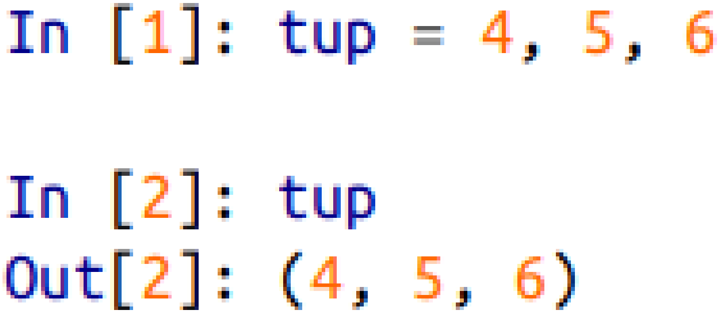
Lecture 2: Built-in Data Structures, Functions, and Files

Alpar Sultan, PhD

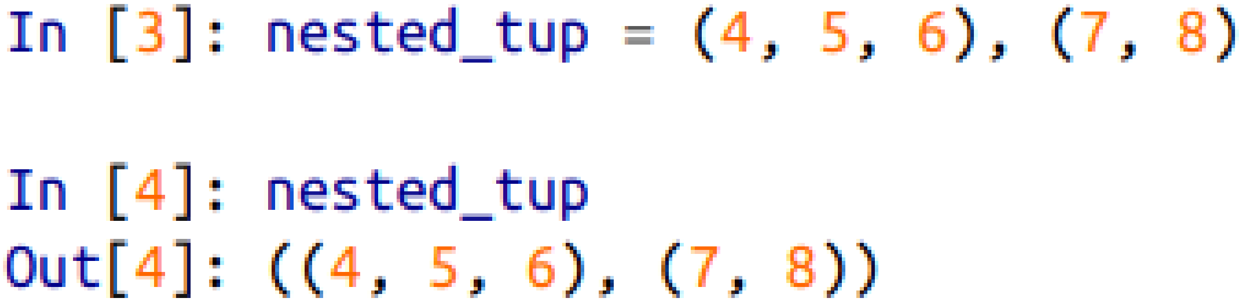
Associate professor

Tuple

- A tuple is a fixed-length, immutable sequence of Python objects. The easiest way to create one is with a comma-separated sequence of values:



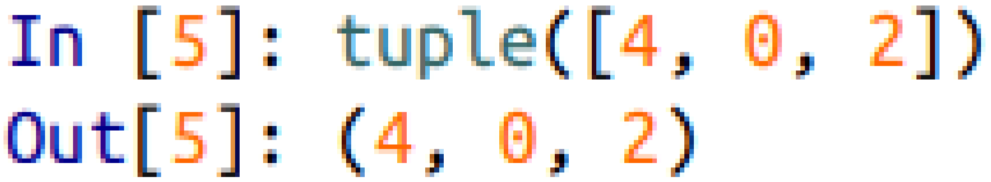
```
In [1]: tup = 4, 5, 6

In [2]: tup
Out[2]: (4, 5, 6)
```



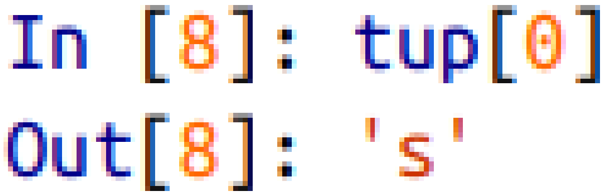
```
In [3]: nested_tup = (4, 5, 6), (7, 8)

In [4]: nested_tup
Out[4]: ((4, 5, 6), (7, 8))
```

Tuple

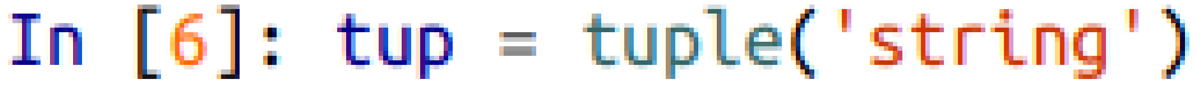
- You can convert any sequence or iterator to a tuple by invoking tuple:



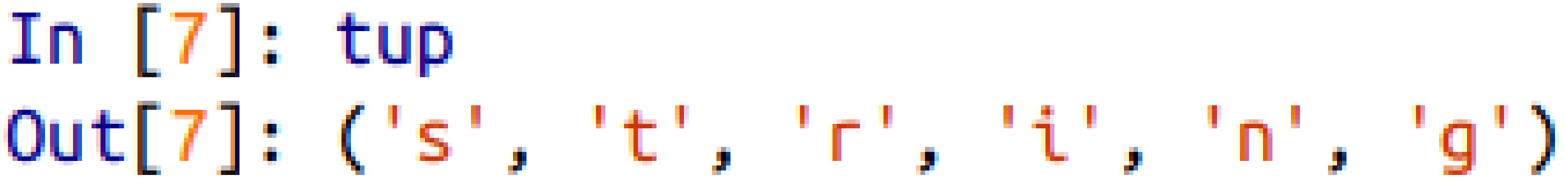
```
In [5]: tuple([4, 0, 2])
Out[5]: (4, 0, 2)
```



```
In [8]: tup[0]
Out[8]: 's'
```



```
In [6]: tup = tuple('string')
```

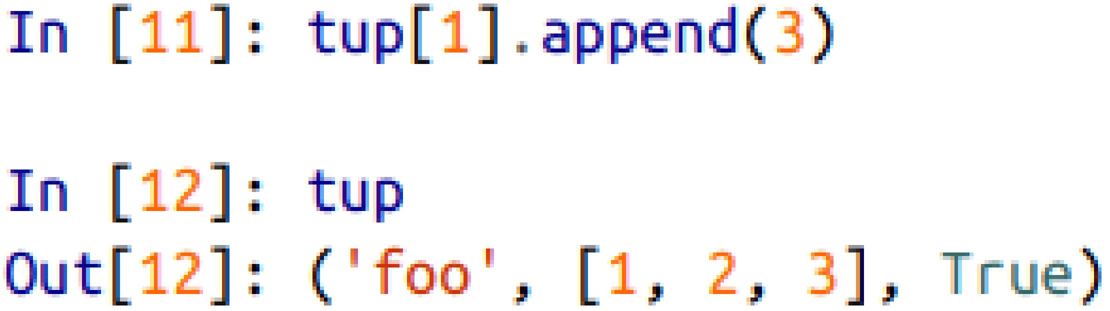


```
In [7]: tup
Out[7]: ('s', 't', 'r', 'i', 'n', 'g')
```

Tuple

- once the tuple is created it’s not possible to modify which object is stored in each slot

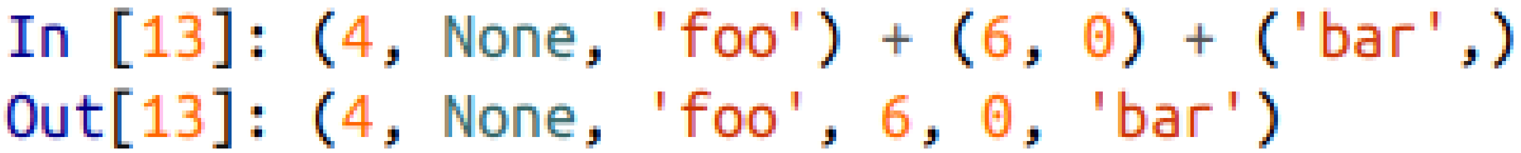
- If an object inside a tuple is mutable, such as a list, you can modify it in-place:



```
In [11]: tup[1].append(3)

In [12]: tup
Out[12]: ('foo', [1, 2, 3], True)
```

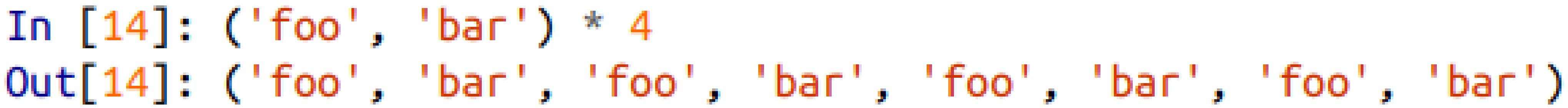
Tuple

- You can concatenate tuples using the + operator to produce longer tuples:



```
In [13]: (4, None, 'foo') + (6, 0) + ('bar',)
Out[13]: (4, None, 'foo', 6, 0, 'bar')
```

- Multiplying a tuple by an integer, as with lists, has the effect of concatenating together that many copies of the tuple:

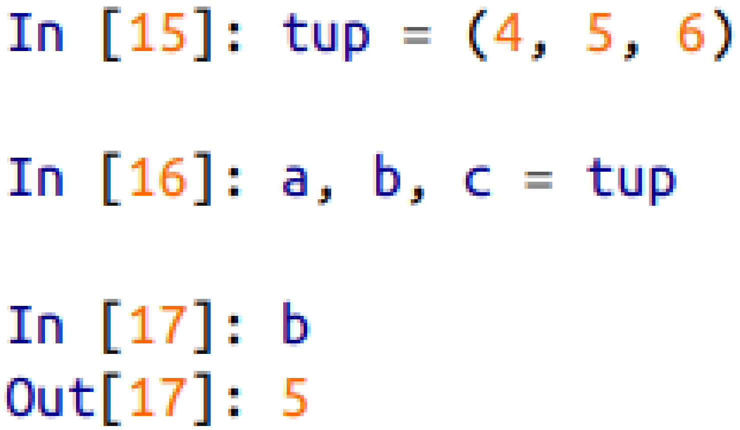


```
In [14]: ('foo', 'bar') * 4
Out[14]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

- Note that the objects themselves are not copied, only the references to them.

Unpacking tuples

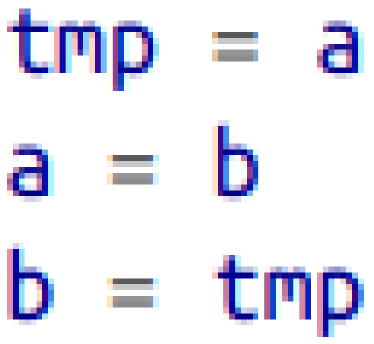
- If you try to assign to a tuple-like expression of variables, Python will attempt to unpack the value on the righthand side of the equals sign:



```
In [15]: tup = (4, 5, 6)

