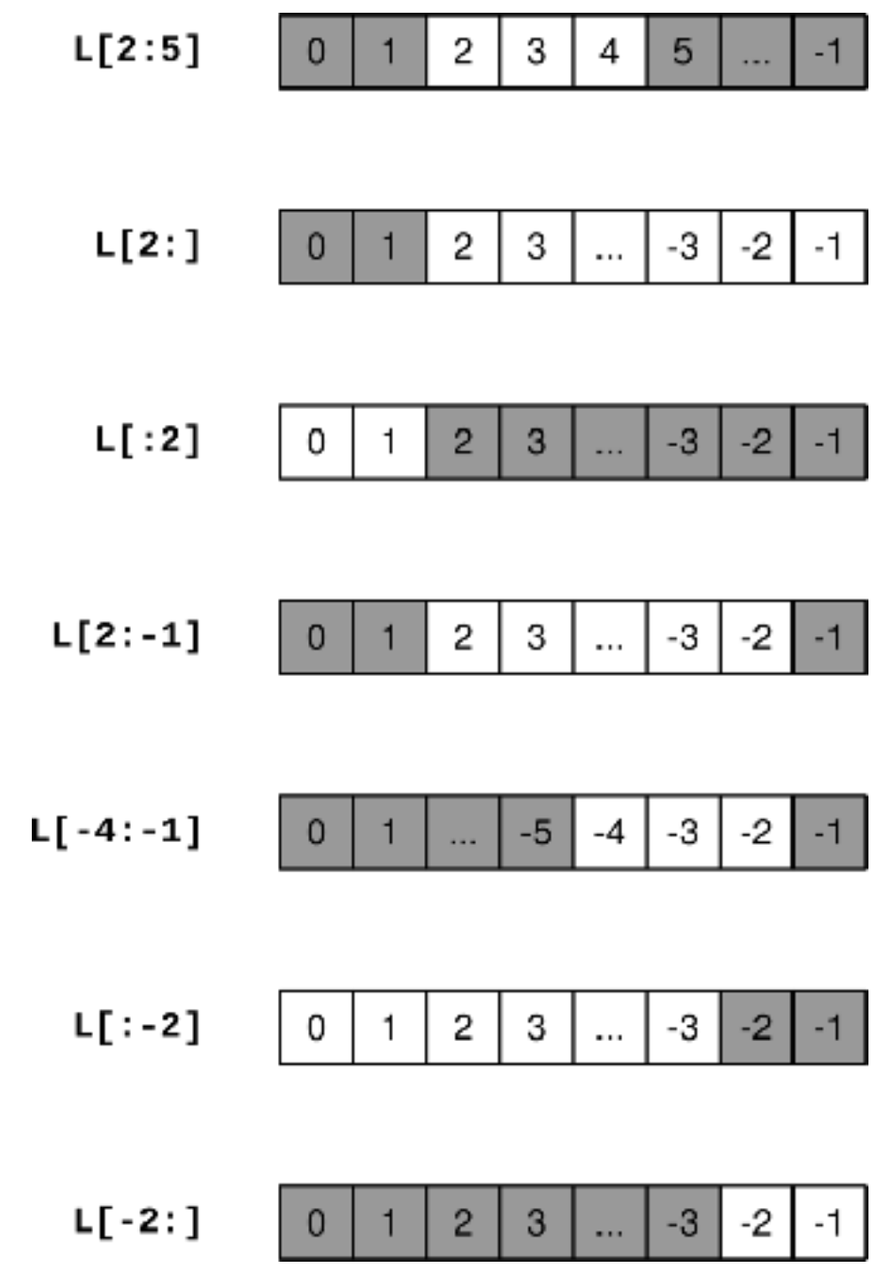
More about lists

- A list of objects of any kind. The individual objects are enumerated by assigning each element an index. The first element in the list gets index 0. (Consider the usual indexing of coefficients of a polynomial)

```
M = [3,['a', -3.0, 5]]
print(M[1])
print(M[1][2])
```

```
['a', -3.0, 5]
5
```

- Slicing a list `L[i:j]` means create a list by taking all elements from `L` starting at `L[i]` until `L[j-1]`.



