Python Built-in Data Structures, Functions, and Files

Part 3

Data Structures and Sequences

Part 3

Built-in Sequence Functions

- Python has a handful of useful sequence functions that you should familiarize yourself with and use at any opportunity.

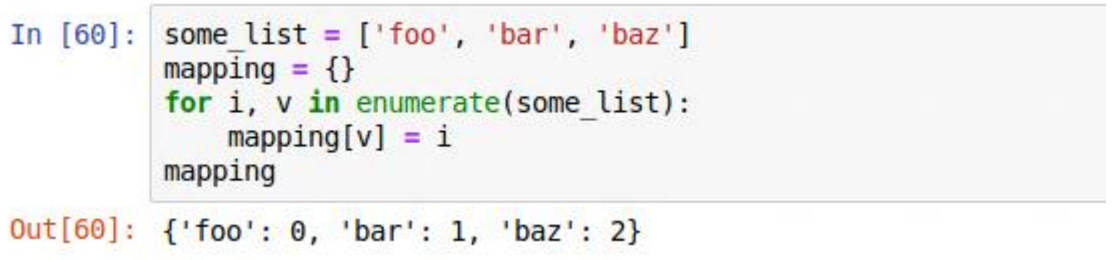
enumerate

- It’s common when iterating over a sequence to want to keep track of the index of the current item.
- A do-it-yourself approach would look like:

  -
    ```
    i = 0
    for value in collection:
        # do something with value
        i += 1
    ```
- Since this is so common, Python has a built-in function, `enumerate`, which returns a sequence of `(i, value)` tuples:

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

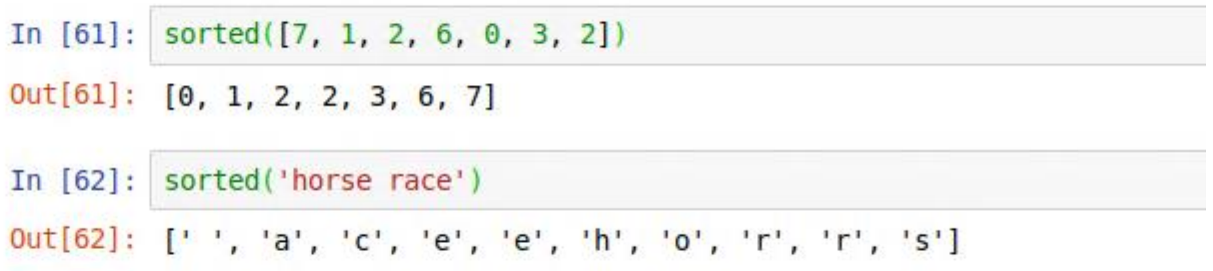
- When you are indexing data, a helpful pattern that uses `enumerate` is computing a `dict` mapping the values of a sequence (which are assumed to be unique) to their locations in the sequence:



```
In [60]: some_list = ['foo', 'bar', 'baz']
         mapping = {}
         for i, v in enumerate(some_list):
             mapping[v] = i
         mapping

Out[60]: {'foo': 0, 'bar': 1, 'baz': 2}
```

sorted

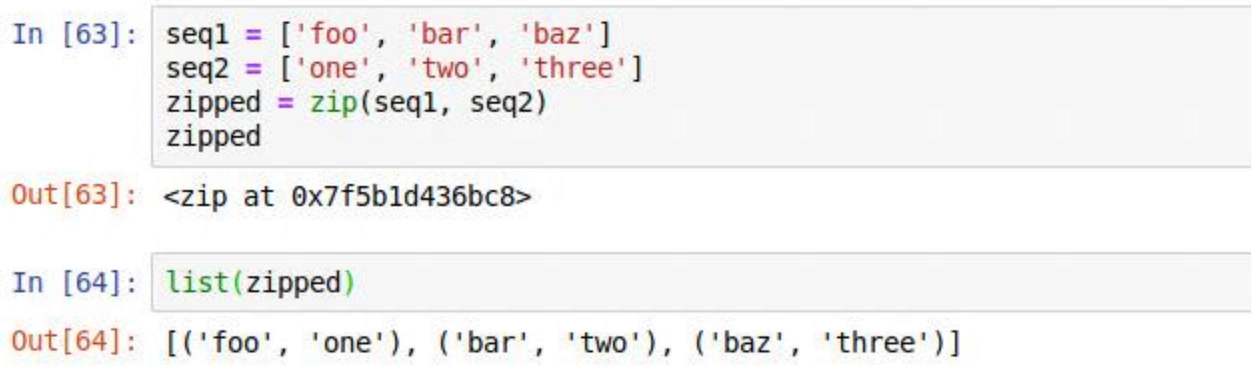
- The `sorted` function returns a new sorted list from the elements of any sequence:



```
In [61]: sorted([7, 1, 2, 6, 0, 3, 2])
Out[61]: [0, 1, 2, 2, 3, 6, 7]

In [62]: sorted('horse race')
Out[62]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

zip

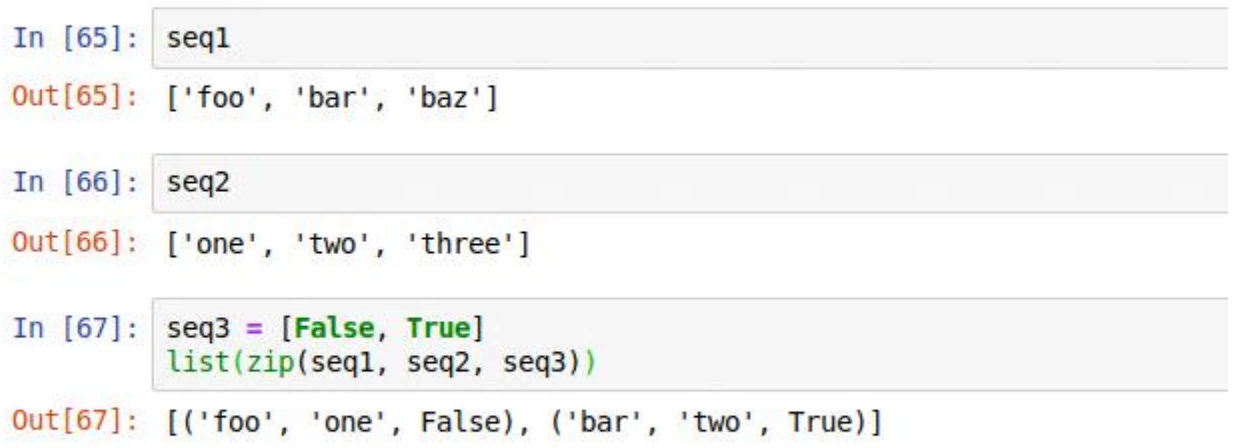
- `zip` “pairs” up the elements of a number of lists, tuples, or other sequences to create a list of tuples:



```
In [63]: seq1 = ['foo', 'bar', 'baz']
         seq2 = ['one', 'two', 'three']
         zipped = zip(seq1, seq2)
         zipped

Out[63]: <zip at 0x7f5b1d436bc8>

In [64]: list(zipped)

Out[64]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

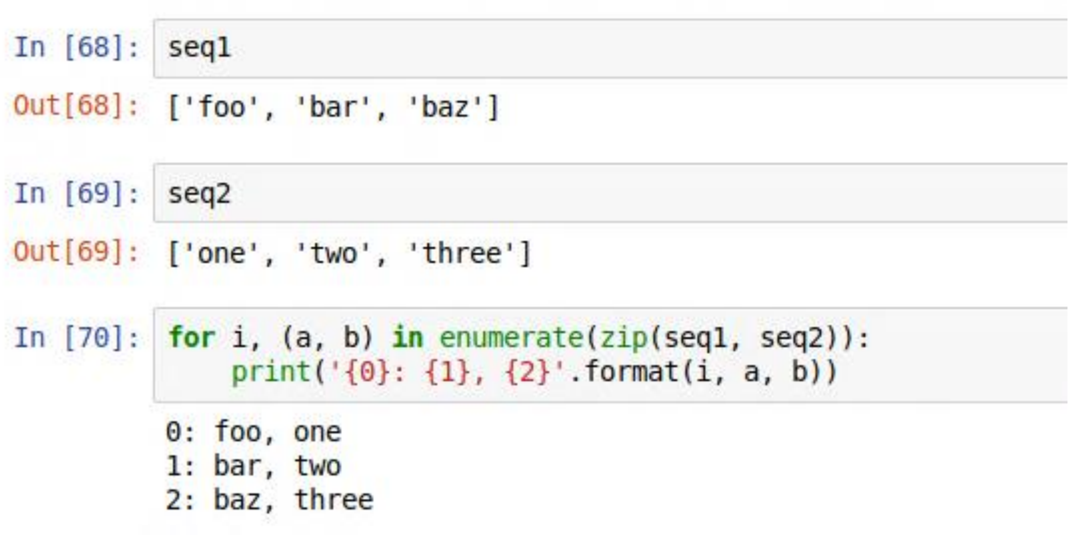
- `zip` returns an iterator of tuples.

- `zip` can take an arbitrary number of sequences, and the number of elements it produces is determined by the shortest sequence:



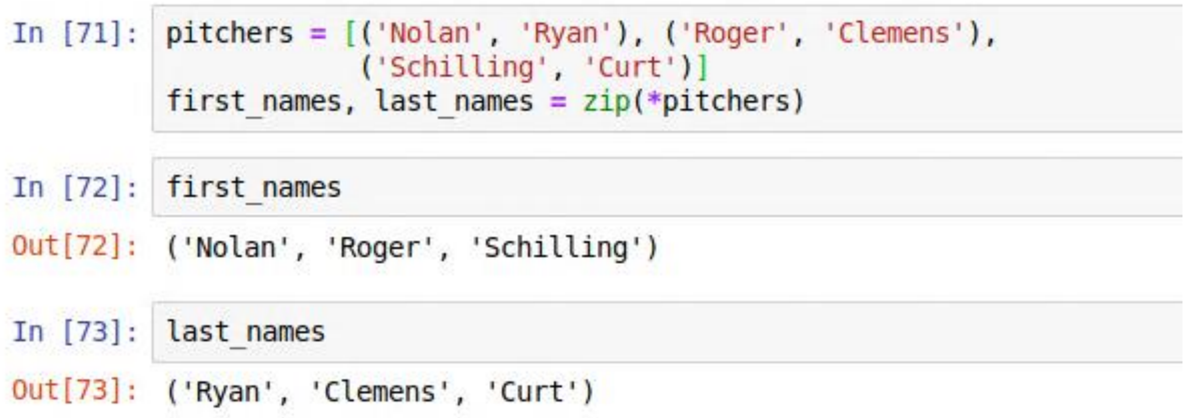
```
In [65]: seq1
Out[65]: ['foo', 'bar', 'baz']