In [16]: a, b, c = tup

In [17]: b
Out[17]: 5
```

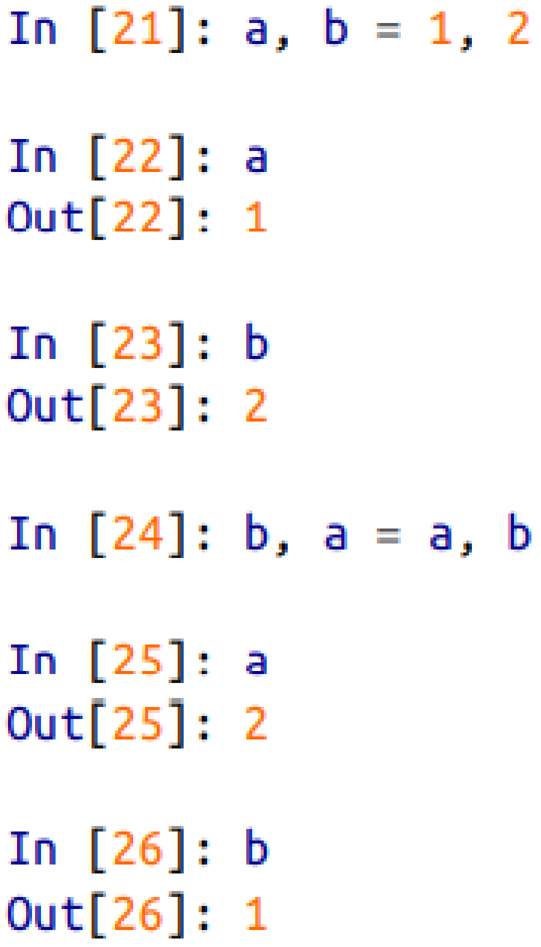
Swap

- Using this functionality you can easily swap variable names, a task which in many languages might look like:



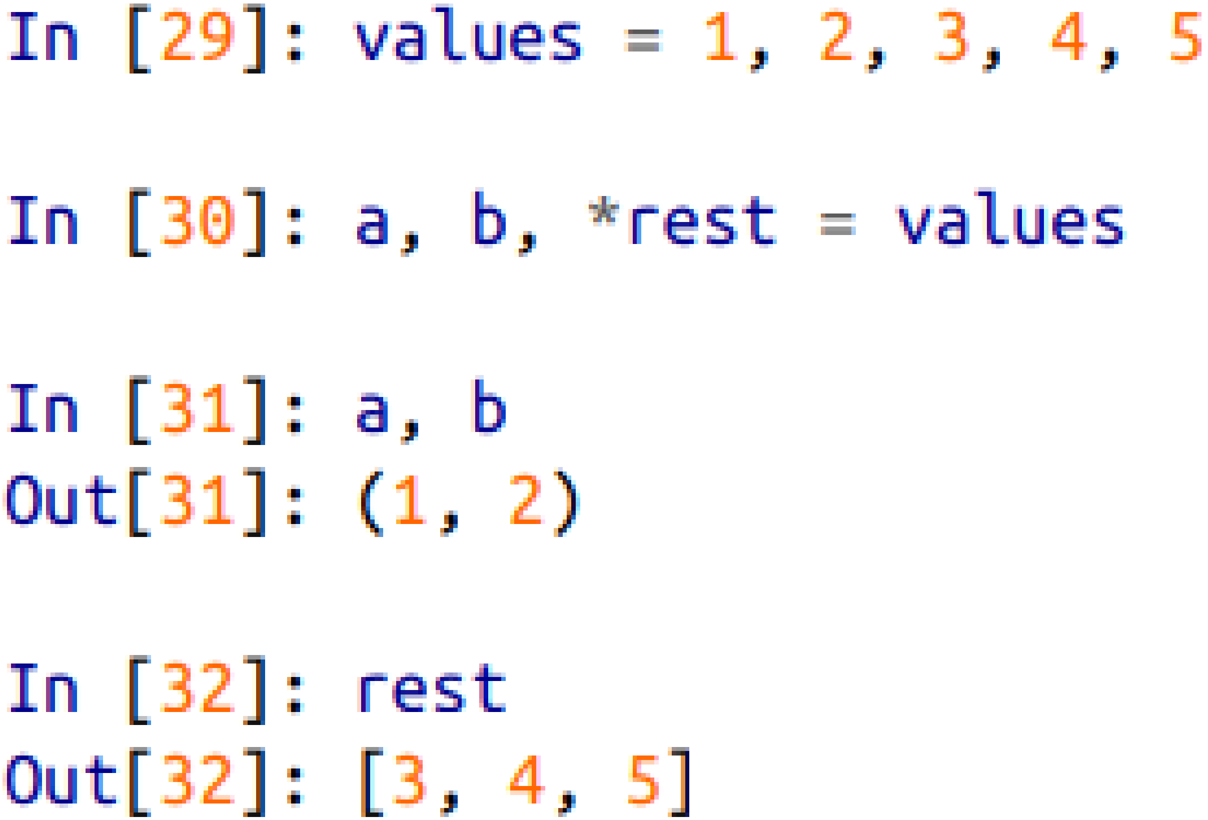
```
tmp = a
a = b
b = tmp
```

But, in Python, the swap can be done like this:



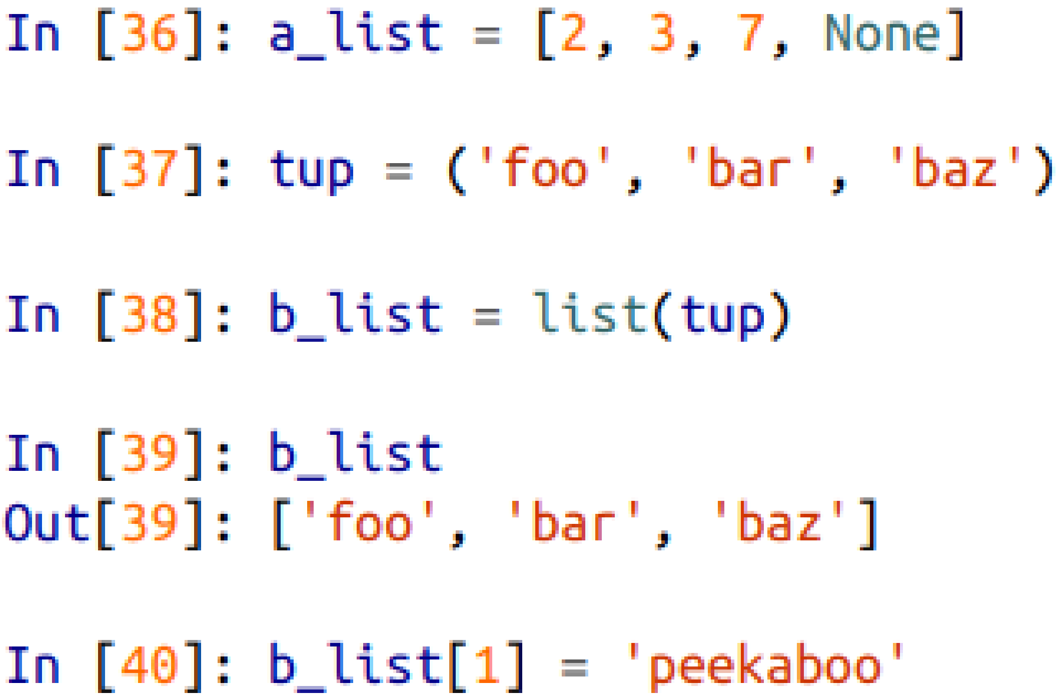
```
In [21]: a, b = 1, 2

In [22]: a
Out[22]: 1

In [23]: b
Out[23]: 2

In [24]: b, a = a, b

In [25]: a
Out[25]: 2

In [26]: b
Out[26]: 1
```

*rest

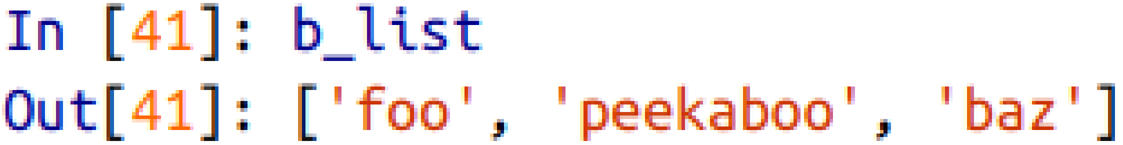


```
In [29]: values = 1, 2, 3, 4, 5

In [30]: a, b, *rest = values

In [31]: a, b
Out[31]: (1, 2)

In [32]: rest
Out[32]: [3, 4, 5]
```

List



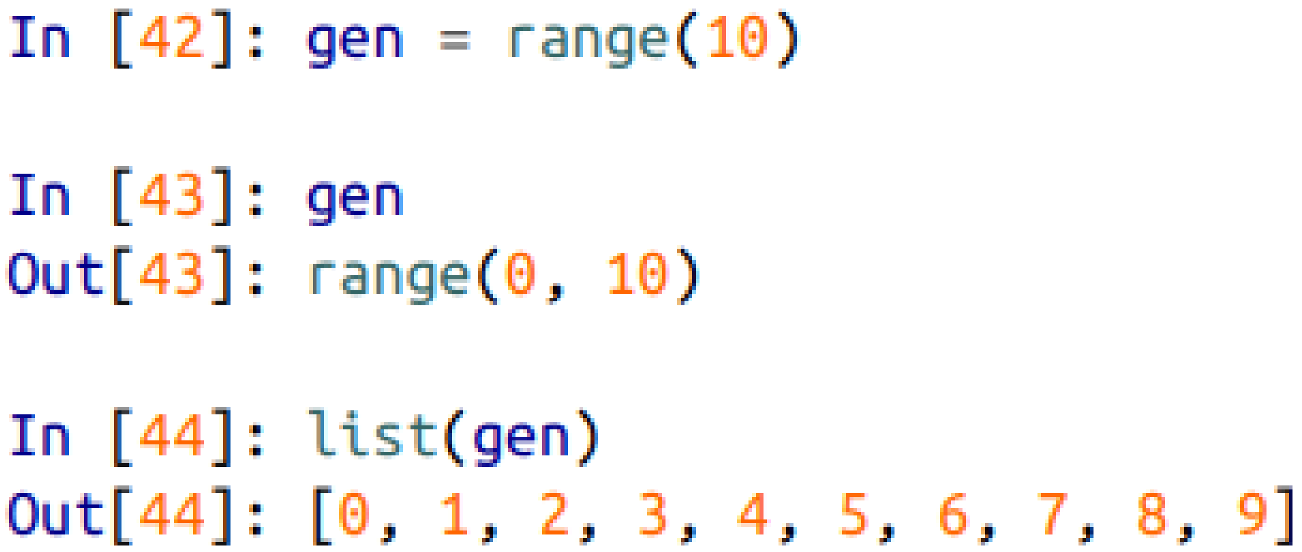
```
In [36]: a_list = [2, 3, 7, None]

In [37]: tup = ('foo', 'bar', 'baz')

In [38]: b_list = list(tup)

In [39]: b_list
Out[39]: ['foo', 'bar', 'baz']

In [40]: b_list[1] = 'peekaboo'
```



```
In [41]: b_list
Out[41]: ['foo', 'peekaboo', 'baz']
```

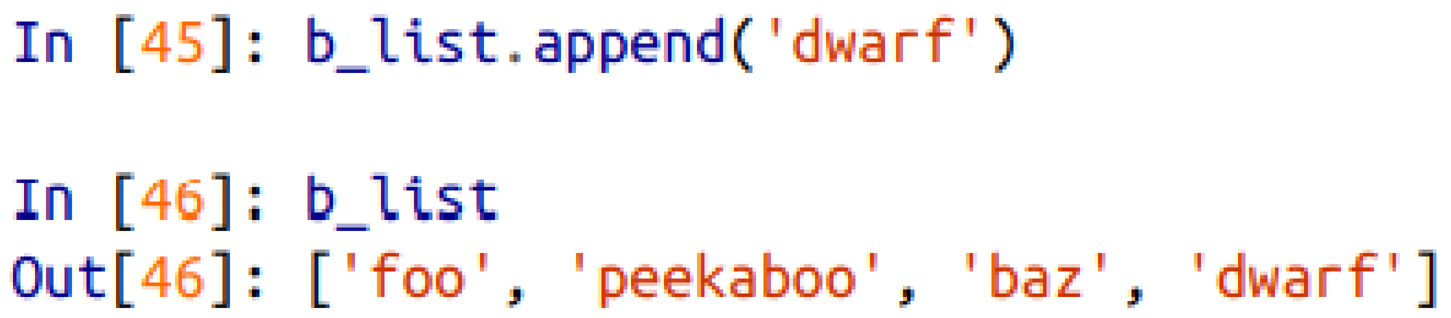
List

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

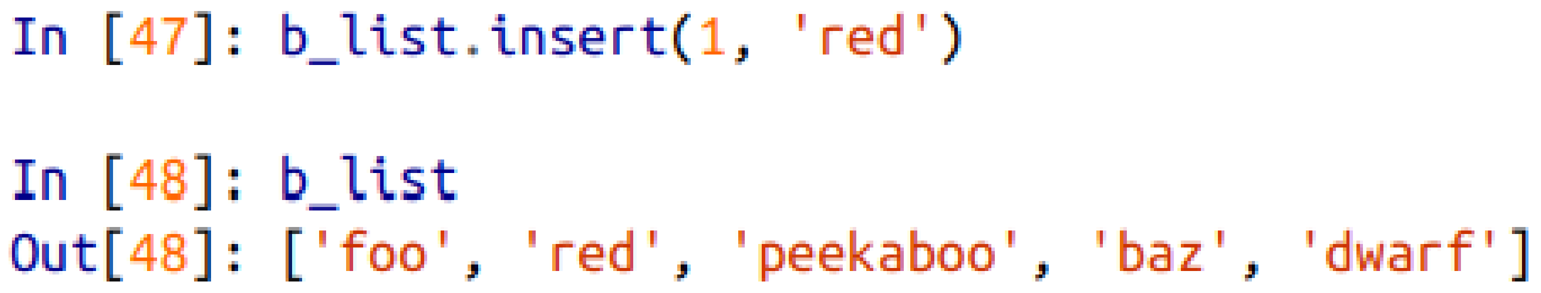


```
In [42]: gen = range(10)

In [43]: gen
Out[43]: range(0, 10)

In [44]: list(gen)
Out[44]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Adding and removing elements

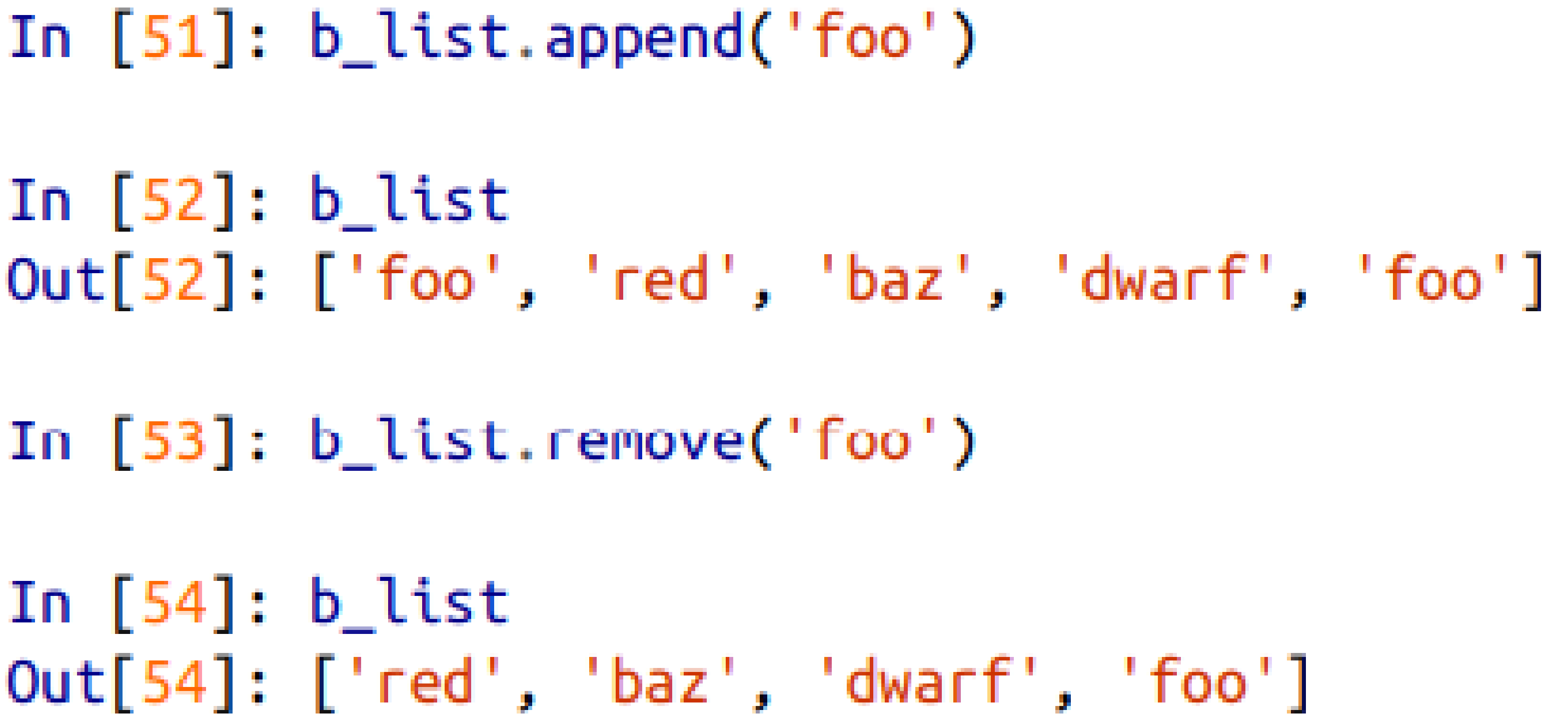


```
In [45]: b_list.append('dwarf')

In [46]: b_list
Out[46]: ['foo', 'peekaboo', 'baz', 'dwarf']
```



```
In [47]: b_list.insert(1, 'red')

In [48]: b_list
Out[48]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

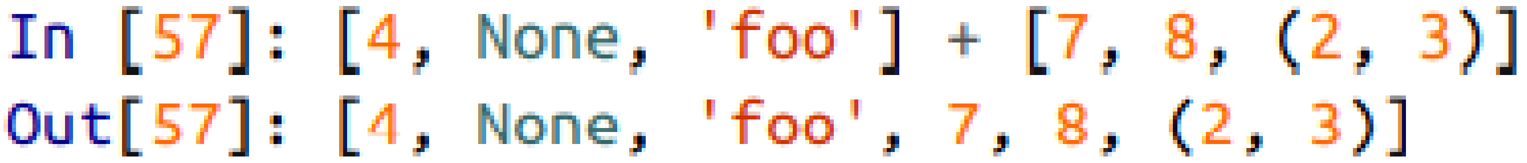
- Elements can be removed by value with remove, which locates the first such value and removes it from the last:



