Python Built-in Data Structures, Functions, and Files

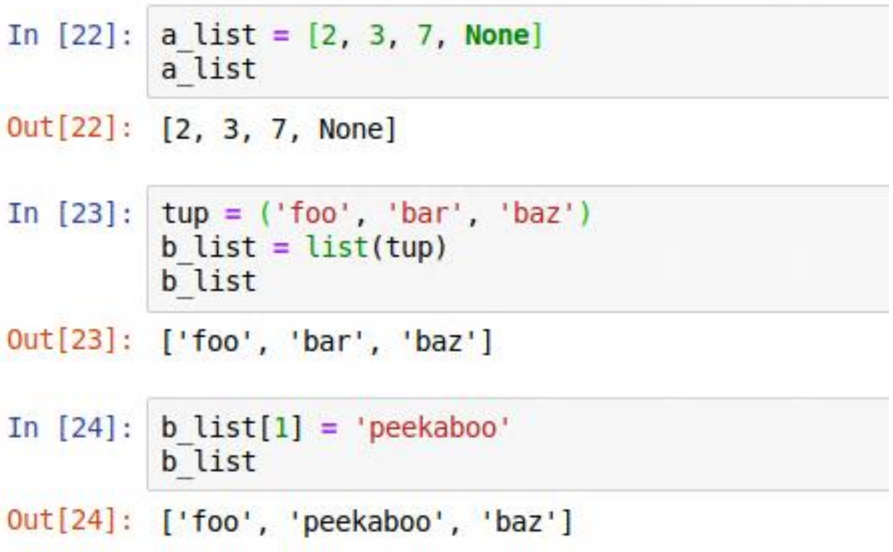
Part 2

Data Structures and Sequences

Part 2

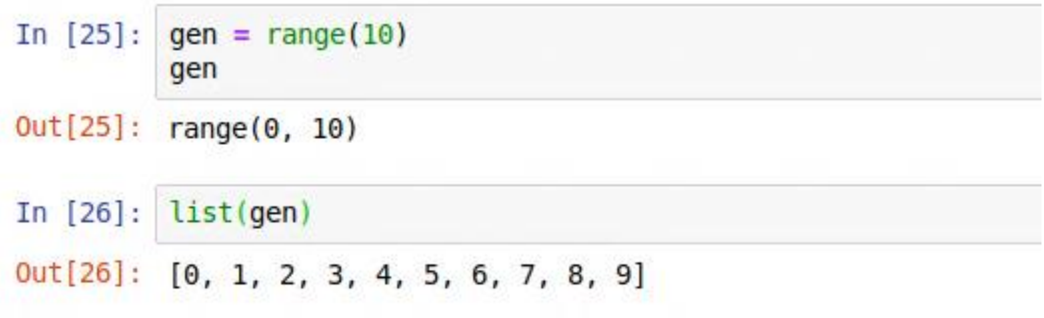
List

- In contrast with tuples, lists are variable-length and their contents can be modified in-place.
- You can define them using square brackets `[]` or using the `list` type function:



```
In [22]: a_list = [2, 3, 7, None]
         a_list

Out[22]: [2, 3, 7, None]

In [23]: tup = ('foo', 'bar', 'baz')
         b_list = list(tup)
         b_list

Out[23]: ['foo', 'bar', 'baz']

In [24]: b_list[1] = 'peekaboo'
         b_list

Out[24]: ['foo', 'peekaboo', 'baz']
```