In [66]: seq2
Out[66]: ['one', 'two', 'three']

In [67]: seq3 = [False, True]
         list(zip(seq1, seq2, seq3))
Out[67]: [('foo', 'one', False), ('bar', 'two', True)]
```

- A very common use of `zip` is simultaneously iterating over multiple sequences, possibly also combined with `enumerate`:



```
In [68]: seq1
Out[68]: ['foo', 'bar', 'baz']

In [69]: seq2
Out[69]: ['one', 'two', 'three']

In [70]: 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
```

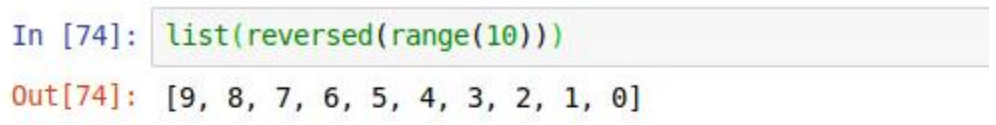
- Given a "zipped" sequence, `zip` can be applied in a clever way to "unzip" the sequence.
- Another way to think about this is converting a list of *rows* into a list of `columns`.
- The syntax, which looks a bit magical, is:



```
In [71]: pitchers = [('Nolan', 'Ryan'), ('Roger', 'Clemens'),
                     ('Schilling', 'Curt')]
         first_names, last_names = zip(*pitchers)

In [72]: first_names

Out[72]: ('Nolan', 'Roger', 'Schilling')

In [73]: last_names

Out[73]: ('Ryan', 'Clemens', 'Curt')
```

reversed

- `reversed` iterates over the elements of a sequence in reverse order:

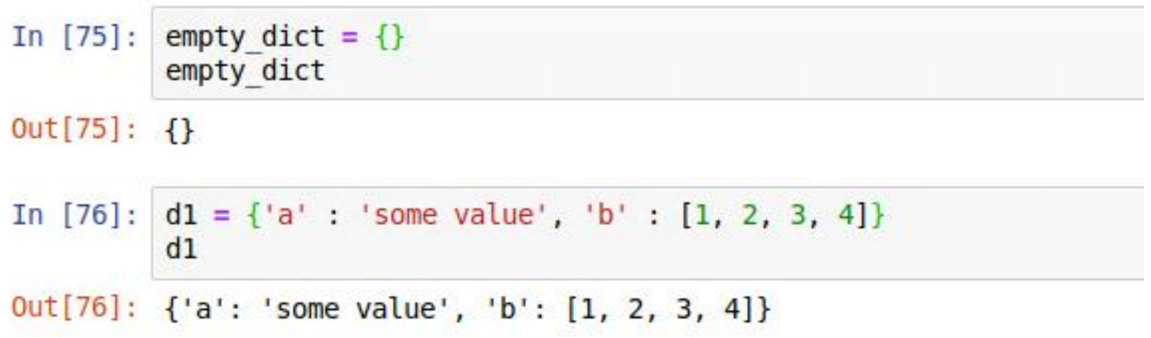


```
In [74]: list(reversed(range(10)))
Out[74]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

- Keep in mind that `reversed` is a generator, so it does not create the reversed sequence until materialized (e.g., with `list` or a `for` loop).