```
In [51]: b_list.append('foo')

In [52]: b_list
Out[52]: ['foo', 'red', 'baz', 'dwarf', 'foo']

In [53]: b_list.remove('foo')

In [54]: b_list
Out[54]: ['red', 'baz', 'dwarf', 'foo']
```

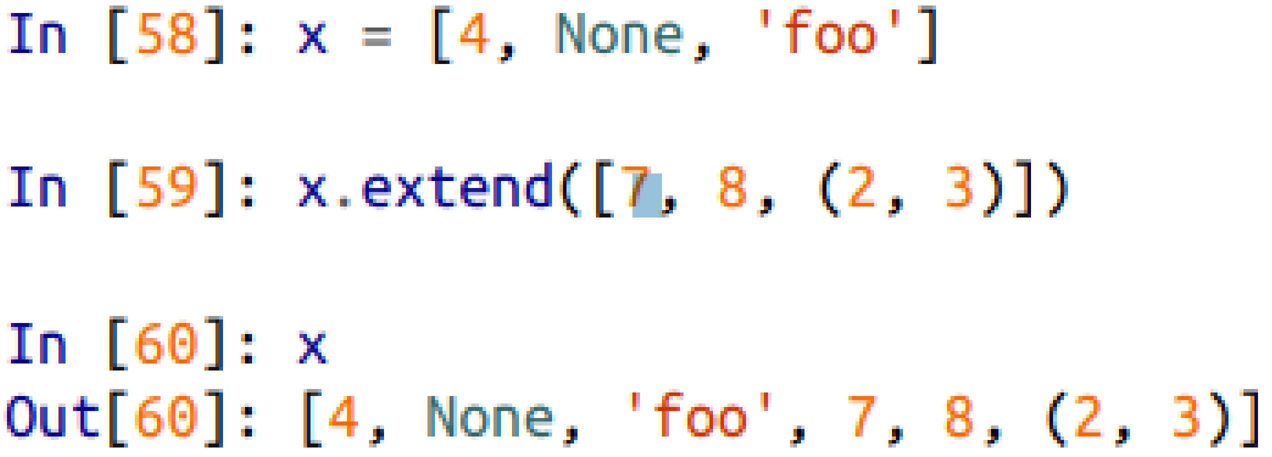
Concatenating and combining lists

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



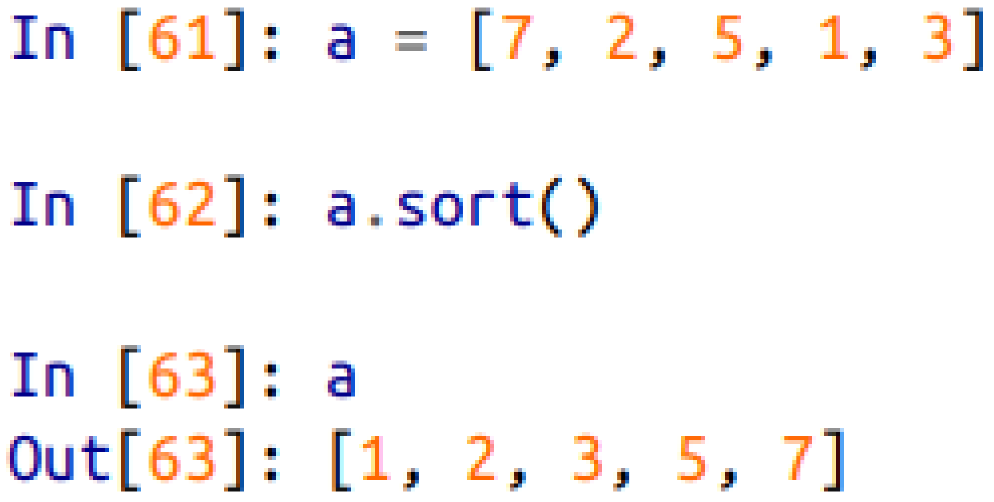
```
In [57]: [4, None, 'foo'] + [7, 8, (2, 3)]
Out[57]: [4, None, 'foo', 7, 8, (2, 3)]
```

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

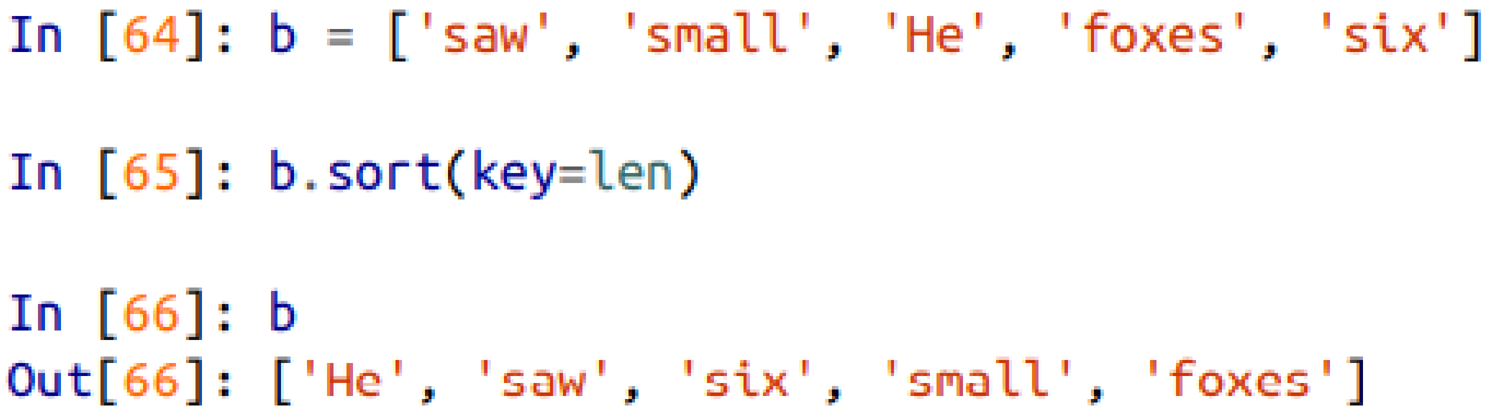


```
In [58]: x = [4, None, 'foo']

In [59]: x.extend([7, 8, (2, 3)])

In [60]: x
Out[60]: [4, None, 'foo', 7, 8, (2, 3)]
```

Sorting

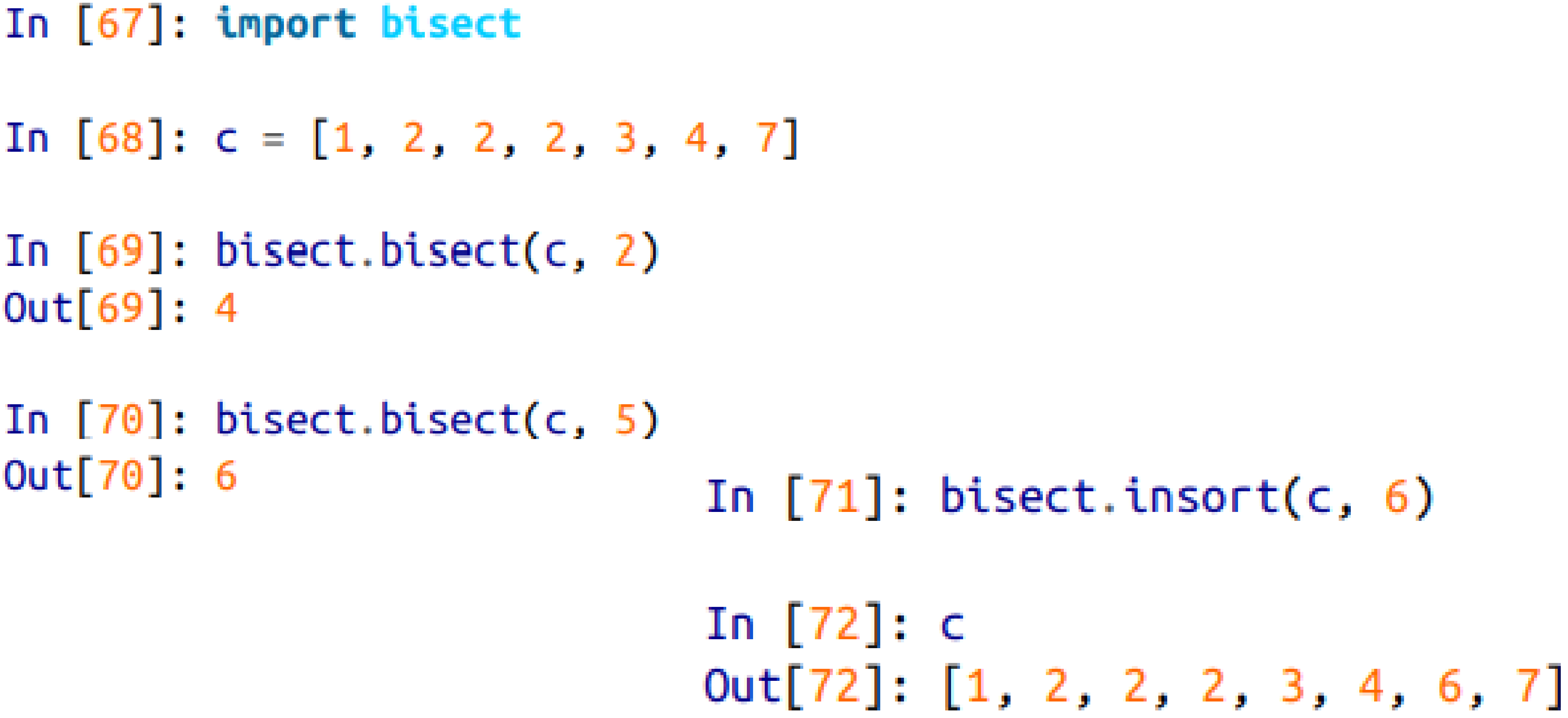


```
In [61]: a = [7, 2, 5, 1, 3]

In [62]: a.sort()

In [63]: a
Out[63]: [1, 2, 3, 5, 7]
```



```
In [64]: b = ['saw', 'small', 'He', 'foxes', 'six']

In [65]: b.sort(key=len)

In [66]: b
Out[66]: ['He', 'saw', 'six', 'small', 'foxes']
```

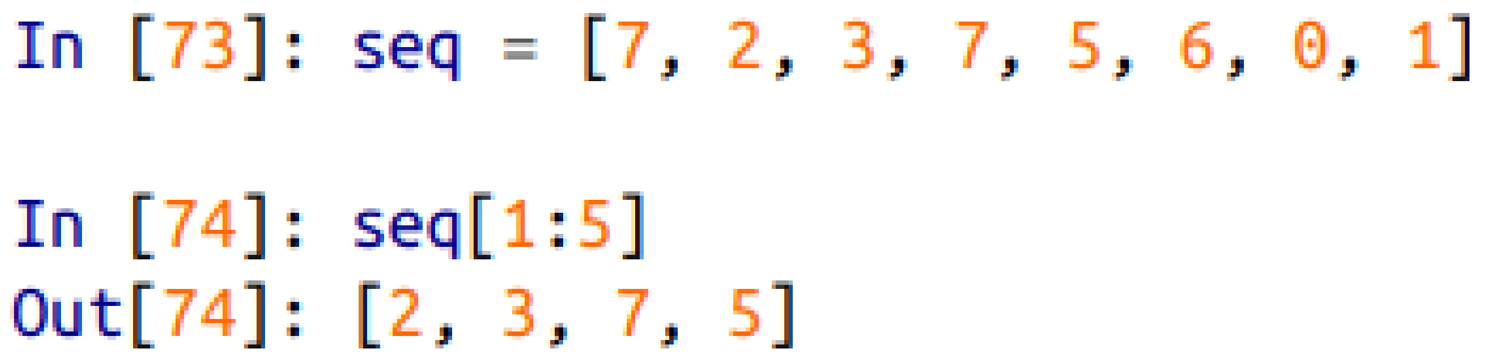
Binary search and maintaining a sorted list



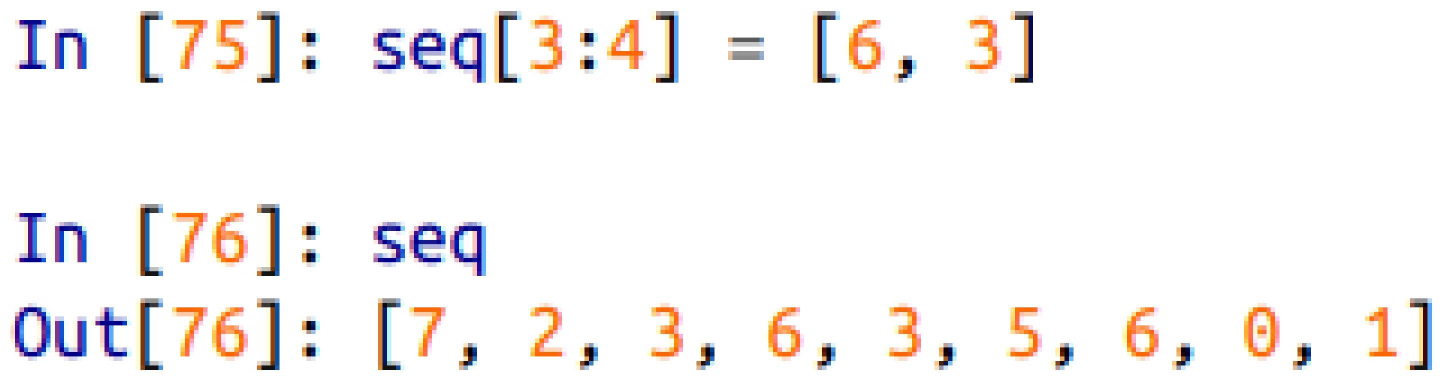
```
In [67]: import bisect