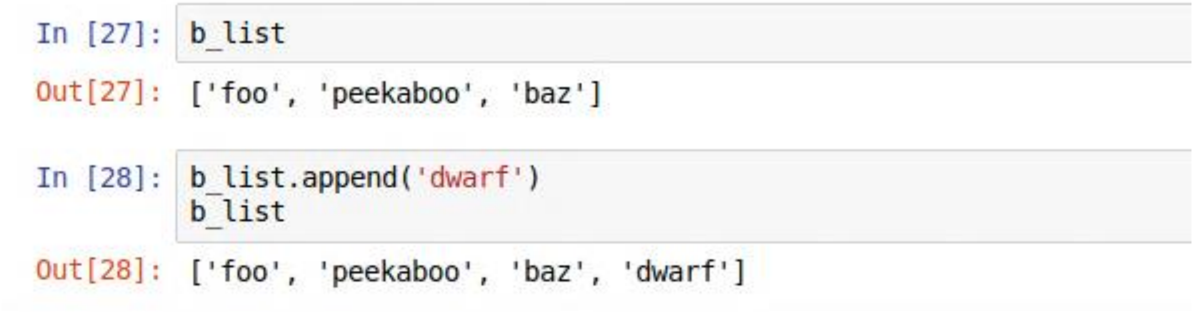
- The `list` function is frequently used in data processing as a way to materialize an iterator or generator expression:



```
In [25]: gen = range(10)
         gen

Out[25]: range(0, 10)

In [26]: list(gen)

Out[26]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

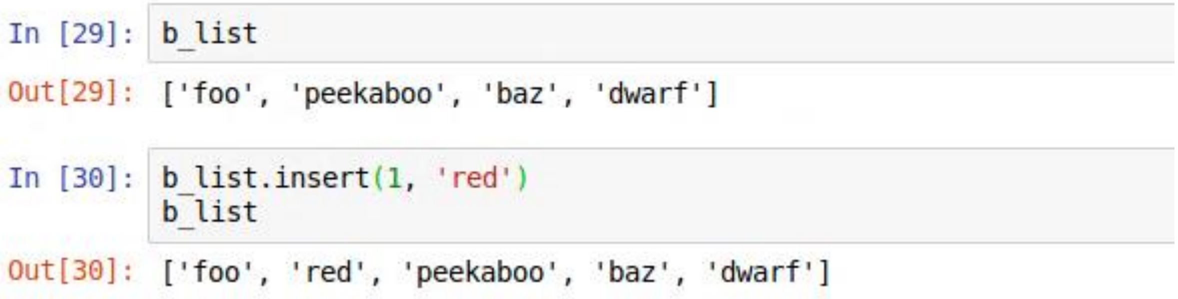
Adding and removing elements

- Elements can be appended to the end of the list with the `append` method:



```
In [27]: b_list

Out[27]: ['foo', 'peekaboo', 'baz']