Quiz!!

```
L = list(range(4))
print(L[1:100])
print(L[-100:-1])
print(L[5:0])
print(L[2:-2])
print(L[4])
```

```
[1, 2, 3]
```

```
a = [1,2,3]
for iteration in range(4):
    print(sum(a[0:iteration-1]))
```

Quiz!!

```
L = list(range(4))
print(L[1:100])
print(L[-100:-1])
print(L[5:0])
print(L[2:-2])
print(L[4])
```

```
[1, 2, 3]
[0, 1, 2]
```

```
a = [1,2,3]
for iteration in range(4):
    print(sum(a[0:iteration-1]))
```

Quiz!!

```
L = list(range(4))
print(L[1:100])
print(L[-100:-1])
print(L[5:0])
print(L[2:-2])
print(L[4])
```

```
[1, 2, 3]
[0, 1, 2]
[]
```

```
a = [1,2,3]
for iteration in range(4):
    print(sum(a[0:iteration-1]))
```

Quiz!!

```
L = list(range(4))
print(L[1:100])
print(L[-100:-1])
print(L[5:0])
print(L[2:-2])
print(L[4])
```

```
[1, 2, 3]
[0, 1, 2]
[]
[]
```

```
a = [1,2,3]
for iteration in range(4):
    print(sum(a[0:iteration-1]))
```

Quiz!!

```
L = list(range(4))
print(L[1:100])
print(L[-100:-1])
print(L[5:0])
print(L[2:-2])
print(L[4])
```

```
[1, 2, 3]
[0, 1, 2]
[]
[]
```

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

```
a = [1,2,3]
for iteration in range(4):
    print(sum(a[0:iteration-1]))
```

Quiz!!

```
L = list(range(4))
print(L[1:100])
print(L[-100:-1])
print(L[5:0])
print(L[2:-2])
print(L[4])
```

```
[1, 2, 3]
[0, 1, 2]
[]
[]
```

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

```
a = [1,2,3]
for iteration in range(4):
    print(sum(a[0:iteration-1]))
```

```
3
0
1
3
```

Strides（跨步）

- One may also specify a stride, which is the length of the step from one index to the other.
- The default stride is one. The stride may also be negative.

```
L = list(range(100))
print(L[:10:2])
print(L[::20])
print(L[10:20:3])
print(L[20:10:-3])
```

- It is also possible to create a new list that is reversed, using a negative stride.

```
L = [1, 2, 3]
R = L[::-1] # L is not modified
print(L)
print(R)
```

```
[1, 2, 3]
[3, 2, 1]
```

Strides（跨步）

- One may also specify a stride, which is the length of the step from one index to the other.
- The default stride is one. The stride may also be negative.

```
L = list(range(100))
print(L[:10:2])
print(L[::20])
print(L[10:20:3])
print(L[20:10:-3])
```

```
[0, 2, 4, 6, 8]
```

- It is also possible to create a new list that is reversed, using a negative stride.

```
L = [1, 2, 3]
R = L[::-1] # L is not modified
print(L)
print(R)
```

```
[1, 2, 3]
[3, 2, 1]
```

Strides（跨步）

- One may also specify a stride, which is the length of the step from one index to the other.
- The default stride is one. The stride may also be negative.

```
L = list(range(100))
print(L[:10:2])
print(L[::20])
print(L[10:20:3])
print(L[20:10:-3])
```

```
[0, 2, 4, 6, 8]
[0, 20, 40, 60, 80]
```

- It is also possible to create a new list that is reversed, using a negative stride.

```
L = [1, 2, 3]
R = L[::-1] # L is not modified
print(L)
print(R)
```

```
[1, 2, 3]
[3, 2, 1]
```

Strides（跨步）

- One may also specify a stride, which is the length of the step from one index to the other.
- The default stride is one. The stride may also be negative.

```
L = list(range(100))
print(L[:10:2])
print(L[::20])
print(L[10:20:3])
print(L[20:10:-3])
```

```
[0, 2, 4, 6, 8]
[0, 20, 40, 60, 80]
[10, 13, 16, 19]
```