In [68]: c = [1, 2, 2, 2, 3, 4, 7]

In [69]: bisect.bisect(c, 2)
Out[69]: 4

In [70]: bisect.bisect(c, 5)
Out[70]: 6
```

```
In [71]: bisect.insort(c, 6)

In [72]: c
Out[72]: [1, 2, 2, 2, 3, 4, 6, 7]
```

Slicing

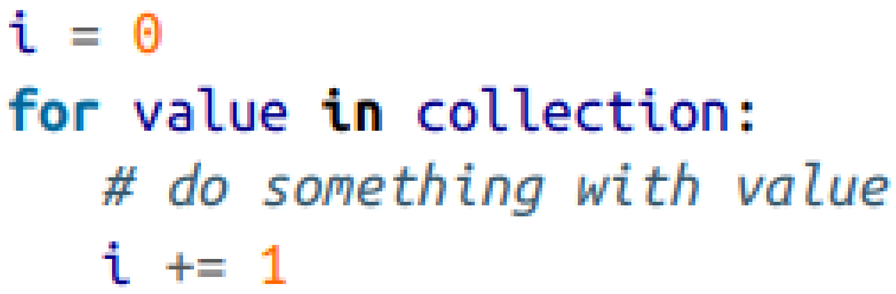


```
In [73]: seq = [7, 2, 3, 7, 5, 6, 0, 1]

In [74]: seq[1:5]
Out[74]: [2, 3, 7, 5]
```

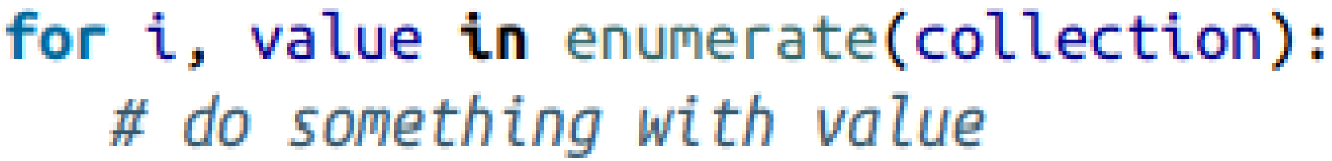


```
In [75]: seq[3:4] = [6, 3]

In [76]: seq
Out[76]: [7, 2, 3, 6, 3, 5, 6, 0, 1]
```

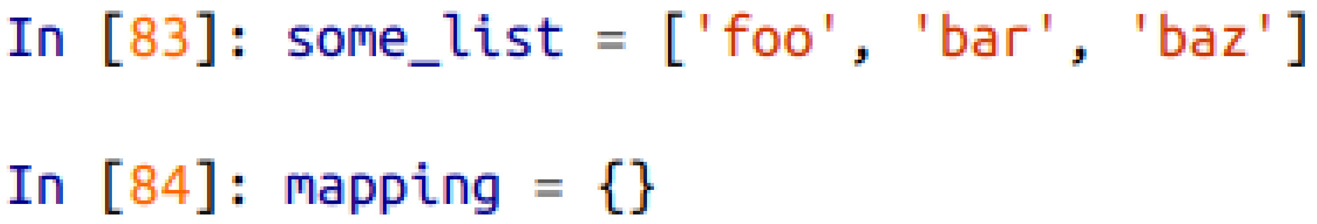
enumerate



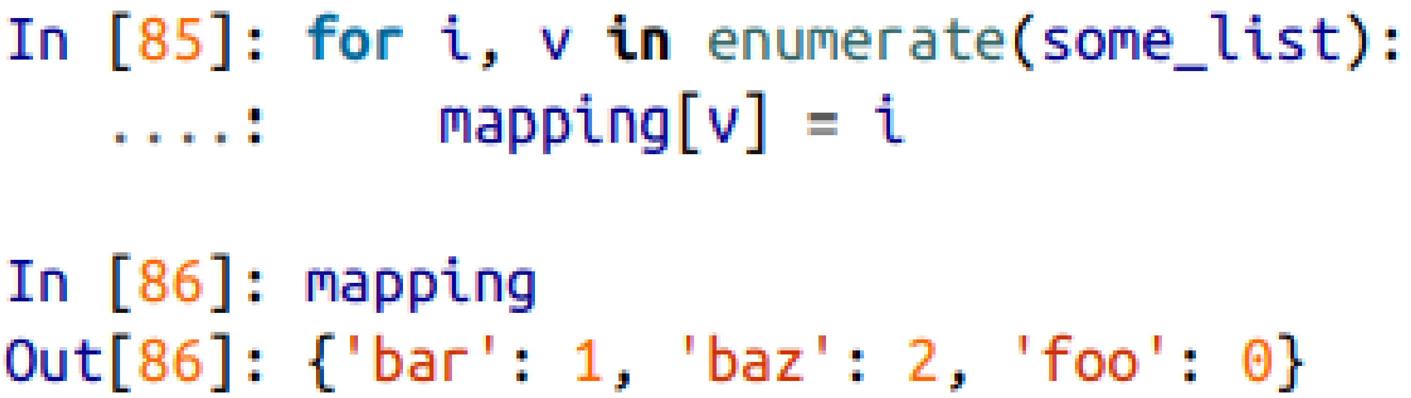
```python
i = 0
for value in collection:
    # do something with value
    i += 1
```



```python
for i, value in enumerate(collection):
    # do something with value
```



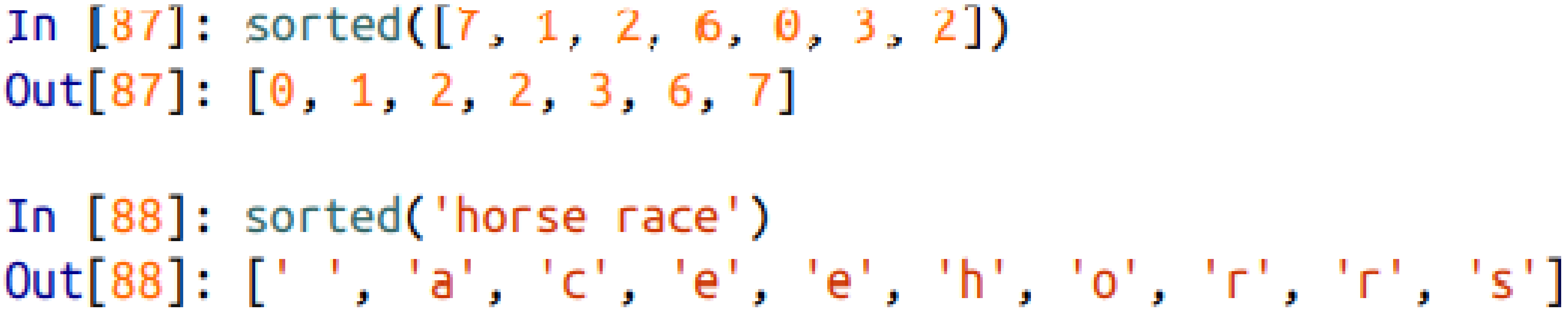
```
In [83]: some_list = ['foo', 'bar', 'baz']

In [84]: mapping = {}
```



```
In [85]: for i, v in enumerate(some_list):
   ....:     mapping[v] = i

In [86]: mapping
Out[86]: {'bar': 1, 'baz': 2, 'foo': 0}
```

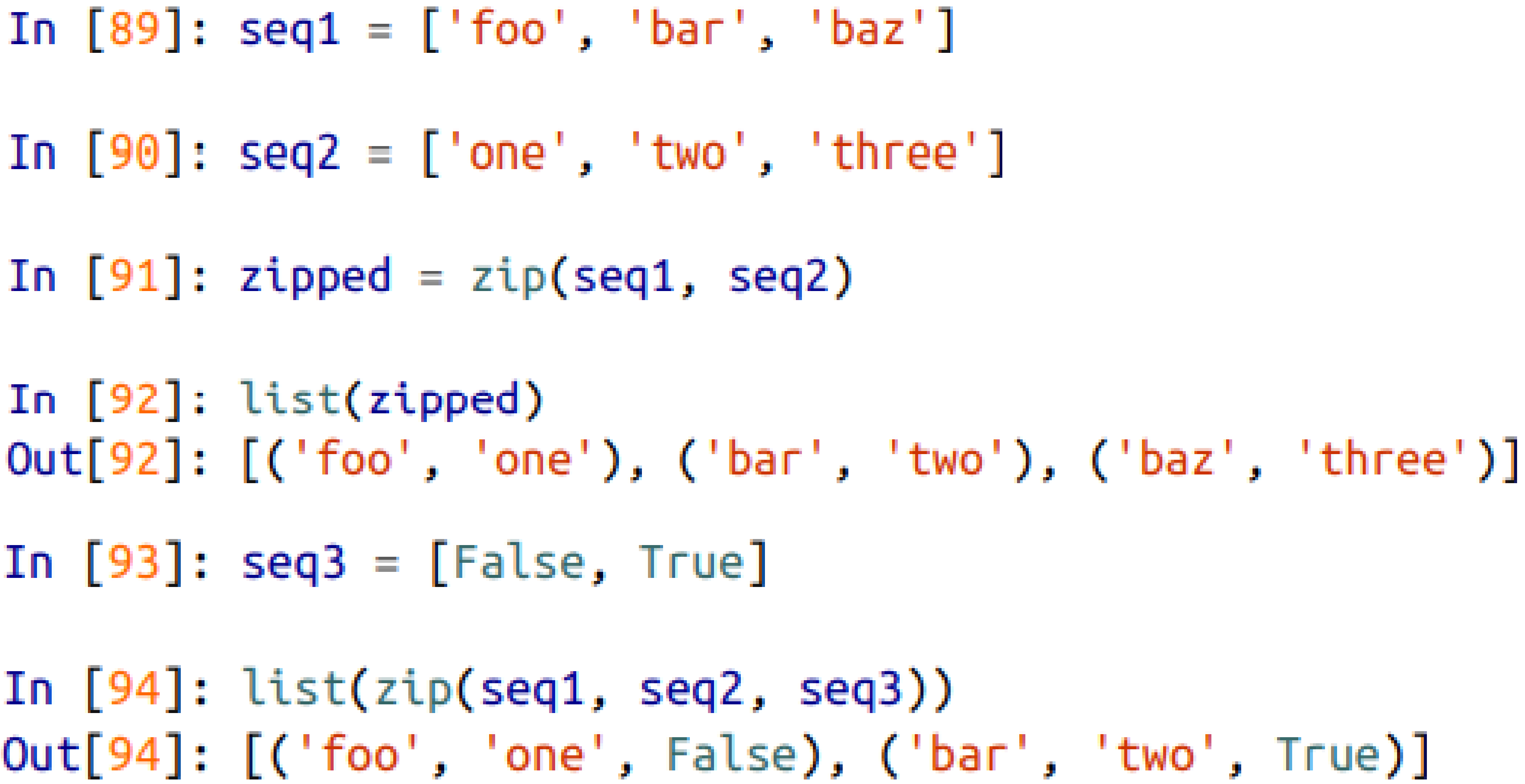
sorted

- The sorted function returns a new sorted list from the elements of any sequence:

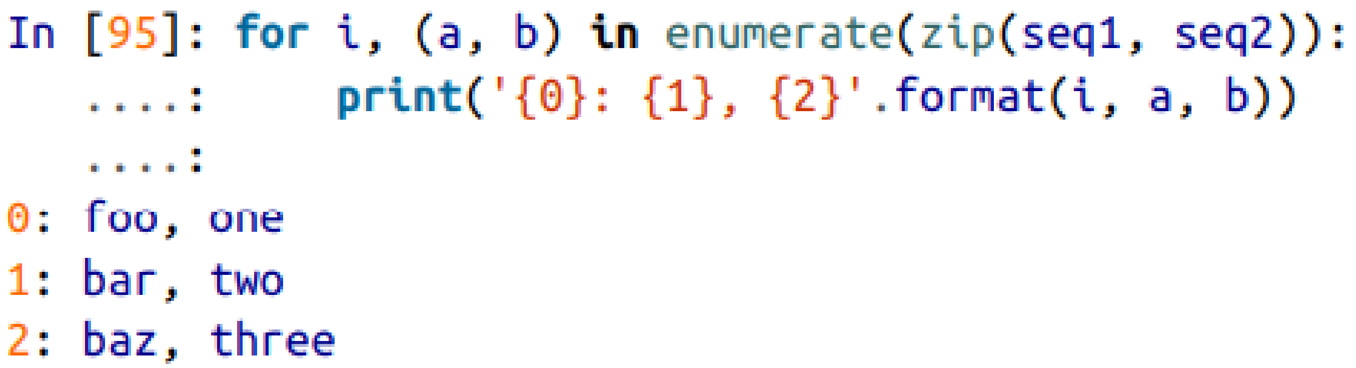