In [28]: b_list.append('dwarf')
         b_list

Out[28]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

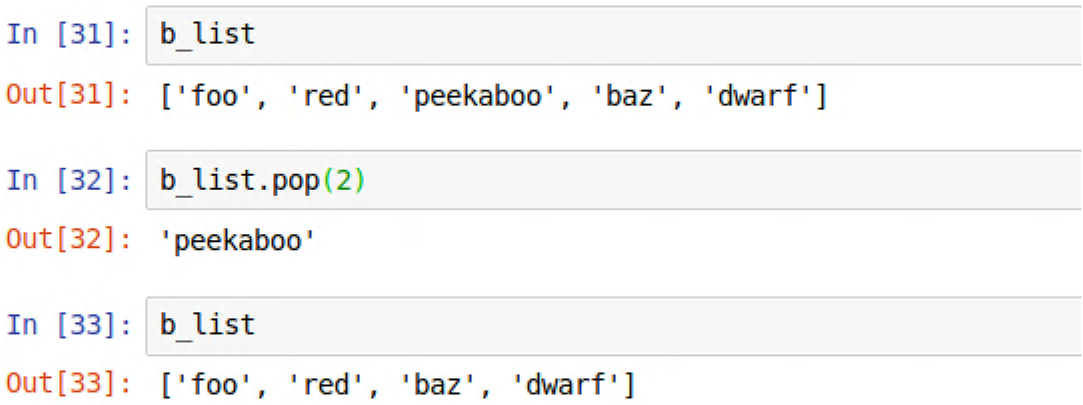
- Using `insert` you can insert an element at a specific location in the list:



```
In [29]: b_list

Out[29]: ['foo', 'peekaboo', 'baz', 'dwarf']

In [30]: b_list.insert(1, 'red')
         b_list

Out[30]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

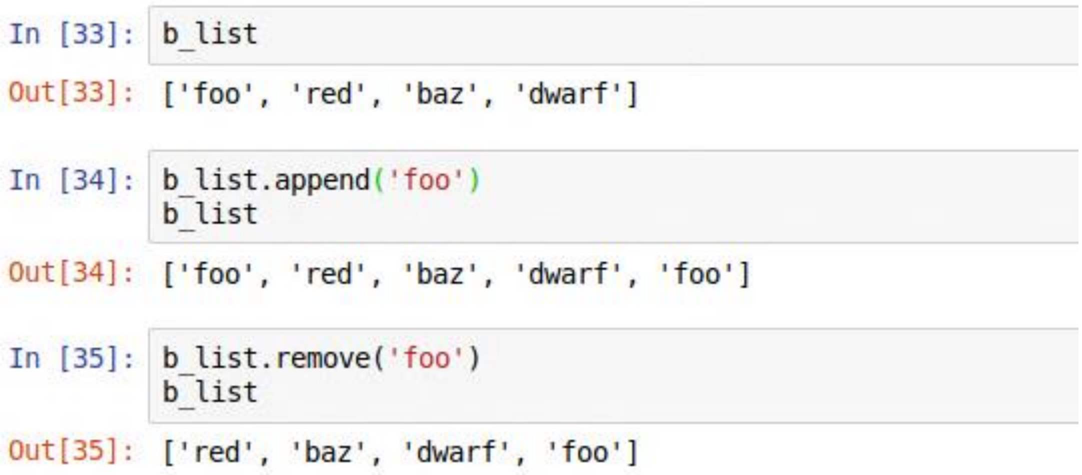
- `insert` is computationally expensive compared with `append`, because references to subsequent elements have to be shifted internally to make room for the new element.
- If you need to insert elements at both the beginning and end of a sequence, you may wish to explore `collections.deque`, a double-ended queue, for this purpose.

- The inverse operation to `insert` is `pop`, which removes and returns an element at a particular index:



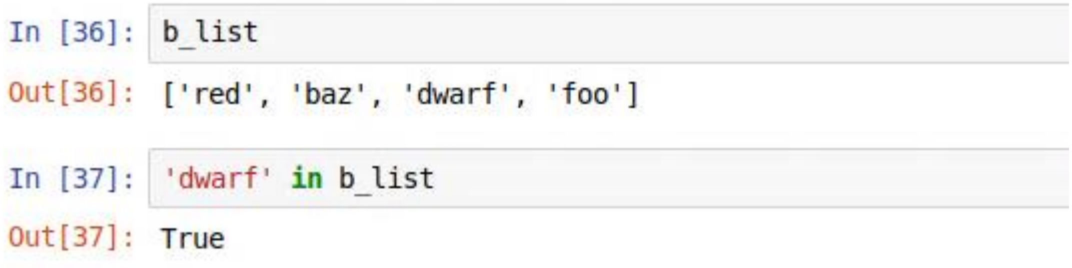
```
In [31]: b_list
Out[31]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']

In [32]: b_list.pop(2)
Out[32]: 'peekaboo'

In [33]: b_list
Out[33]: ['foo', 'red', 'baz', 'dwarf']
```

- Elements can be removed by value with `remove`, which locates the first such value and removes it from the list:



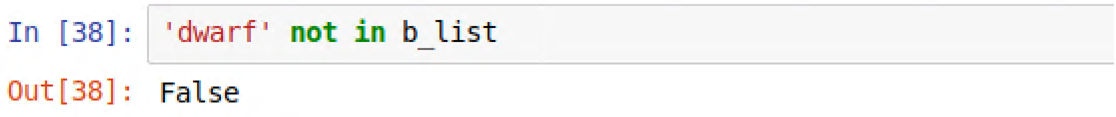
```
In [33]: b_list

Out[33]: ['foo', 'red', 'baz', 'dwarf']