- It is also possible to create a new list that is reversed, using a negative stride.

```
L = [1, 2, 3]
R = L[::-1] # L is not modified
print(L)
print(R)
```

```
[1, 2, 3]
[3, 2, 1]
```

Strides（跨步）

- One may also specify a stride, which is the length of the step from one index to the other.
- The default stride is one. The stride may also be negative.

```
L = list(range(100))
print(L[:10:2])
print(L[::20])
print(L[10:20:3])
print(L[20:10:-3])
```

```
[0, 2, 4, 6, 8]
[0, 20, 40, 60, 80]
[10, 13, 16, 19]
[20, 17, 14, 11]
```

- It is also possible to create a new list that is reversed, using a negative stride.

```
L = [1, 2, 3]
R = L[::-1] # L is not modified
print(L)
print(R)
```

```
[1, 2, 3]
[3, 2, 1]
```

Altering lists

- Deletion: replacing a part of a list by an empty list `[ ]`

```
L = ['a', 1, 2, 3, 4]
L[2:3] = []
print(L)
L[3:] = []
print(L)
```

```
['a', 1, 3, 4]
['a', 1, 3]
```

- Insertion: replacing an empty slice with the list to be inserted

```
L[1:1] = [1000, 2000]
print(L)
```

```
['a', 1000, 2000, 1, 3]
```

- Methods of the datatype list

Command	Action
`list.append(x)`	Add `x` to the end of the list.
`list.extend(L)`	Expand the list by the elements of the list `L`.
`list.insert(i,x)`	Insert `x` at position `i`.
`list.remove(x)`	Remove the first item from the list whose value is `x`.
`list.count(x)`	The number of times `x` appears in the list.
`list.sort()`	Sort the items of the list, in place.
`list.reverse()`	Reverse the elements of the list, in place.
`list.pop()`	Remove the last element of the list, in place.

In–place operations

- Some methods modify an object and do not return anything (that is, they return None). It is difficult to know whether a method changes an object, except by looking at the code or the documentation.

L is modified in-place

```
L = [0, 1, 2, 2, 2, 3, 4]
L.append(5)
print(L)
L.reverse()
print(L)
L.sort()
print(L)
L.remove(0)
print(L)
L.pop()
print(L)
L.pop()
print(L)
newL = L.extend(['a','b','c'])
print(L)
print(newL)
L_Count = L.count(2)
print(L_Count)
```

```
[0, 1, 2, 2, 2, 3, 4, 5]
[5, 4, 3, 2, 2, 2, 1, 0]
[0, 1, 2, 2, 2, 3, 4, 5]
[1, 2, 2, 2, 3, 4, 5]


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

In–place operations

- Some methods modify an object and do not return anything (that is, they return None). It is difficult to know whether a method changes an object, except by looking at the code or the documentation.

L is modified in-place

```
L = [0, 1, 2, 2, 2, 3, 4]
L.append(5)
print(L)
L.reverse()
print(L)
L.sort()
print(L)
L.remove(0)
print(L)
L.pop()
print(L)
L.pop()
print(L)
newL = L.extend(['a','b','c'])
print(L)
print(newL)
L_Count = L.count(2)
print(L_Count)
```

```
[0, 1, 2, 2, 2, 3, 4, 5]
[5, 4, 3, 2, 2, 2, 1, 0]
[0, 1, 2, 2, 2, 3, 4, 5]
[1, 2, 2, 2, 3, 4, 5]
[1, 2, 2, 2, 3, 4]

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

In–place operations

- Some methods modify an object and do not return anything (that is, they return None). It is difficult to know whether a method changes an object, except by looking at the code or the documentation.

L is modified in-place

```
L = [0, 1, 2, 2, 2, 3, 4]
L.append(5)
print(L)
L.reverse()
print(L)
L.sort()
print(L)
L.remove(0)
print(L)
L.pop()
print(L)
L.pop()
print(L)
newL = L.extend(['a','b','c'])
print(L)
print(newL)
L_Count = L.count(2)
print(L_Count)
```

```
[0, 1, 2, 2, 2, 3, 4, 5]
[5, 4, 3, 2, 2, 2, 1, 0]
[0, 1, 2, 2, 2, 3, 4, 5]
[1, 2, 2, 2, 3, 4, 5]
[1, 2, 2, 2, 3, 4]
[1, 2, 2, 2, 3]
[1, 2, 2, 2, 3, 'a', 'b', 'c']
```

In–place operations

- Some methods modify an object and do not return anything (that is, they return None). It is difficult to know whether a method changes an object, except by looking at the code or the documentation.