```
In [87]: sorted([7, 1, 2, 6, 0, 3, 2])
Out[87]: [0, 1, 2, 2, 3, 6, 7]

In [88]: sorted('horse race')
Out[88]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

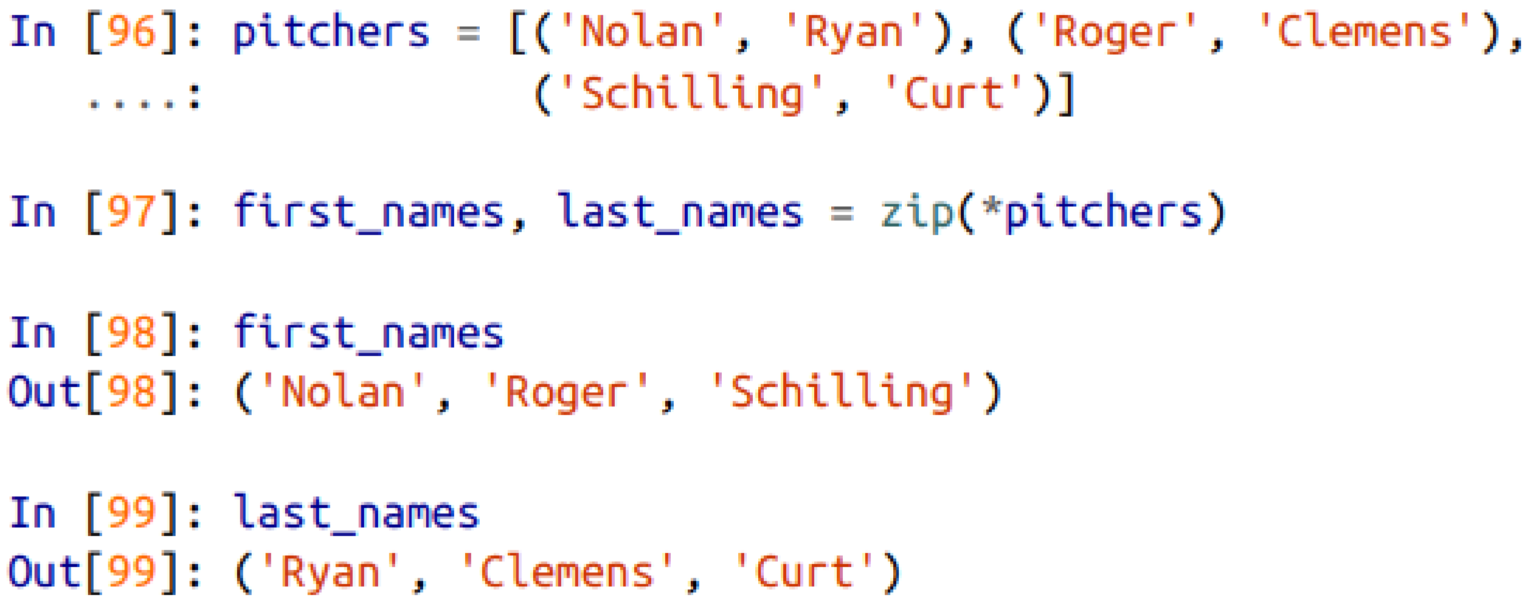
zip



```
In [89]: seq1 = ['foo', 'bar', 'baz']

In [90]: seq2 = ['one', 'two', 'three']

In [91]: zipped = zip(seq1, seq2)

In [92]: list(zipped)
Out[92]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]

In [93]: seq3 = [False, True]

In [94]: list(zip(seq1, seq2, seq3))
Out[94]: [('foo', 'one', False), ('bar', 'two', True)]
```

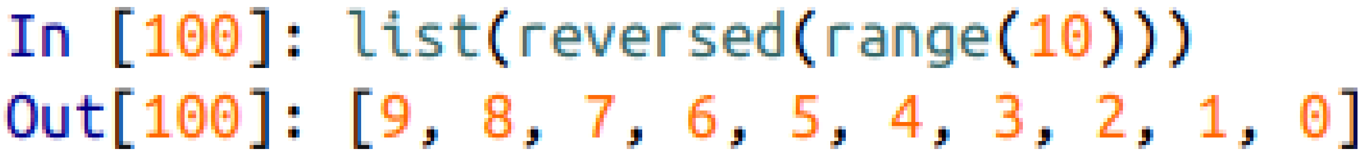


```
In [95]: for i, (a, b) in enumerate(zip(seq1, seq2)):
   ....:     print('{0}: {1}, {2}'.format(i, a, b))
   ....:
0: foo, one
1: bar, two
2: baz, three
```



```
In [96]: pitchers = [('Nolan', 'Ryan'), ('Roger', 'Clemens'),
   ....:             ('Schilling', 'Curt')]

In [97]: first_names, last_names = zip(*pitchers)

In [98]: first_names
Out[98]: ('Nolan', 'Roger', 'Schilling')

In [99]: last_names
Out[99]: ('Ryan', 'Clemens', 'Curt')
```

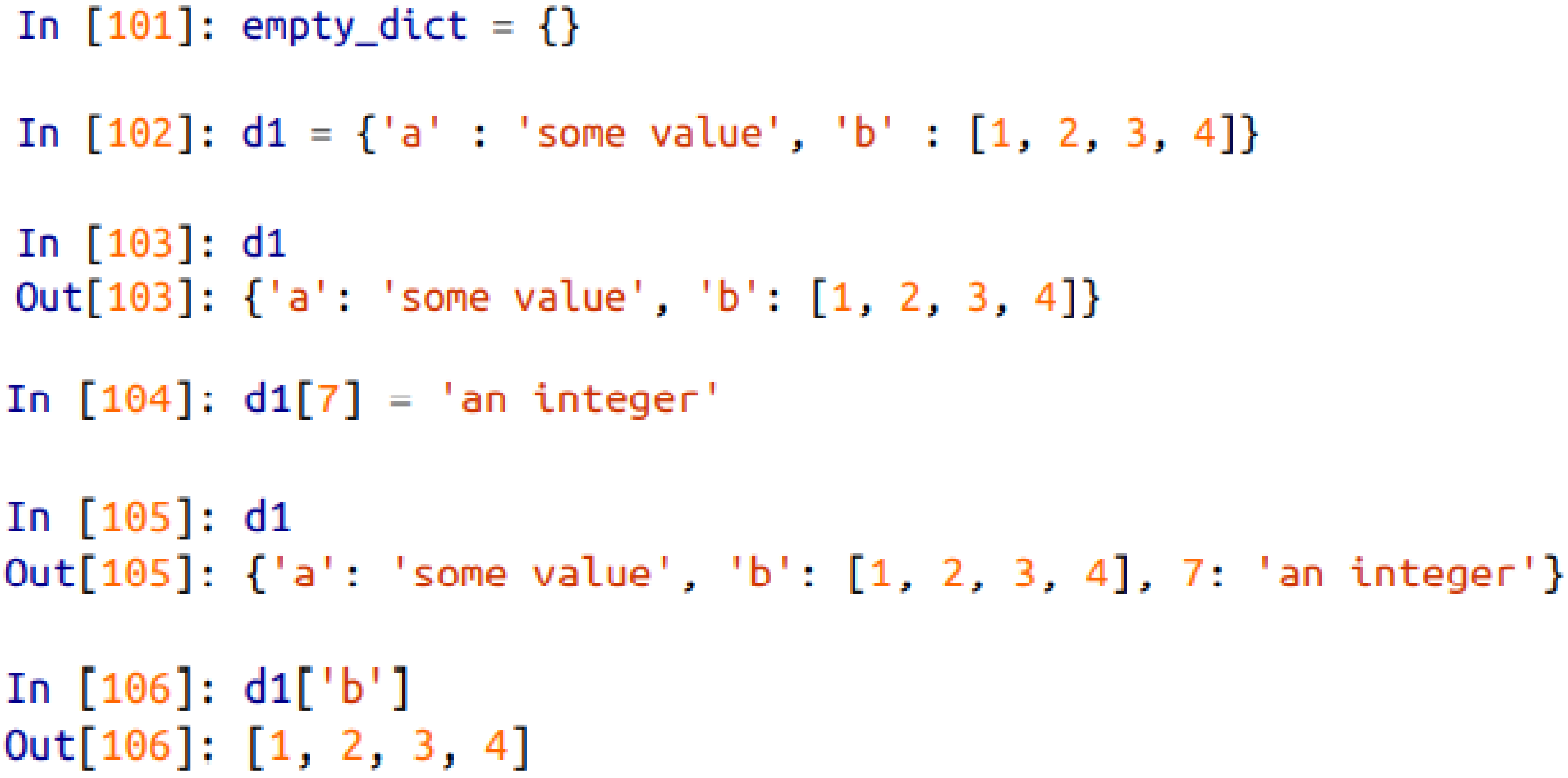
reserved



```
In [100]: list(reversed(range(10)))
Out[100]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

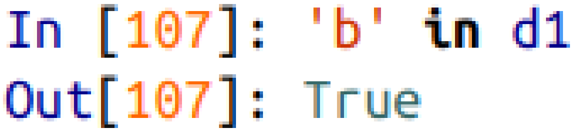
Keep in mind that reversed is a generator (to be discussed in some more detail later), so it does not create the reversed sequence until materialized (e.g., with list or a for loop).

dict



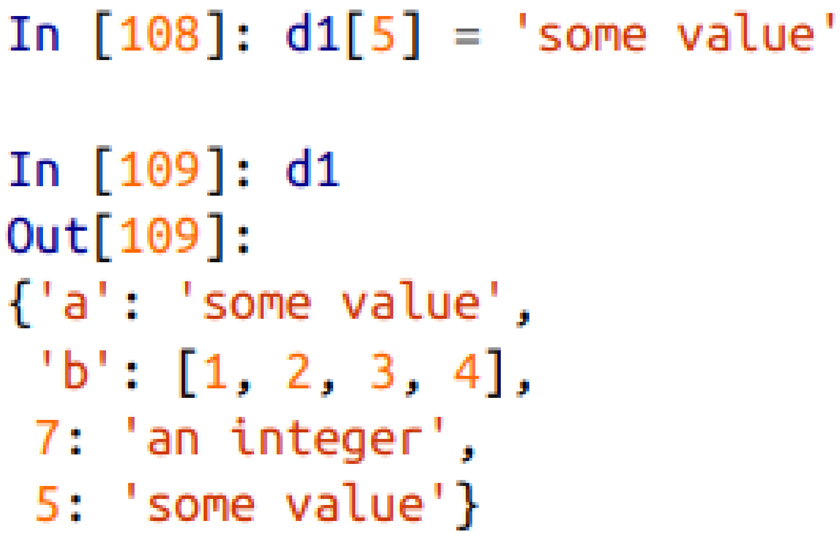
```
In [101]: empty_dict = {}

In [102]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}

In [103]: d1
Out[103]: {'a': 'some value', 'b': [1, 2, 3, 4]}

In [104]: d1[7] = 'an integer'

In [105]: d1
Out[105]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}

In [106]: d1['b']
Out[106]: [1, 2, 3, 4]
```

- You can check if a dict contains a key using the same syntax used for checking whether a list or tuple contains a value:

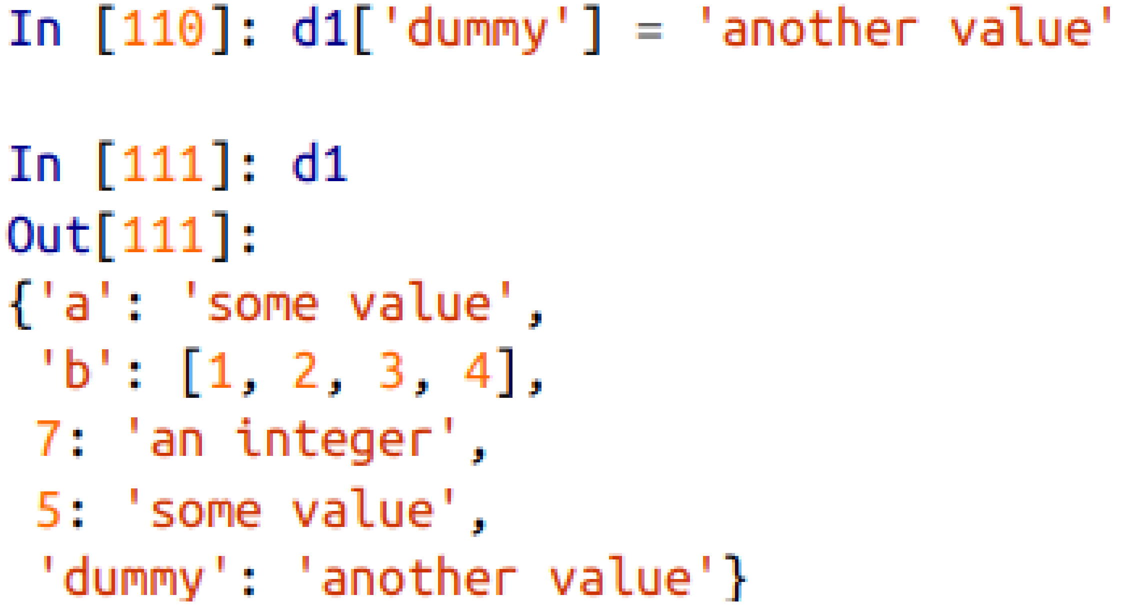