In [34]: b_list.append('foo')
         b_list

Out[34]: ['foo', 'red', 'baz', 'dwarf', 'foo']

In [35]: b_list.remove('foo')
         b_list

Out[35]: ['red', 'baz', 'dwarf', 'foo']
```

- Check if a list contains a value using the `in` keyword:



```
In [36]: b_list
Out[36]: ['red', 'baz', 'dwarf', 'foo']

In [37]: 'dwarf' in b_list
Out[37]: True
```

- The keyword `not` can be used to negate `in`:

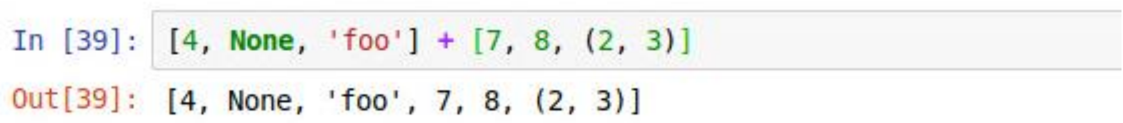


```
In [38]: 'dwarf' not in b_list
Out[38]: False
```

- Checking whether a list contains a value is a lot slower than doing so with dicts and sets, as Python makes a linear scan across the values of the list, whereas it can check the others (based on hash tables) in constant time.

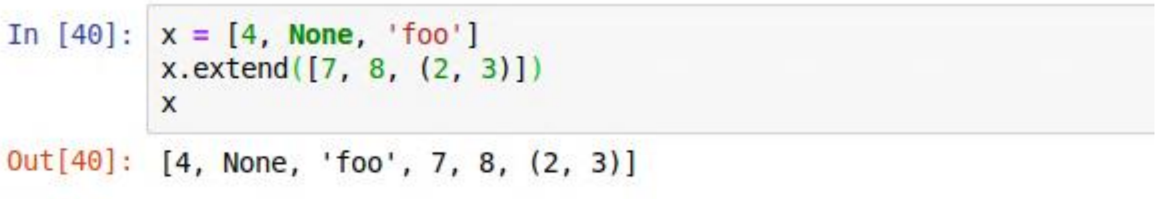
Concatenating and combining lists

- Similar to tuples, adding two lists together with + concatenates them:



```
In [39]: [4, None, 'foo'] + [7, 8, (2, 3)]

Out[39]: [4, None, 'foo', 7, 8, (2, 3)]
```

- If you have a list already defined, you can append multiple elements to it using the `extend` method:



```
In [40]: x = [4, None, 'foo']
         x.extend([7, 8, (2, 3)])
         x

Out[40]: [4, None, 'foo', 7, 8, (2, 3)]
```

- Note that list concatenation by addition is a comparatively expensive operation since a new list must be created and the objects copied over.
- Using `extend` to append elements to an existing list, especially if you are building up a large list, is usually preferable.
- Thus,

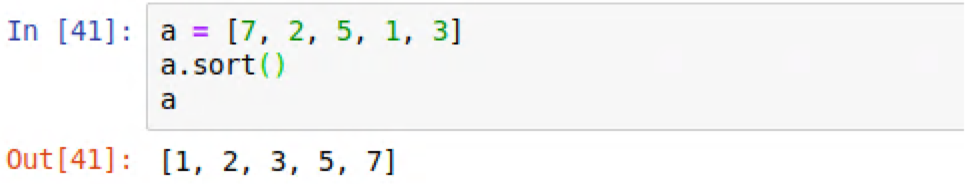
  ```
  everything = []
  for chunk in list_of_lists:
      everything.extend(chunk)
  ```

  is faster than the concatenative alternative:

  ```
  everything = []
  for chunk in list_of_lists:
      everything = everything + chunk
  ```

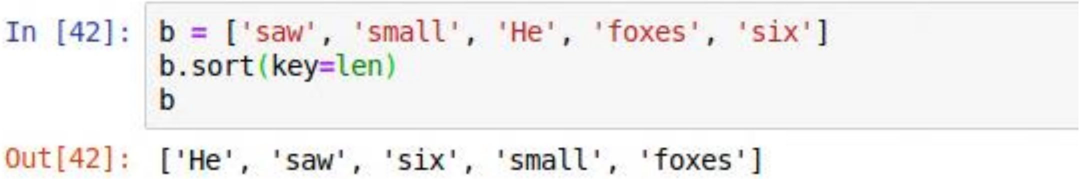
Sorting

- You can sort a list in-place (without creating a new object) by calling its `sort` function:



```
In [41]: a = [7, 2, 5, 1, 3]
         a.sort()
         a

Out[41]: [1, 2, 3, 5, 7]
```

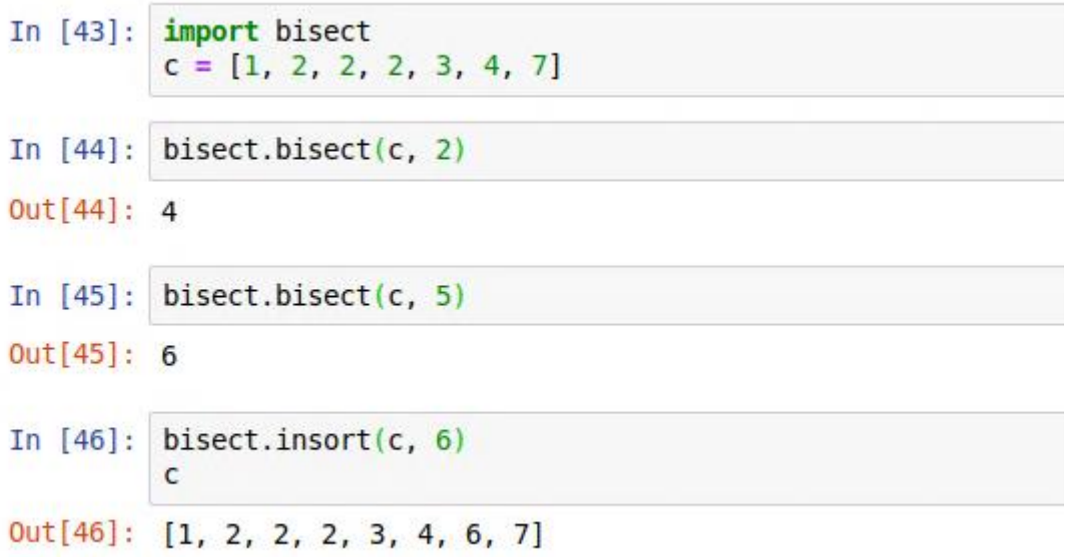
- `sort` has a few options that will occasionally come in handy.
- One is the ability to pass a secondary *sort key*—that is, a function that produces a value to use to sort the objects.
- For example, we could sort a collection of strings by their lengths:



```
In [42]: b = ['saw', 'small', 'He', 'foxes', 'six']
         b.sort(key=len)
         b

Out[42]: ['He', 'saw', 'six', 'small', 'foxes']
```

Binary search and maintaining a sorted list

- The built-in `bisect` module implements binary search and insertion into a sorted list.
- `bisect.bisect` finds the location where an element should be inserted to keep it sorted, while `bisect.insort` actually inserts the element into that location:

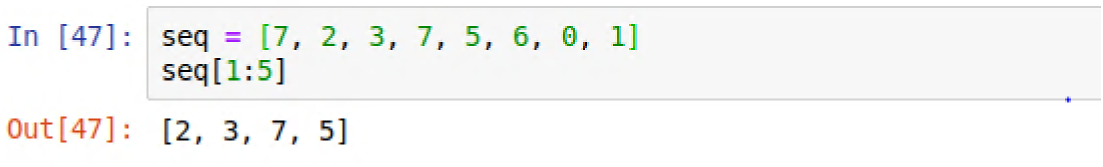


```
In [43]: import bisect
         c = [1, 2, 2, 2, 3, 4, 7]

In [44]: bisect.bisect(c, 2)

Out[44]: 4

In [45]: bisect.bisect(c, 5)