L is modified in-place

```
L = [0, 1, 2, 2, 2, 3, 4]
L.append(5)
print(L)
L.reverse()
print(L)
L.sort()
print(L)
L.remove(0)
print(L)
L.pop()
print(L)
L.pop()
print(L)
newL = L.extend(['a','b','c'])
print(L)
print(newL)
L_Count = L.count(2)
print(L_Count)
```

```
[0, 1, 2, 2, 2, 3, 4, 5]
[5, 4, 3, 2, 2, 2, 1, 0]
[0, 1, 2, 2, 2, 3, 4, 5]
[1, 2, 2, 2, 3, 4, 5]
[1, 2, 2, 2, 3, 4]
[1, 2, 2, 2, 3]
[1, 2, 2, 2, 3, 'a', 'b', 'c']
None
```

In–place operations

- Some methods modify an object and do not return anything (that is, they return None). It is difficult to know whether a method changes an object, except by looking at the code or the documentation.

L is modified in-place

```
L = [0, 1, 2, 2, 2, 3, 4]
L.append(5)
print(L)
L.reverse()
print(L)
L.sort()
print(L)
L.remove(0)
print(L)
L.pop()
print(L)
L.pop()
print(L)
newL = L.extend(['a','b','c'])
print(L)
print(newL)
L_Count = L.count(2)
print(L_Count)
```

```
[0, 1, 2, 2, 2, 3, 4, 5]
[5, 4, 3, 2, 2, 2, 1, 0]
[0, 1, 2, 2, 2, 3, 4, 5]
[1, 2, 2, 2, 3, 4, 5]
[1, 2, 2, 2, 3, 4]
[1, 2, 2, 2, 3]
[1, 2, 2, 2, 3, 'a', 'b', 'c']
None
3
```

In–place operations

- Some methods modify an object and do not return anything (that is, they return None). It is difficult to know whether a method changes an object, except by looking at the code or the documentation.

L is modified in-place

```
L = [0, 1, 2, 2, 2, 3, 4]
L.append(5)
print(L)
L.reverse()
print(L)
L.sort()
print(L)
L.remove(0)
print(L)
L.pop()
print(L)
L.pop()
print(L)
newL = L.extend(['a','b','c'])
print(L)
print(newL)
L_Count = L.count(2)
print(L_Count)
```

```
[0, 1, 2, 2, 2, 3, 4, 5]
[5, 4, 3, 2, 2, 2, 1, 0]
[0, 1, 2, 2, 2, 3, 4, 5]
[1, 2, 2, 2, 3, 4, 5]
[1, 2, 2, 2, 3, 4]
[1, 2, 2, 2, 3]
[1, 2, 2, 2, 3, 'a', 'b', 'c']
None
3
```

Return `None`

In–place operations

- Some methods modify an object and do not return anything (that is, they return None). It is difficult to know whether a method changes an object, except by looking at the code or the documentation.

L is modified in-place

```
L = [0, 1, 2, 2, 2, 3, 4]
L.append(5)
print(L)
L.reverse()
print(L)
L.sort()
print(L)
L.remove(0)
print(L)
L.pop()
print(L)
L.pop()
print(L)
newL = L.extend(['a','b','c'])
print(L)
print(newL)
L_Count = L.count(2)
print(L_Count)
```

```
[0, 1, 2, 2, 2, 3, 4, 5]
[5, 4, 3, 2, 2, 2, 1, 0]
[0, 1, 2, 2, 2, 3, 4, 5]
[1, 2, 2, 2, 3, 4, 5]
[1, 2, 2, 2, 3, 4]
[1, 2, 2, 2, 3]
[1, 2, 2, 2, 3, 'a', 'b', 'c']
None
3
```

Return `None`

Return `3`

List comprehension

- List comprehension is closely related to the mathematical notation for sets. Compare: $L_2 = \{2x;\ x \in L\}$ and `L2 = [2*x for x in L]`.