```
In [107]: 'b' in d1
Out[107]: True
```

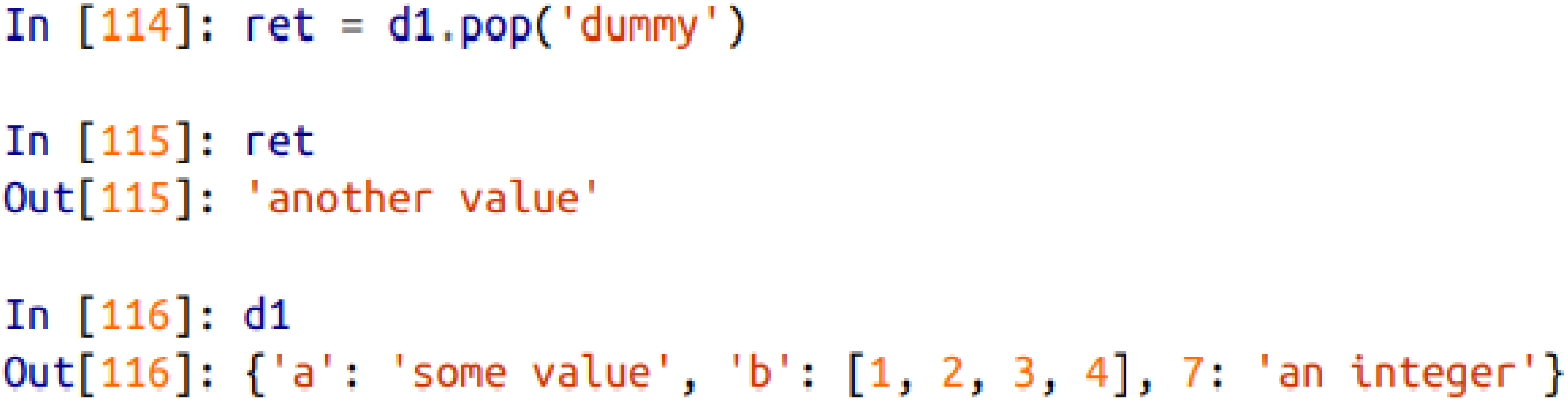
- You can delete values either using the del keyword or the pop method (which simultaneously returns the value and deletes the key)



```
In [108]: d1[5] = 'some value'

In [109]: d1
Out[109]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 5: 'some value'}
```

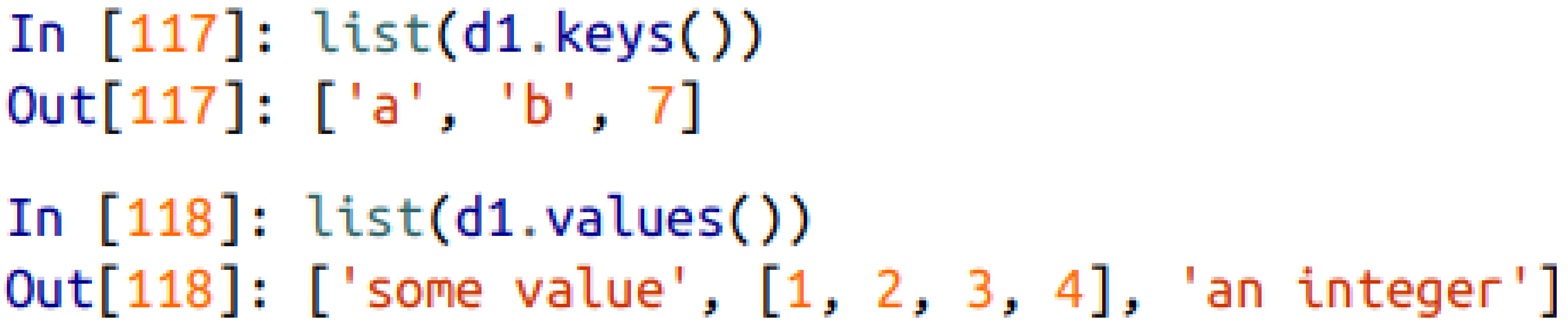


```
In [110]: d1['dummy'] = 'another value'

In [111]: d1
Out[111]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 5: 'some value',
 'dummy': 'another value'}
```



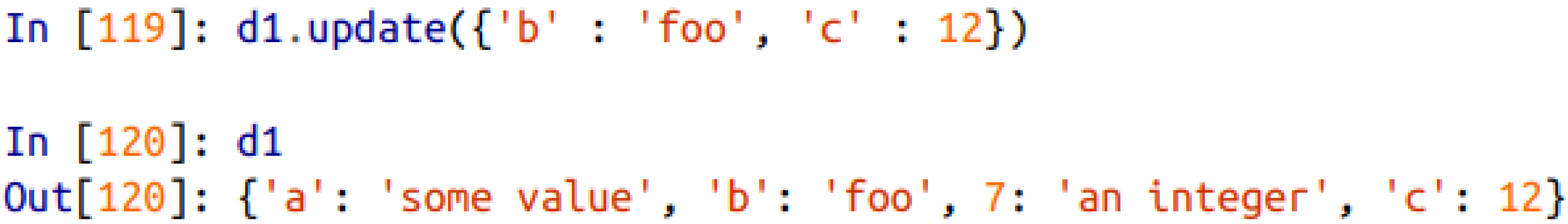
```
In [114]: ret = d1.pop('dummy')

In [115]: ret
Out[115]: 'another value'

In [116]: d1
Out[116]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

- The keys and values method give you iterators of the dict’s keys and values, respectively. While the key-value pairs are not in any particular order, these functions out- put the keys and values in the same order:



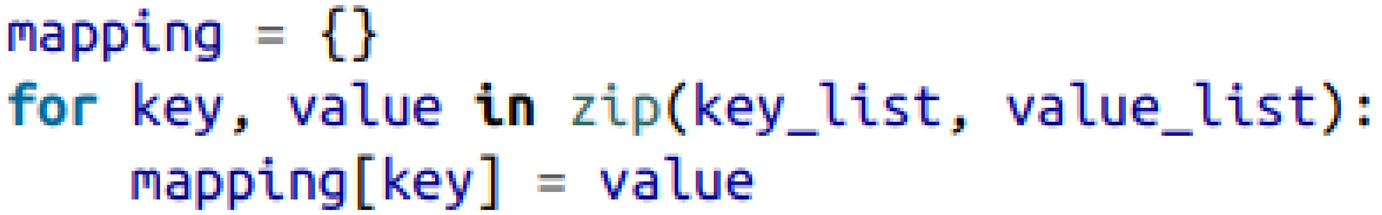
```
In [117]: list(d1.keys())
Out[117]: ['a', 'b', 7]

In [118]: list(d1.values())
Out[118]: ['some value', [1, 2, 3, 4], 'an integer']
```

You can merge one dict into another using the update

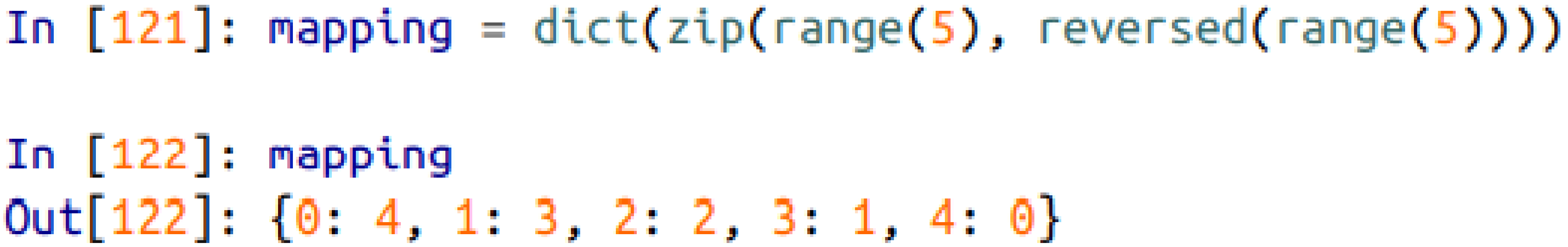


```
In [119]: d1.update({'b' : 'foo', 'c' : 12})

In [120]: d1
Out[120]: {'a': 'some value', 'b': 'foo', 7: 'an integer', 'c': 12}
```

Creating dicts from sequences

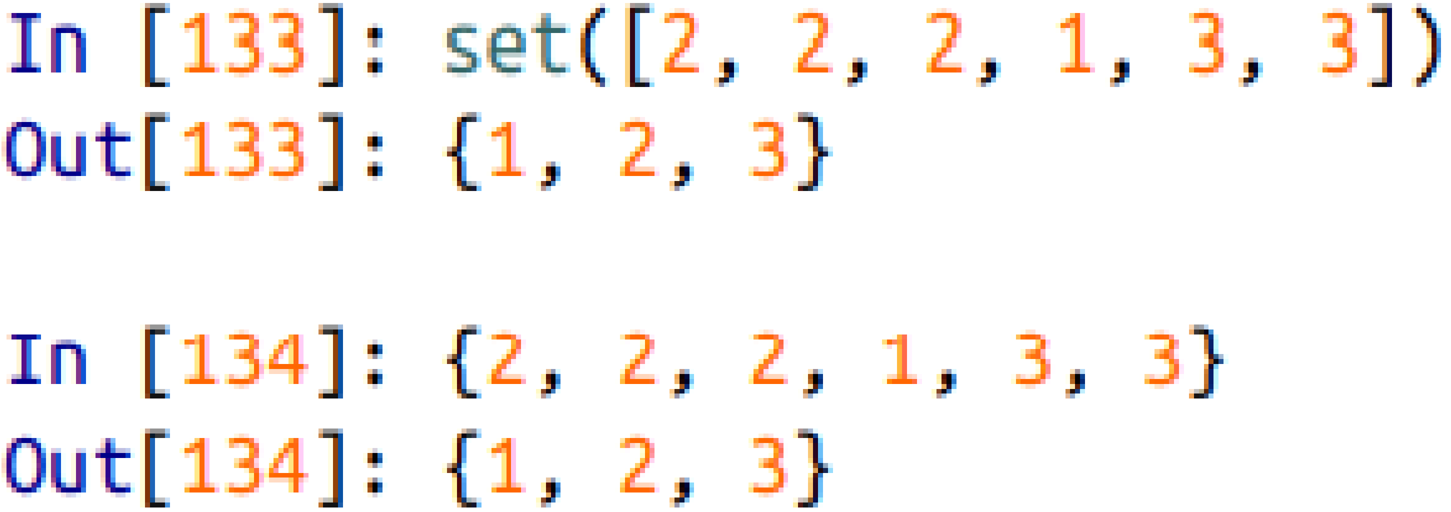


```
mapping = {}
for key, value in zip(key_list, value_list):
    mapping[key] = value
```



```
In [121]: mapping = dict(zip(range(5), reversed(range(5))))

In [122]: mapping
Out[122]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

set

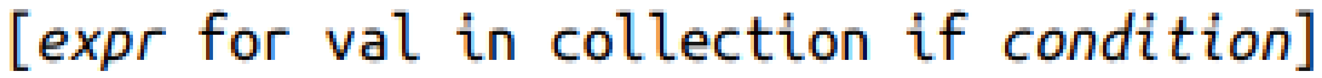


```
In [133]: set([2, 2, 2, 1, 3, 3])
Out[133]: {1, 2, 3}

In [134]: {2, 2, 2, 1, 3, 3}
Out[134]: {1, 2, 3}
```

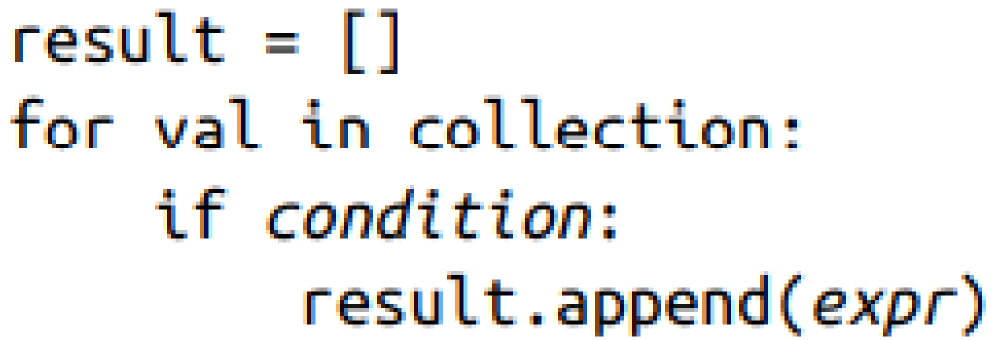
Function	Alternative syntax	Description
`a.add(x)`	N/A	Add element `x` to the set `a`
`a.clear()`	N/A	Reset the set `a` to an empty state, discarding all of its elements
`a.remove(x)`	N/A	Remove element `x` from the set `a`
`a.pop()`	N/A	Remove an arbitrary element from the set `a`, raising `KeyError` if the set is empty
`a.union(b)`	`a | b`	All of the unique elements in `a` and `b`
`a.update(b)`	`a |= b`	Set the contents of `a` to be the union of the elements in `a` and `b`
`a.intersection(b)`	`a & b`	All of the elements in *both* `a` and `b`
`a.intersection_update(b)`	`a &= b`	Set the contents of `a` to be the intersection of the elements in `a` and `b`
`a.difference(b)`	`a - b`	The elements in `a` that are not in `b`
`a.difference_update(b)`	`a -= b`	Set `a` to the elements in `a` that are not in `b`
`a.symmetric_difference(b)`	`a ^ b`	All of the elements in either `a` or `b` but *not both*
`a.symmetric_difference_update(b)`	`a ^= b`	Set `a` to contain the elements in either `a` or `b` but *not both*
`a.issubset(b)`	N/A	`True` if the elements of `a` are all contained in `b`
`a.issuperset(b)`	N/A	`True` if the elements of `b` are all contained in `a`
`a.isdisjoint(b)`	N/A	`True` if `a` and `b` have no elements in common

List comprehension

- List comprehensions are one of the most-loved Python language features. They allow you to concisely form a new list by filtering the elements of a collection, transforming the elements passing the filter in one concise expression.

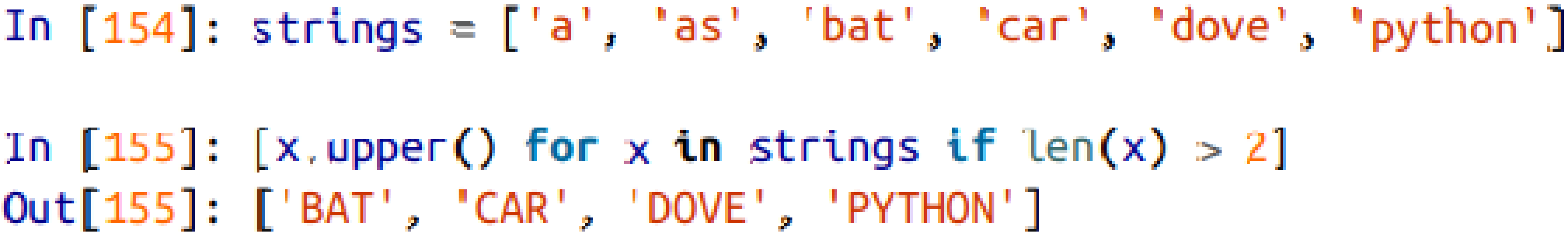