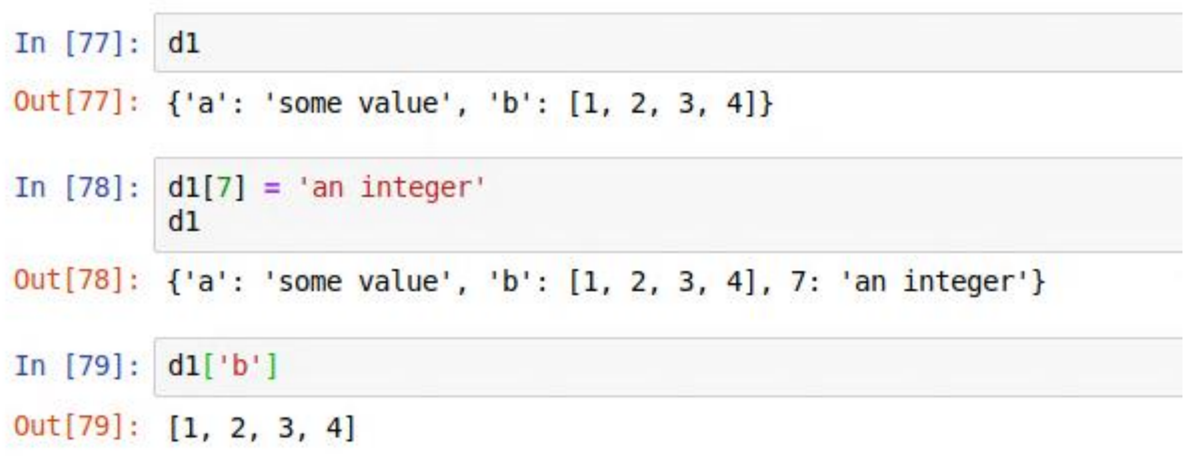
dict

- `dict` is likely the most important built-in Python data structure.
- A more common name for it is *hash map* or *associative array*.
- It is a flexibly sized collection of *key-value* pairs, where *key* and *value* are Python objects.
- One approach for creating one is to use curly braces `{ }` and colons to separate keys and values:



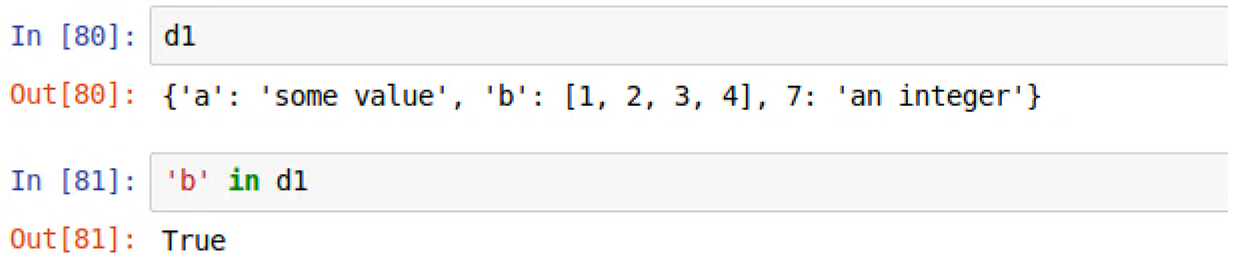
```
In [75]: empty_dict = {}
         empty_dict

Out[75]: {}

In [76]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}
         d1

Out[76]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

- You can access, insert, or set elements using the same syntax as for accessing elements of a list or tuple:



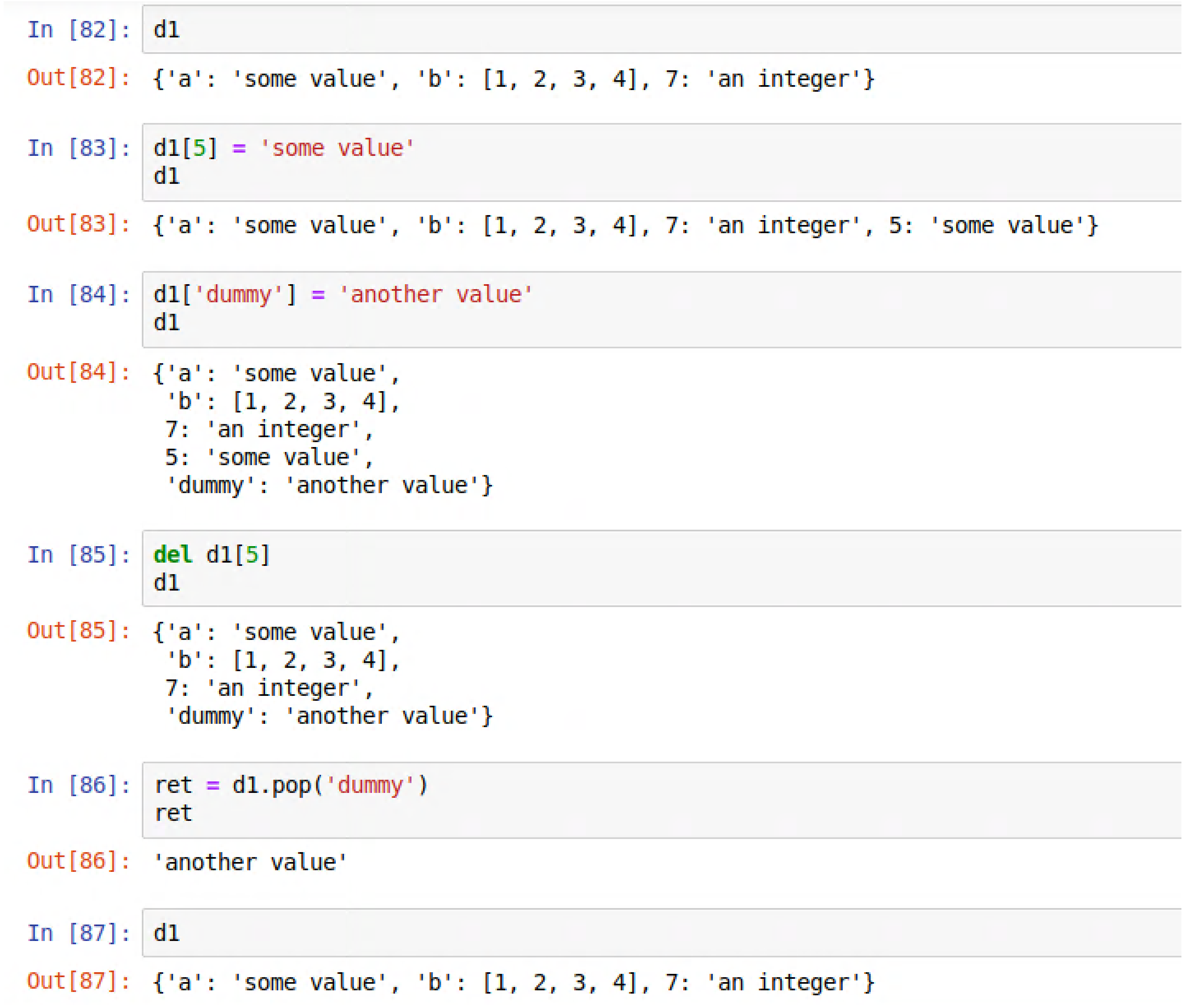
```
In [77]: d1

Out[77]: {'a': 'some value', 'b': [1, 2, 3, 4]}

In [78]: d1[7] = 'an integer'
         d1

Out[78]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}

In [79]: d1['b']

Out[79]: [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:



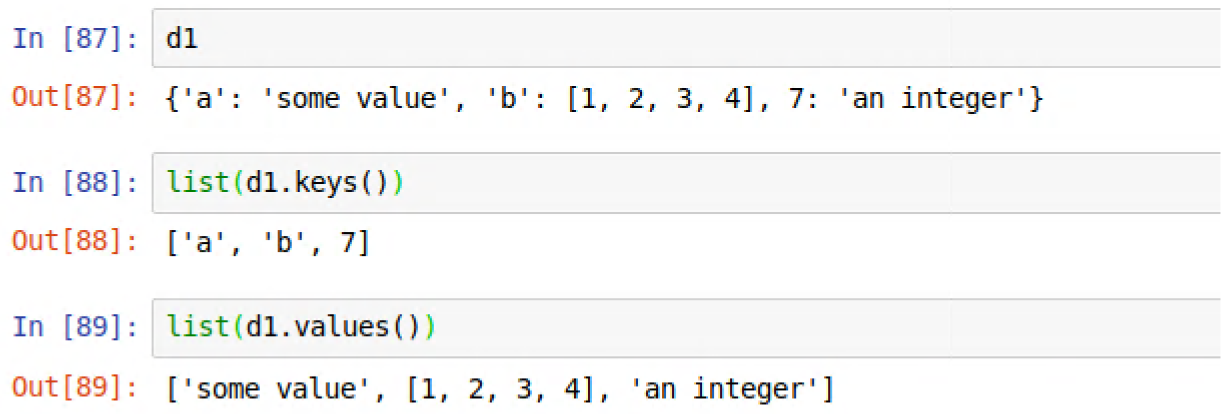
```
In [80]: d1
Out[80]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}

In [81]: 'b' in d1
Out[81]: True
```