- The syntax of a list comprehension is:

```
[<expr> for <variable> in <list> if <condition>]
```

- Here is an example:

```
L = [2, 3, 10, 1, 5]
L2 = [x*2 for x in L]
L3 = [x*2 for x in L if 4 < x <= 10]
print(L2)
print(L3)
```

```
[4, 6, 20, 2, 10]
[20, 10]
```

- It is possible to have several for loops inside a list comprehension:

```
M = [[1,2,3,4,5,6],[7,8,9],[10,11,12]]
L = [M[i][j] for i in range(2) for j in range(3)]
print(L)
```

List comprehension

- List comprehension is closely related to the mathematical notation for sets. Compare: $L_2 = \{2x;\ x \in L\}$ and `L2 = [2*x for x in L]`.

- The syntax of a list comprehension is:

```
[<expr> for <variable> in <list> if <condition>]
```

- Here is an example:

```
L = [2, 3, 10, 1, 5]
L2 = [x*2 for x in L]
L3 = [x*2 for x in L if 4 < x <= 10]
print(L2)
print(L3)
```

```
[4, 6, 20, 2, 10]
[20, 10]
```

- It is possible to have several for loops inside a list comprehension:

```
[1, 2, 3, 7, 8, 9]
```

```
M = [[1,2,3,4,5,6],[7,8,9],[10,11,12]]
L = [M[i][j] for i in range(2) for j in range(3)]
print(L)
```

Tuples

- A tuple is an immutable list. Immutable means that it cannot be modified. A tuple is just a comma-separated sequence of objects (a list without brackets). To increase readability, one often encloses a tuple in a pair of parentheses

```
my_tuple = 1, 2, 3# our first tuple
my_tuple = (1, 2, 3) # the same
my_tuple = 1, 2, 3,  # again the same
len(my_tuple) # 3, same as for lists
my_tuple[0] = 'a' # error! tuples are immutable
```

- The comma indicates that the object is a tuple

```
singleton = 1,  # note the comma
print(len(singleton))
x = 1
print(len(x))
```

```
1
```

```
TypeError: object of type 'int' has no len()
```

Tuples

- Tuples are useful when a group of values goes together. One may assign several variables at once by unpacking a list or tuple. For example, they are used to return multiple values from functions.

```
a, b = 0, 1 # a gets 0 and b gets 1
print(a)
print(b)
a, b = [0, 1] # exactly the same effect
(a, b) = 0, 1 # same
[a,b] = [0,1] # same thing
```

```
0
1
```

- The swapping trick:

```
a, b = b, a
print(a)
print(b)
```

```
1
0
```

- The notation without parentheses is convenient but dangerous. You should use parentheses when you are not sure:

```
print(1, 2 == 3, 4)
print((1, 2) == (3, 4))
```

Tuples

- Tuples are useful when a group of values goes together. One may assign several variables at once by unpacking a list or tuple. For example, they are used to return multiple values from functions.

```
a, b = 0, 1 # a gets 0 and b gets 1
print(a)
print(b)
a, b = [0, 1] # exactly the same effect
(a, b) = 0, 1 # same
[a,b] = [0,1] # same thing
```

```
0
1
```

- The swapping trick:

```
a, b = b, a
print(a)
print(b)
```

```
1
0
```

- The notation without parentheses is convenient but dangerous. You should use parentheses when you are not sure:

```
print(1, 2 == 3, 4)
print((1, 2) == (3, 4))
```

```
1 False 4
```

Tuples

- Tuples are useful when a group of values goes together. One may assign several variables at once by unpacking a list or tuple. For example, they are used to return multiple values from functions.

```
a, b = 0, 1 # a gets 0 and b gets 1
print(a)
print(b)
a, b = [0, 1] # exactly the same effect
(a, b) = 0, 1 # same
[a,b] = [0,1] # same thing
```

```
0
1
```