```
[expr for val in collection if condition]
```



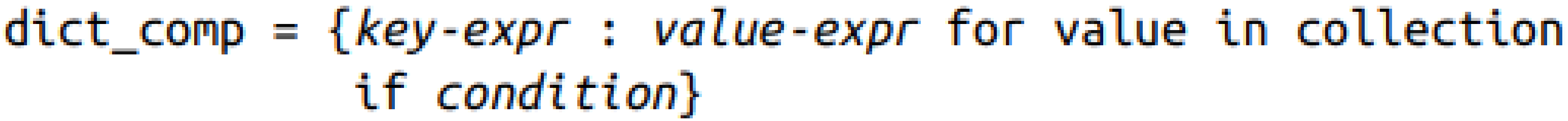
```
result = []
for val in collection:
    if condition:
        result.append(expr)
```

Example

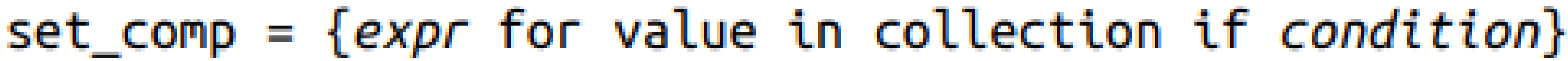


```
In [154]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']

In [155]: [x.upper() for x in strings if len(x) > 2]
Out[155]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

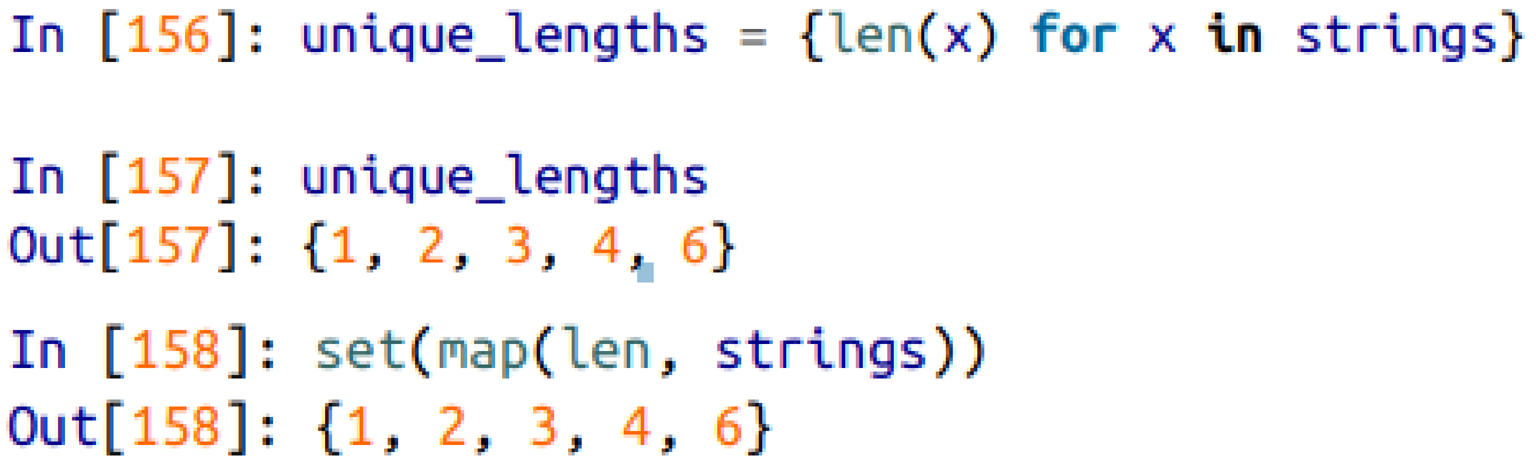
Dictionary and set comprehension



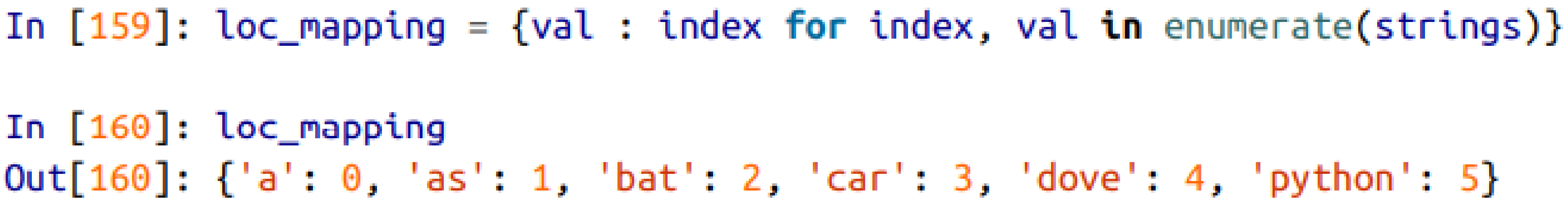
```
dict_comp = {key-expr : value-expr for value in collection
             if condition}
```



```
set_comp = {expr for value in collection if condition}
```

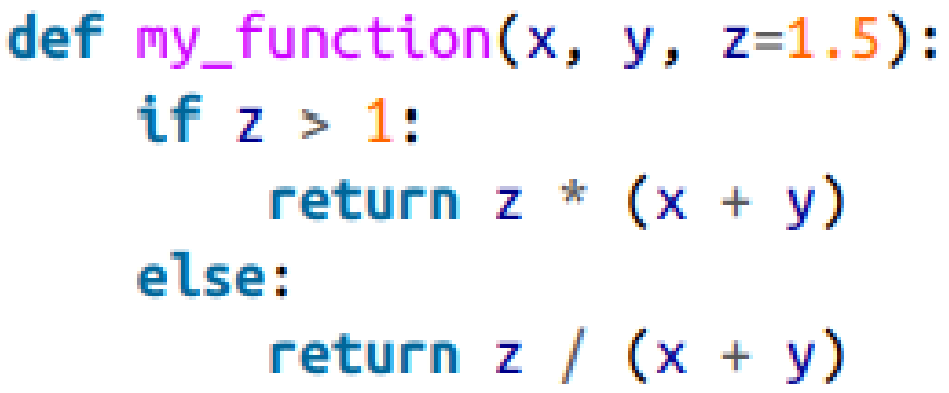


```
In [156]: unique_lengths = {len(x) for x in strings}

In [157]: unique_lengths
Out[157]: {1, 2, 3, 4, 6}

In [158]: set(map(len, strings))
Out[158]: {1, 2, 3, 4, 6}
```

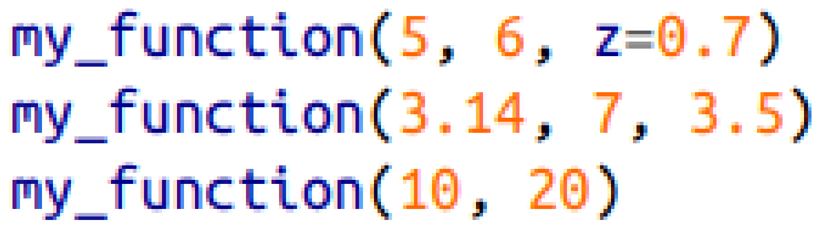


```
In [159]: loc_mapping = {val : index for index, val in enumerate(strings)}

In [160]: loc_mapping
Out[160]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

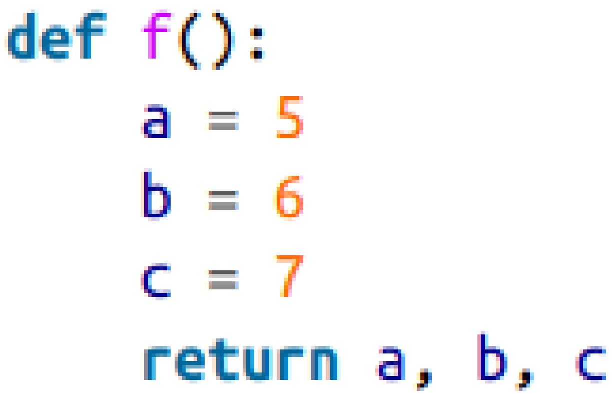
Functions



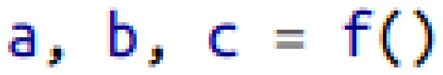
```python
def my_function(x, y, z=1.5):
    if z > 1:
        return z * (x + y)
    else:
        return z / (x + y)
```



```python
my_function(5, 6, z=0.7)
my_function(3.14, 7, 3.5)
my_function(10, 20)
```

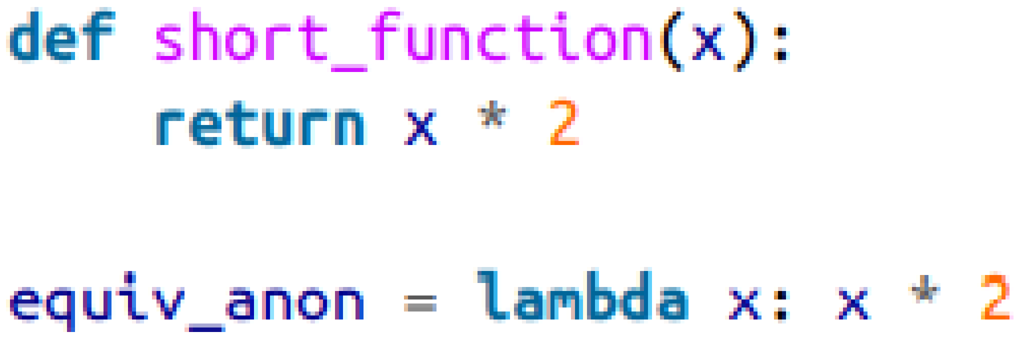


```python
def f():
    a = 5
    b = 6
    c = 7
    return a, b, c
```

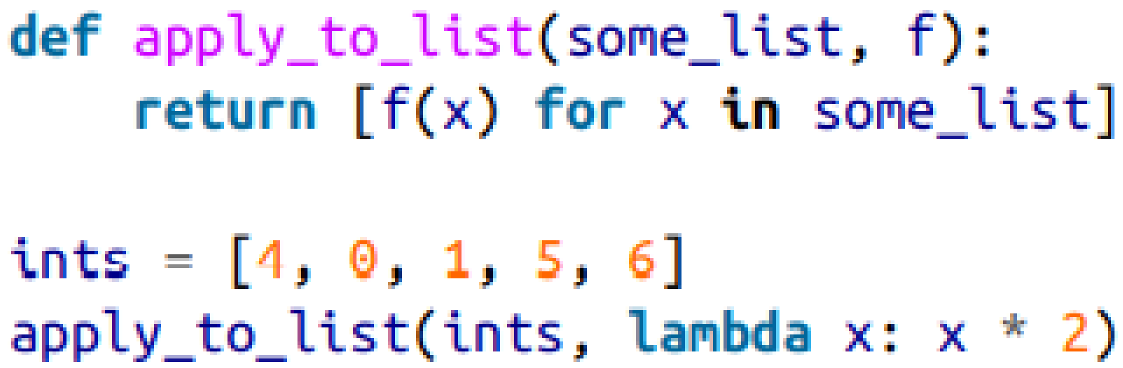


```python
a, b, c = f()
```

Anonymous (Lambda) Functions

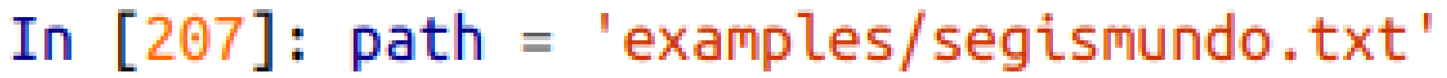