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



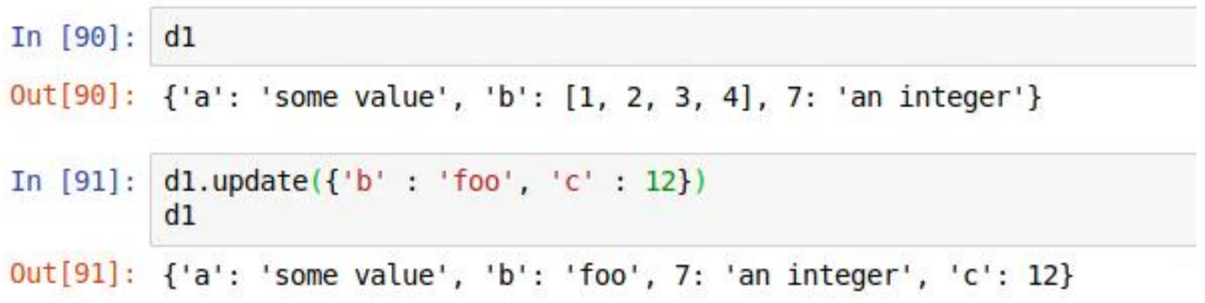
```
In [82]: d1

Out[82]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}

In [83]: d1[5] = 'some value'
         d1

Out[83]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer', 5: 'some value'}

In [84]: d1['dummy'] = 'another value'
         d1

Out[84]: {'a': 'some value',
          'b': [1, 2, 3, 4],
          7: 'an integer',
          5: 'some value',
          'dummy': 'another value'}

In [85]: del d1[5]
         d1

Out[85]: {'a': 'some value',
          'b': [1, 2, 3, 4],
          7: 'an integer',
          'dummy': 'another value'}

In [86]: ret = d1.pop('dummy')
         ret

Out[86]: 'another value'

In [87]: d1

Out[87]: {'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 output the keys and values in the same order:



```
In [87]: d1
Out[87]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}

In [88]: list(d1.keys())
Out[88]: ['a', 'b', 7]

In [89]: list(d1.values())
Out[89]: ['some value', [1, 2, 3, 4], 'an integer']
```

- You can merge one dict into another using the `update` method:



```
In [90]: d1
Out[90]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}

In [91]: d1.update({'b' : 'foo', 'c' : 12})
         d1
Out[91]: {'a': 'some value', 'b': 'foo', 7: 'an integer', 'c': 12}
```

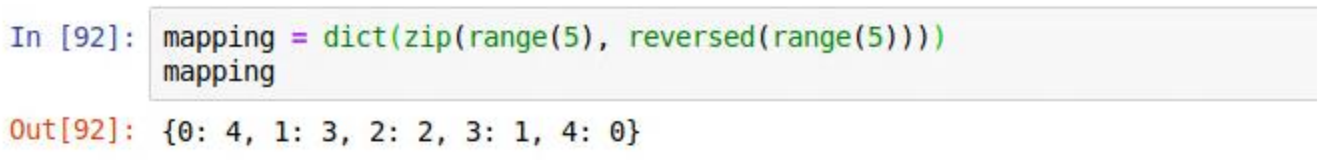
- The `update` method changes dicts in-place, so any existing keys in the data passed to `update` will have their old values discarded.

Creating dicts from sequences

- It’s common to occasionally end up with two sequences that you want to pair up element-wise in a dict.
- As a first cut, you might write code like this:

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

- Since a dict is essentially a collection of 2-tuples, the `dict` function accepts a list of 2-tuples:



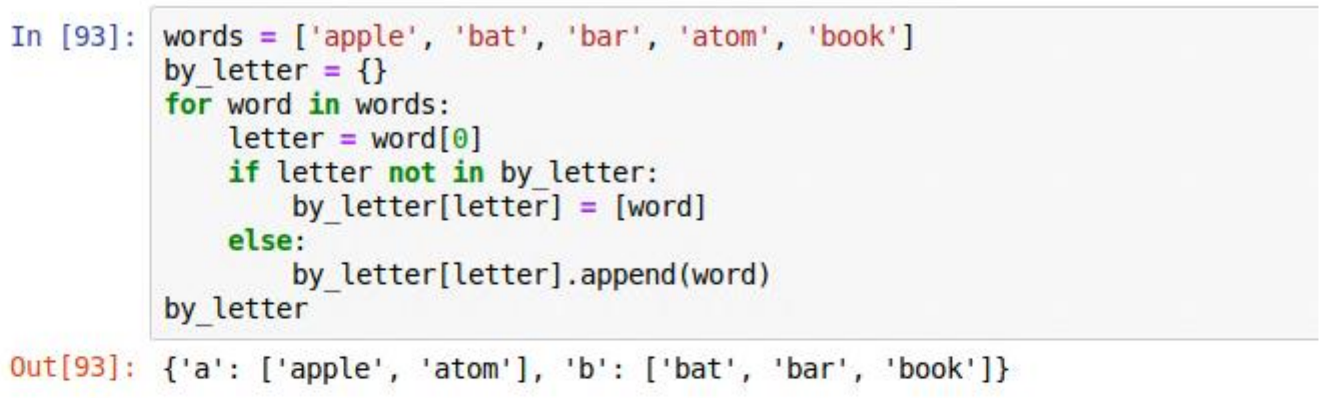
```
In [92]: mapping = dict(zip(range(5), reversed(range(5))))
         mapping