Out[45]: 6

In [46]: bisect.insort(c, 6)
         c

Out[46]: [1, 2, 2, 2, 3, 4, 6, 7]
```

- The `bisect` module functions do not check whether the list is sorted, as doing so would be computationally expensive.
- Thus, using them with an unsorted list will succeed without error but may lead to incorrect results.

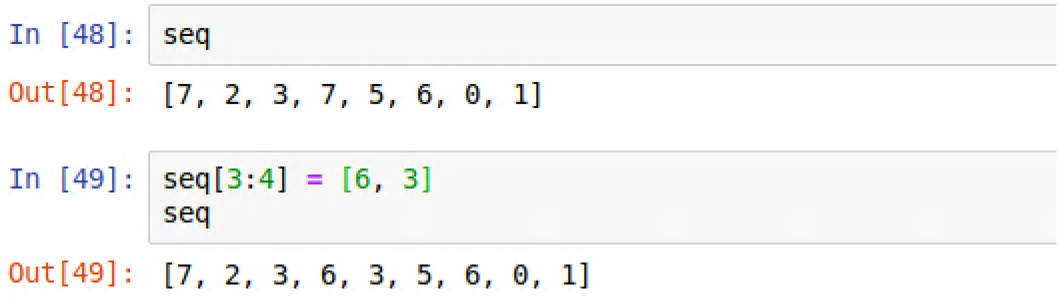
Slicing

- You can select sections of most sequence types by using slice notation, which in its basic form consists of `start:stop` passed to the indexing operator `[]`:



```
In [47]: seq = [7, 2, 3, 7, 5, 6, 0, 1]
         seq[1:5]

Out[47]: [2, 3, 7, 5]
```

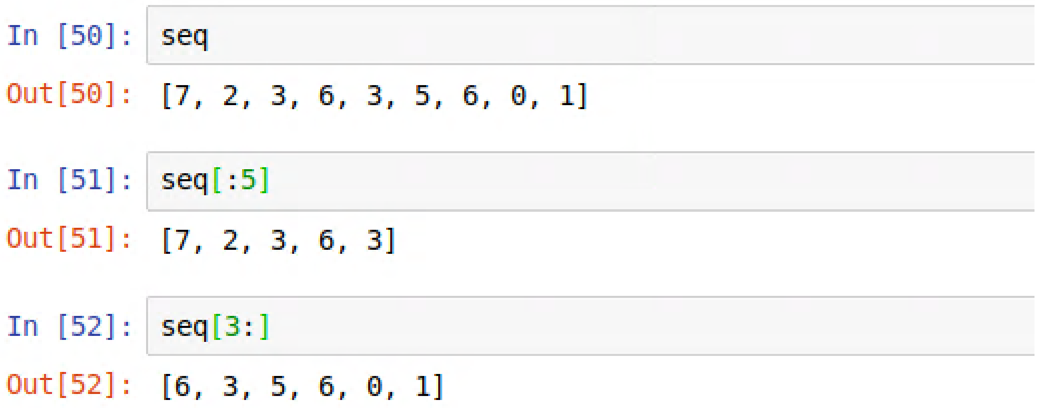
- Slices can also be assigned to with a sequence:



```
In [48]:  seq

Out[48]:  [7, 2, 3, 7, 5, 6, 0, 1]