```python
def short_function(x):
    return x * 2

equiv_anon = lambda x: x * 2
```

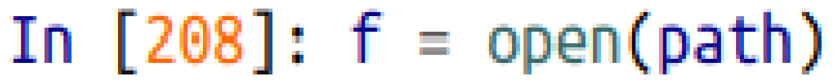


```python
def apply_to_list(some_list, f):
    return [f(x) for x in some_list]

ints = [4, 0, 1, 5, 6]
apply_to_list(ints, lambda x: x * 2)
```

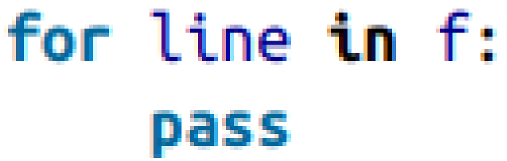
Files



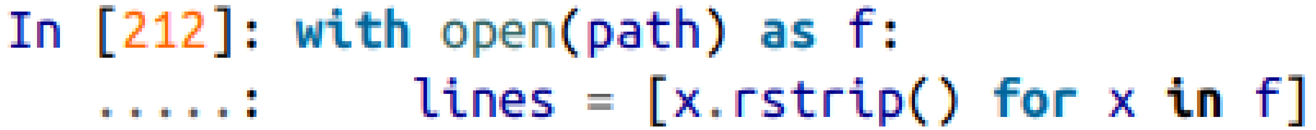
```
In [207]: path = 'examples/segismundo.txt'
```



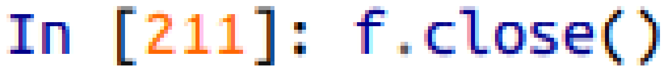
```
In [208]: f = open(path)
```



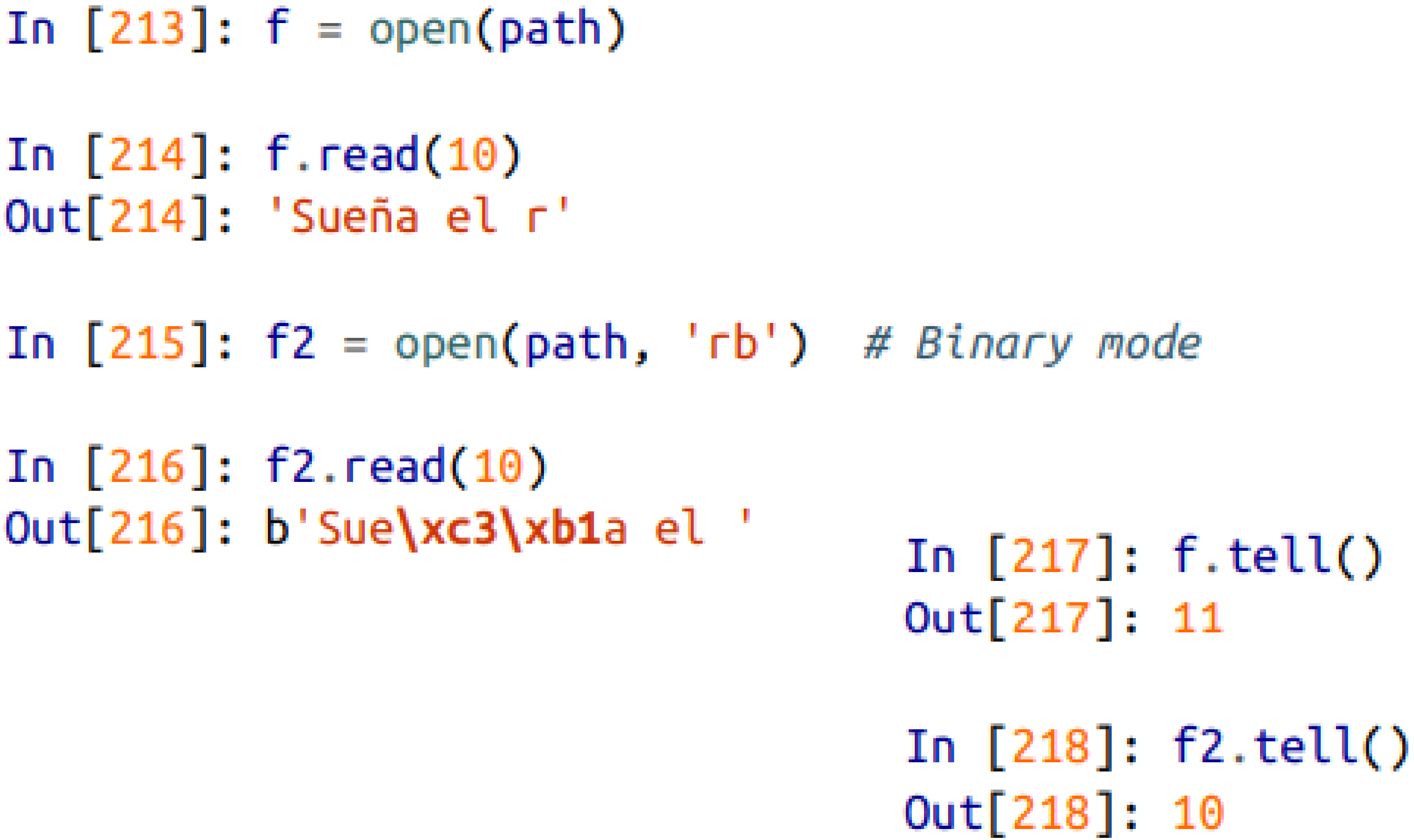
```
for line in f:
    pass
```



```
In [212]: with open(path) as f:
   .....:     lines = [x.rstrip() for x in f]
```



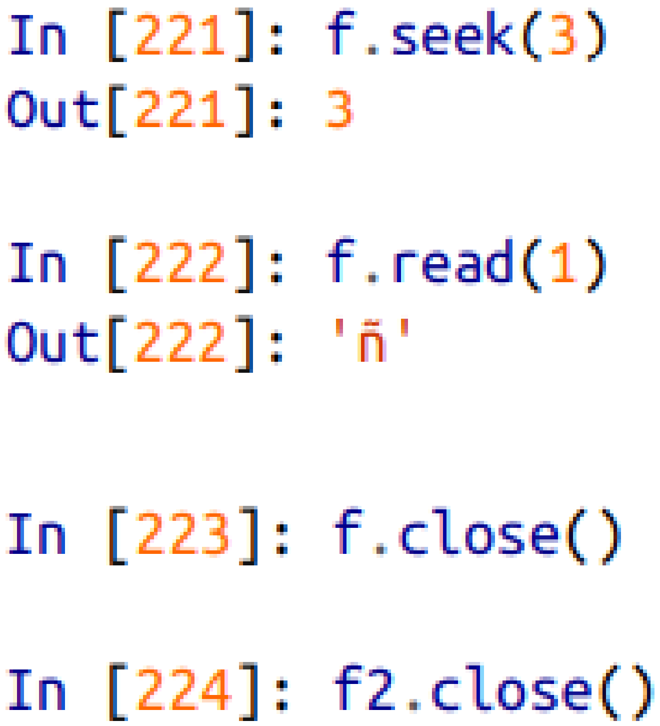
```
In [211]: f.close()
```



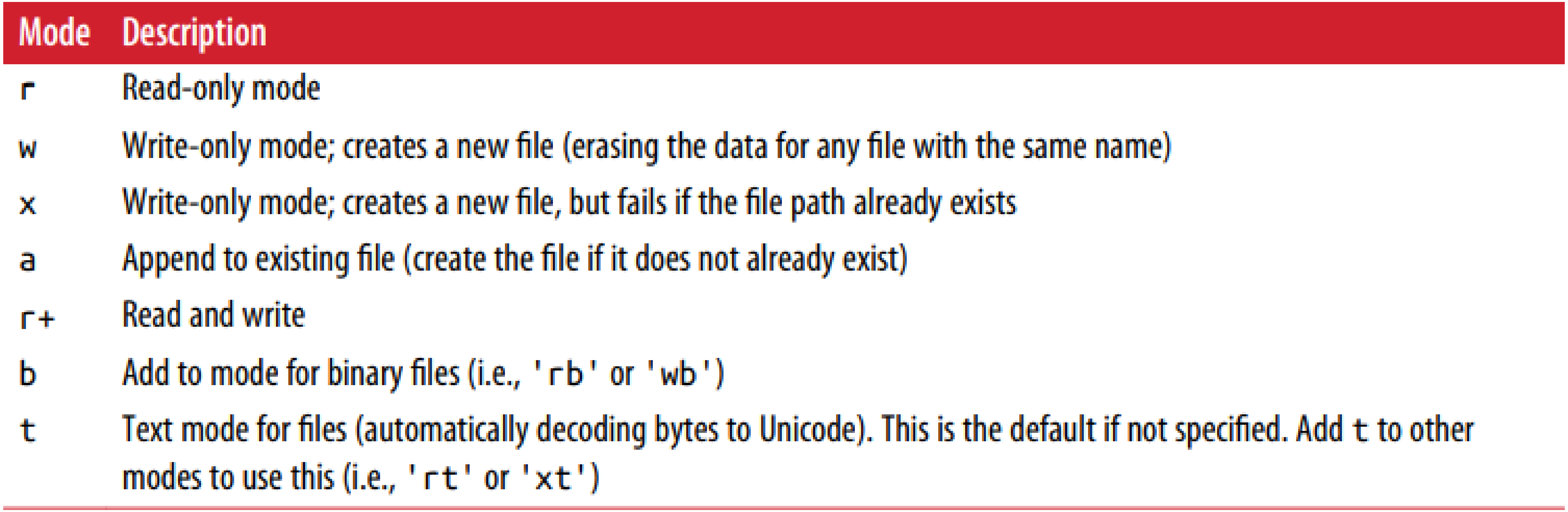
```
In [213]: f = open(path)

In [214]: f.read(10)
Out[214]: 'Sueña el r'

In [215]: f2 = open(path, 'rb')  # Binary mode

In [216]: f2.read(10)
Out[216]: b'Sue\xc3\xb1a el '
```

```
In [217]: f.tell()
Out[217]: 11

In [218]: f2.tell()
Out[218]: 10
```

- seek changes the file position to the indicated byte in the file:

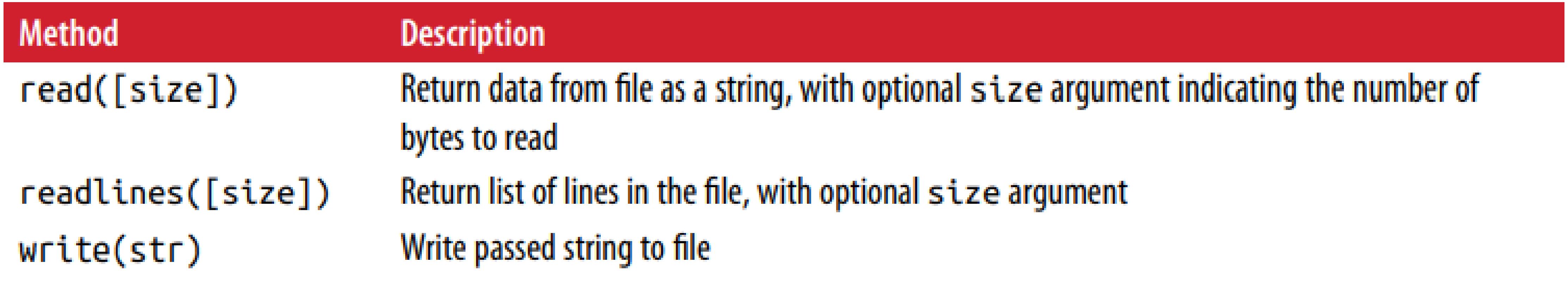


```
In [221]: f.seek(3)
Out[221]: 3

In [222]: f.read(1)
Out[222]: 'ñ'


In [223]: f.close()

In [224]: f2.close()
```

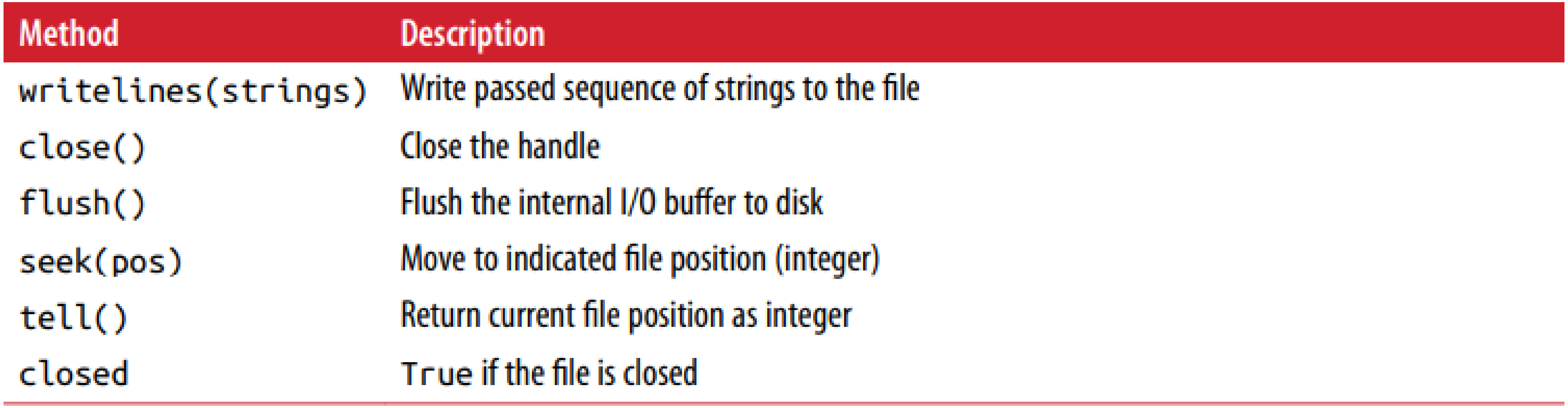


Mode	Description
`r`	Read-only mode
`w`	Write-only mode; creates a new file (erasing the data for any file with the same name)
`x`	Write-only mode; creates a new file, but fails if the file path already exists
`a`	Append to existing file (create the file if it does not already exist)
`r+`	Read and write
`b`	Add to mode for binary files (i.e., `'rb'` or `'wb'`)
`t`	Text mode for files (automatically decoding bytes to Unicode). This is the default if not specified. Add `t` to other modes to use this (i.e., `'rt'` or `'xt'`)

Important Python file methods or attributes



Method	Description
`read([size])`	Return data from file as a string, with optional `size` argument indicating the number of bytes to read
`readlines([size])`	Return list of lines in the file, with optional `size` argument
`write(str)`	Write passed string to file



Method	Description
`writelines(strings)`	Write passed sequence of strings to the file
`close()`	Close the handle
`flush()`	Flush the internal I/O buffer to disk
`seek(pos)`	Move to indicated file position (integer)
`tell()`	Return current file position as integer
`closed`	`True` if the file is closed