Out[92]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

Default values

- It’s very common to have logic like:

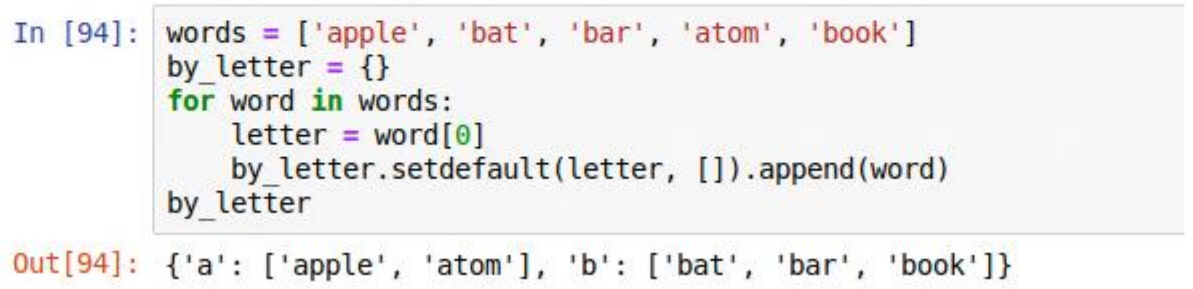
  -
    ```
    if key in some_dict:
        value = some_dict[key]
    else:
        value = default_value
    ```
- Thus, the `dict` methods `get` and `pop` can take a default value to be returned, so that the above `if-else` block can be written simply as:
  - `value = some_dict.get(key, default_value)`
- `get` by default will return `None` if the key is not present, while `pop` will raise an exception.

- With *setting* values, a common case is for the values in a dict to be other collections, like lists.
- For example, you could imagine categorizing a list of words by their first letters as a dict of lists:



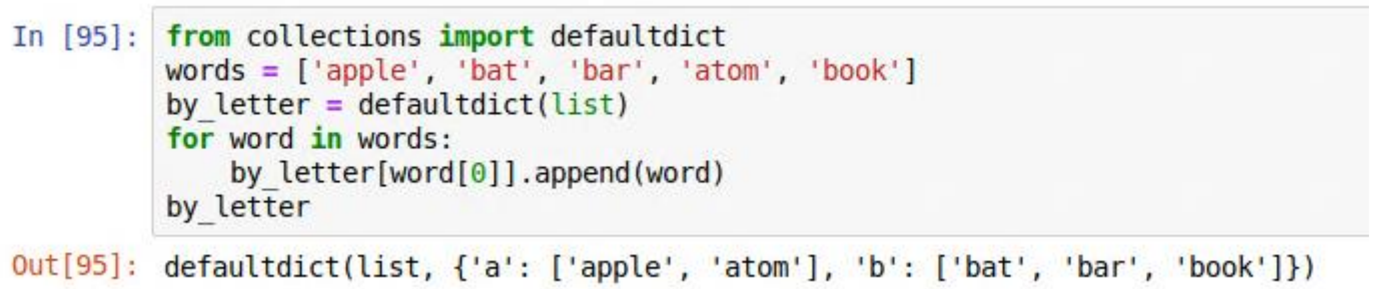
```
In [93]: words = ['apple', 'bat', 'bar', 'atom', 'book']
         by_letter = {}
         for word in words:
             letter = word[0]
             if letter not in by_letter:
                 by_letter[letter] = [word]
             else:
                 by_letter[letter].append(word)
         by_letter