- The swapping trick:

```
a, b = b, a
print(a)
print(b)
```

```
1
0
```

- The notation without parentheses is convenient but dangerous. You should use parentheses when you are not sure:

```
print(1, 2 == 3, 4)
print((1, 2) == (3, 4))
```

```
1 False 4
False
```

Dictionaries

- Lists, tuples, and arrays are ordered sets of objects. The individual objects are inserted, accessed, and processed according to their place in the list. Dictionaries are unordered sets of pairs. One accesses dictionary data by keys.

```
truck_wheel = {'name':'wheel','mass':5.7,
'Ix':20.0,'Iy':1.,'Iz':17.,
'center of mass':[0.,0.,0.]}

print(truck_wheel['name'])
print(truck_wheel['mass'])
```

```
wheel
5.7
```

- New objects are added to the dictionary by creating a new key

```
truck_wheel['Ixy'] = 0.0
```

- The command dict generates a dictionary from a list with key/value tuples:

```
truck_wheel = dict([('name','wheel'),('mass',5.7),
                    ('Ix',20.0), ('Iy',1.), ('Iz',17.),
                    ('center of mass',[0.,0.,0.])])
```

Looping over dictionaries

- There are mainly three ways to loop over dictionaries:
  - By keys:

```
for key in truck_wheel.keys():
    print(key)
```

```
name
mass
Ix
Iy
Iz
center of mass
```

  - By value:

```
for value in truck_wheel.values():
    print(value)
```

```
wheel
5.7
20.0
1.0
17.0
[0.0, 0.0, 0.0]
```

  - By item, that is, key/value pairs:

```
for item in truck_wheel.items():
    print(item)
```

```
('name', 'wheel')
('mass', 5.7)
('Ix', 20.0)
('Iy', 1.0)
('Iz', 17.0)
('center of mass', [0.0, 0.0, 0.0])
```

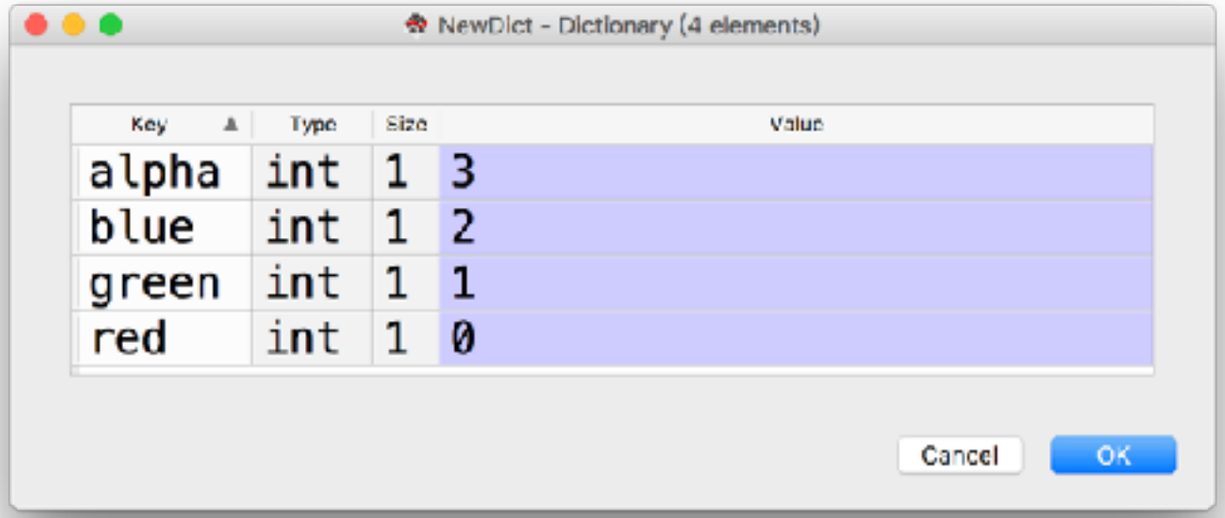
Printed in any order, as order is meaningless in dictionaries.

Merging lists – `zip`

- A particularly useful function for lists is `zip`. It can be used to merge two given lists into a new list by pairing the elements of the original lists. The result is a list of tuples.