In [49]:  seq[3:4] = [6, 3]
          seq

Out[49]:  [7, 2, 3, 6, 3, 5, 6, 0, 1]
```

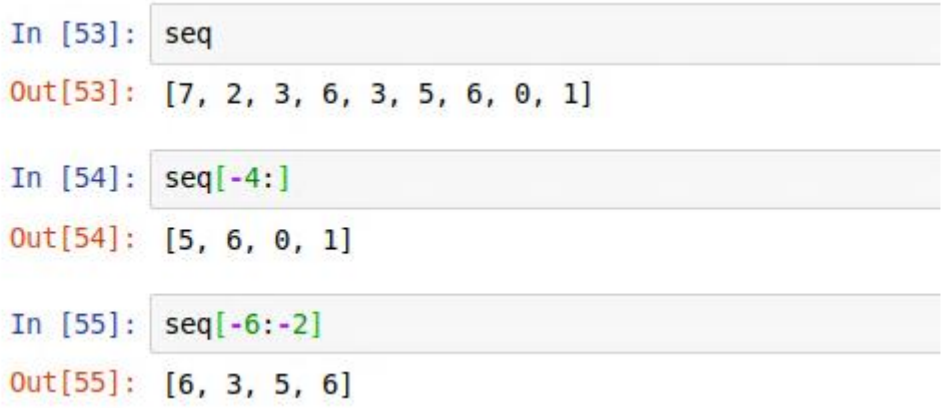
- While the element at the `start` index is included, the `stop` index is *not included*, so that the number of elements in the result is `stop – start`.

- Either the `start` or `stop` can be omitted, in which case they default to the start of the sequence and the end of the sequence, respectively:



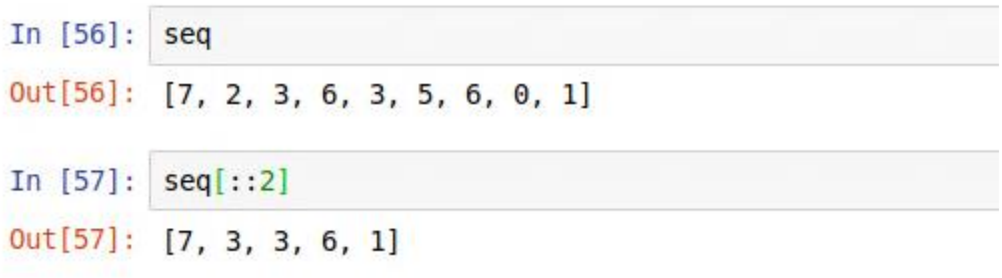
```
In [50]: seq
Out[50]: [7, 2, 3, 6, 3, 5, 6, 0, 1]

In [51]: seq[:5]
Out[51]: [7, 2, 3, 6, 3]

In [52]: seq[3:]
Out[52]: [6, 3, 5, 6, 0, 1]
```

- Negative indices slice the sequence relative to the end:



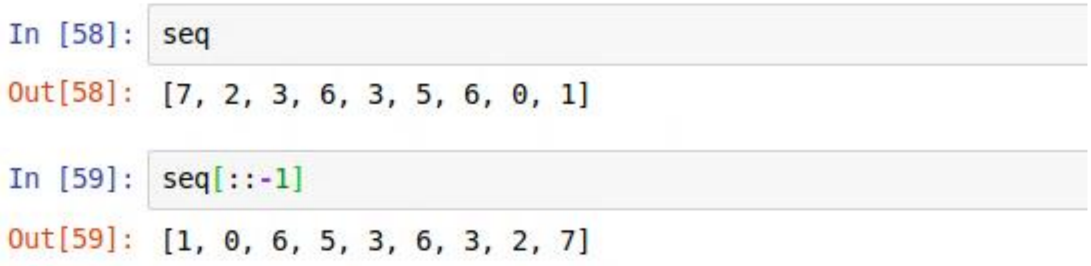
```
In [53]: seq
Out[53]: [7, 2, 3, 6, 3, 5, 6, 0, 1]

In [54]: seq[-4:]
Out[54]: [5, 6, 0, 1]

In [55]: seq[-6:-2]
Out[55]: [6, 3, 5, 6]
```

- A step can also be used after a second colon to, say, take every other element:



```
In [56]: seq
Out[56]: [7, 2, 3, 6, 3, 5, 6, 0, 1]

In [57]: seq[::2]
Out[57]: [7, 3, 3, 6, 1]
```

- A clever use of this is to pass `−1`, which has the useful effect of reversing a list or tuple:



```
In [58]: seq
Out[58]: [7, 2, 3, 6, 3, 5, 6, 0, 1]

In [59]: seq[::-1]
Out[59]: [1, 0, 6, 5, 3, 6, 3, 2, 7]
```