Out[93]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

- The `setdefault` dict method is for precisely this purpose.
- The preceding code can be rewritten as:



```
In [94]: words = ['apple', 'bat', 'bar', 'atom', 'book']
         by_letter = {}
         for word in words:
             letter = word[0]
             by_letter.setdefault(letter, []).append(word)
         by_letter

Out[94]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

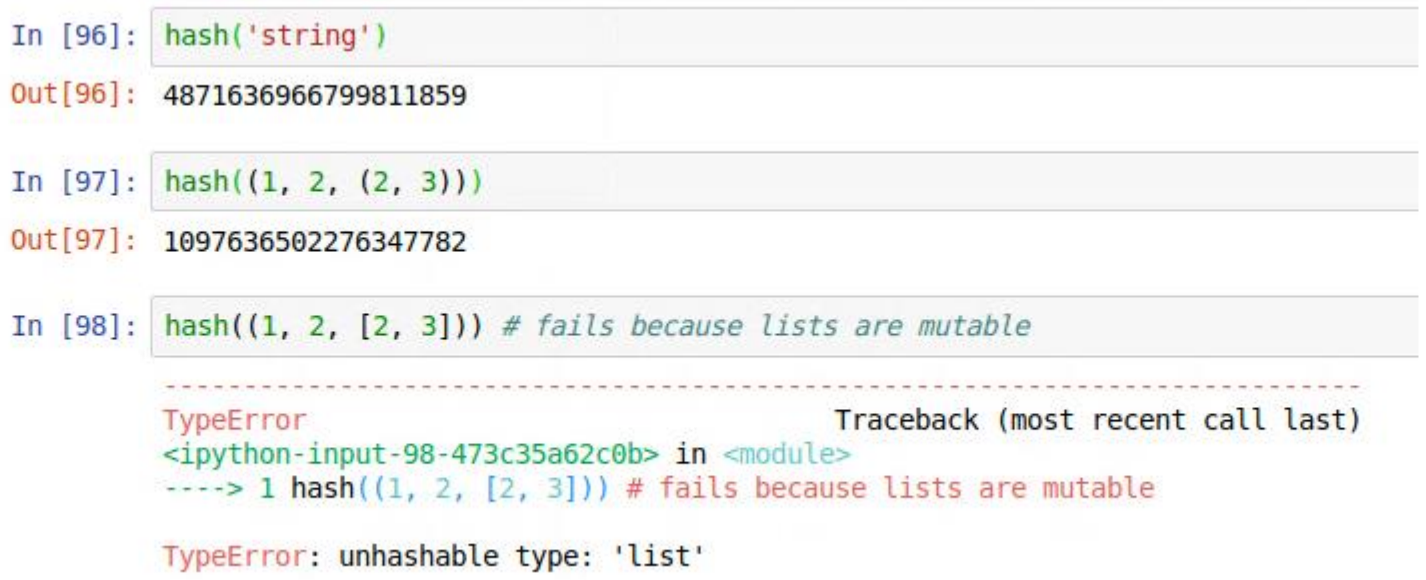
- The built-in `collections` module has a useful class, `defaultdict`, which makes this even easier.
- To create one, you pass a type or function for generating the default value for each slot in the dict:



```
In [95]: from collections import defaultdict
         words = ['apple', 'bat', 'bar', 'atom', 'book']
         by_letter = defaultdict(list)
         for word in words:
             by_letter[word[0]].append(word)
         by_letter

Out[95]: defaultdict(list, {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']})
```

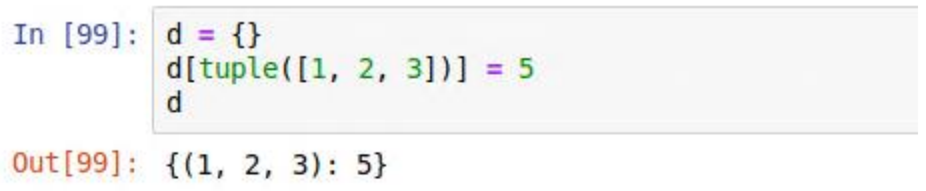
Valid dict key types

- While the values of a dict can be any Python object, the keys generally have to be immutable objects like scalar types (int, float, string) or tuples (all the objects in the tuple need to be immutable, too).
- The technical term here is *hashability*.
- You can check whether an object is hashable (can be used as a key in a dict) with the `hash` function:



```
In [96]: hash('string')
Out[96]: 4871636966799811859

In [97]: hash((1, 2, (2, 3)))
Out[97]: 1097636502276347782

In [98]: hash((1, 2, [2, 3])) # fails because lists are mutable
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-98-473c35a62c0b> in <module>
----> 1 hash((1, 2, [2, 3])) # fails because lists are mutable

TypeError: unhashable type: 'list'
```

- To use a list as a key, one option is to convert it to a tuple, which can be hashed as long as its elements also can:



```
In [99]: d = {}
         d[tuple([1, 2, 3])] = 5
         d

Out[99]: {(1, 2, 3): 5}
```