- The following example demonstrates what happens if the lists have different lengths. The length of the zipped list is the shorter of the two input lists. The zip function may come in handy for creating a dictionary.

```
ind = [0,1,2,3,4]
color = ["red", "green", "blue", "alpha"]
NewDict = dict(list(zip(color,ind)))
```



Sets

- Sets are containers that share properties and operations with sets in mathematics. A mathematical set is a collection of distinct objects:

$$A = \{1, 2, 3, 4\},\ B = \{5\},\ C = A \cup B,\ D = A \cap C,\ E = C \backslash A,\ 5 \in C$$

```
A = {1,2,3,4}
B = {5}
C = A.union(B)
D = A.intersection(C)
E = C.difference(A)
print('{}\n{}\n{}\n{}\n{}'.format(A,B,C,D,E))
print(5 in C)
```

```
{1, 2, 3, 4}
{5}
{1, 2, 3, 4, 5}
{1, 2, 3, 4}
{5}
True
```

- Sets contain an element only once, corresponding to the aforementioned definition. A set is unordered; that is, the order of the elements in the set is not defined

```
A = {1,2,3,3,3}
B = {1,2,3}
print(A == B)
```

```
True
```

```
A = {1,2,3}
B = {1,3,2}
print(A == B)
```

```
True
```

Sets

- Sets in Python can contain numeric objects, strings, and Booleans.

- There are `union` and `intersection` methods:

```
A={1,2,3,4}
B = A.union({5})
print(A)
C = A.intersection({2,4,6})
print(A)
print(B)
print(C)
```

```
{1, 2, 3, 4, 5}
{2, 4}
```

- sets can be compared using the methods `issubset` and `issuperset`:

```
print({2,4}.issubset({1,2,3,4,5}))
print({1,2,3,4,5}.issuperset({2,4}))
```

```
True
True
```

- Empty set: `empty_set=set([])`

- Empty dictionary: `empty_Dict={}`

Name	Type	Size	Value
empty_Dict	dict	0	{}

Sets

- Sets in Python can contain numeric objects, strings, and Booleans.

- There are `union` and `intersection` methods:

```
A={1,2,3,4}
B = A.union({5})
print(A)
C = A.intersection({2,4,6})
print(A)
print(B)
print(C)
```

```
{1, 2, 3, 4}

{1, 2, 3, 4, 5}
{2, 4}
```

- sets can be compared using the methods `issubset` and `issuperset`:

```
print({2,4}.issubset({1,2,3,4,5}))
print({1,2,3,4,5}.issuperset({2,4}))
```

```
True
True
```

- Empty set: `empty_set=set([])`

- Empty dictionary: `empty_Dict={}`

Name	Type	Size	Value
empty_Dict	dict	0	{}

Sets

- Sets in Python can contain numeric objects, strings, and Booleans.

- There are `union` and `intersection` methods:

```
A={1,2,3,4}
B = A.union({5})
print(A)
C = A.intersection({2,4,6})
print(A)
print(B)
print(C)
```

```
{1, 2, 3, 4}
{1, 2, 3, 4}
{1, 2, 3, 4, 5}
{2, 4}
```

- sets can be compared using the methods `issubset` and `issuperset`:

```
print({2,4}.issubset({1,2,3,4,5}))
print({1,2,3,4,5}.issuperset({2,4}))
```

```
True
True
```

- Empty set: `empty_set=set([])`

- Empty dictionary: `empty_Dict={}`

Name	Type	Size	Value
empty_Dict	dict	0	{}

Container conversions

Type	Access	Order	Duplicate values	Mutability
List	index	yes	yes	yes
Tuple	index	yes	yes	no
Dictionary	key	no	yes	yes
Set	no	no	no	yes

- Due to the different properties of the various container types, we frequently convert one type to another:
-

Container Types	Syntax
List → Tuple	`tuple([1,2,3])`
Tuple → List	`list((1,2,3))`
List, Tuple → Set	`set([1,2,3]), set((1,2,3))`
Set → List	`list({1,2,3})`
Dictionary → List	`{'a':4}.values()`
List → Dictionary	–