Python Built-in Data Structures, Functions, and Files

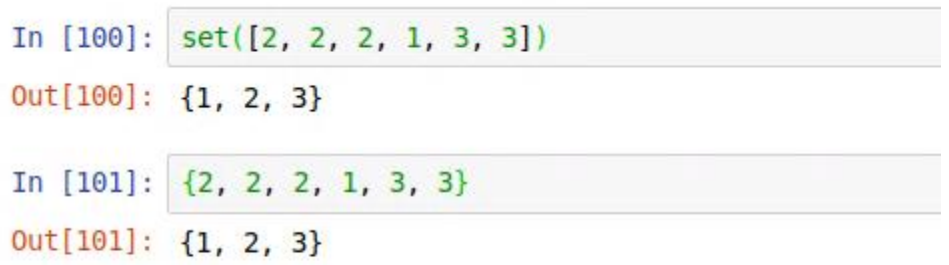
Part 4

Data Structures and Sequences

Part 4

set

- A set is an unordered collection of unique elements.
- You can think of them like dicts, but keys only, no values.
- A set can be created in two ways: via the `set` function or via a *set literal* with curly braces:

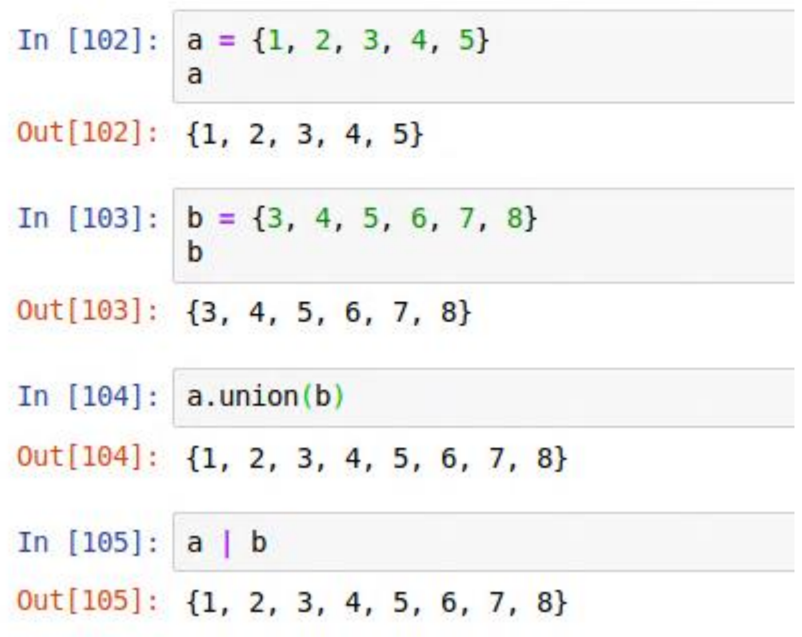


```
In [100]: set([2, 2, 2, 1, 3, 3])
Out[100]: {1, 2, 3}

In [101]: {2, 2, 2, 1, 3, 3}
Out[101]: {1, 2, 3}
```

- Sets support mathematical set operations like union, intersection, difference, and symmetric difference.

- The union of these two sets is the set of distinct elements occurring in either set.
- This can be computed with either the `union` method or the `|` binary operator:



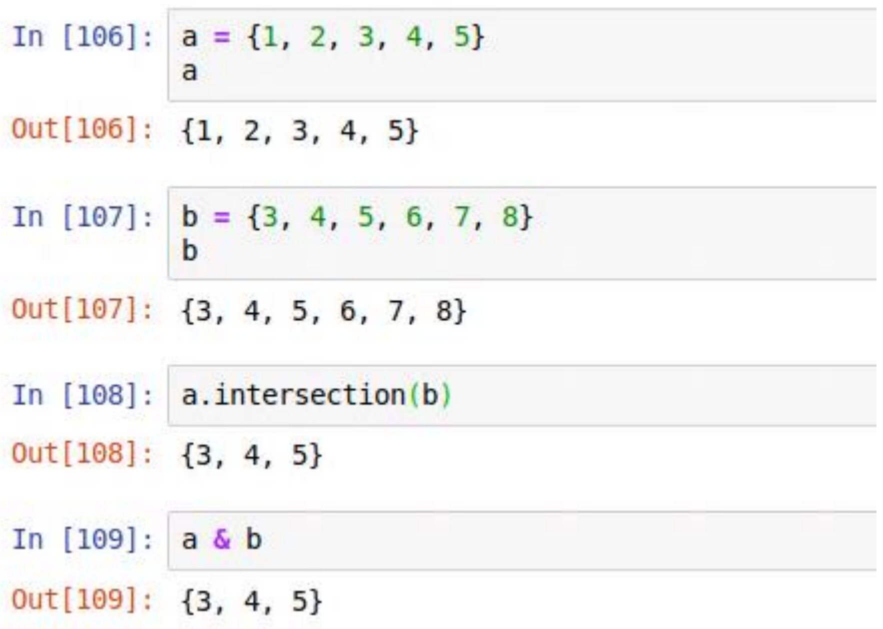
```
In [102]: a = {1, 2, 3, 4, 5}
          a
Out[102]: {1, 2, 3, 4, 5}

In [103]: b = {3, 4, 5, 6, 7, 8}
          b
Out[103]: {3, 4, 5, 6, 7, 8}

In [104]: a.union(b)
Out[104]: {1, 2, 3, 4, 5, 6, 7, 8}

In [105]: a | b
Out[105]: {1, 2, 3, 4, 5, 6, 7, 8}
```

- The intersection contains the elements occurring in both sets.
- The `&` operator or the `intersection` method can be used:



```
In [106]: a = {1, 2, 3, 4, 5}
          a

Out[106]: {1, 2, 3, 4, 5}

In [107]: b = {3, 4, 5, 6, 7, 8}
          b

Out[107]: {3, 4, 5, 6, 7, 8}

In [108]: a.intersection(b)

Out[108]: {3, 4, 5}

In [109]: a & b

Out[109]: {3, 4, 5}
```

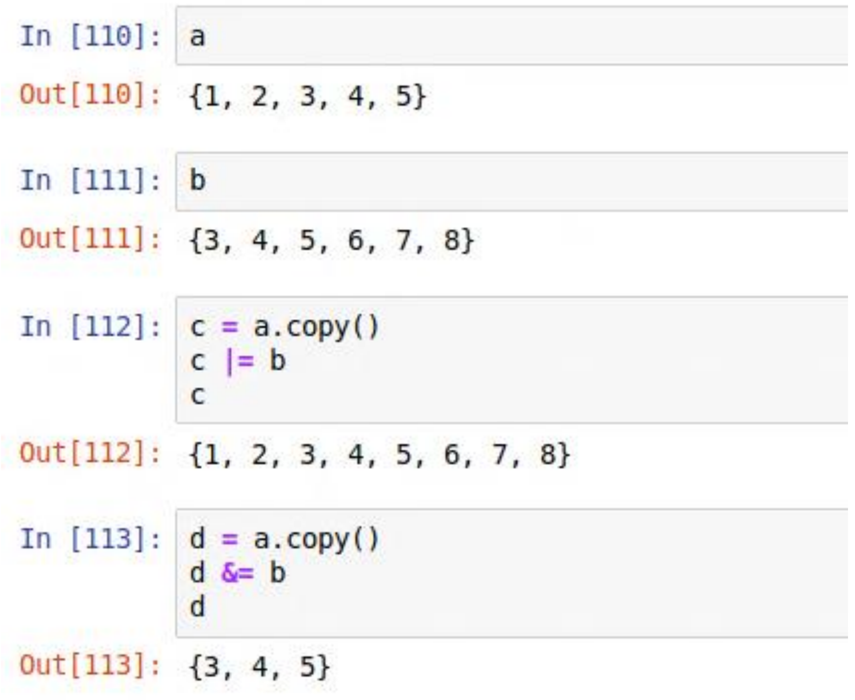
Function	Alternative syntax	Description
`a.add(x)`	N/A	Add element `x` to the set `a`
`a.clear()`	N/A	Reset the set `a` to an empty state, discarding all of its elements
`a.remove(x)`	N/A	Remove element `x` from the set `a`
`a.pop()`	N/A	Remove an arbitrary element from the set `a`, raising `KeyError` 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

- All of the logical set operations have in-place counterparts, which enable you to replace the contents of the set on the left side of the operation with the result.
- For very large sets, this may be more efficient:



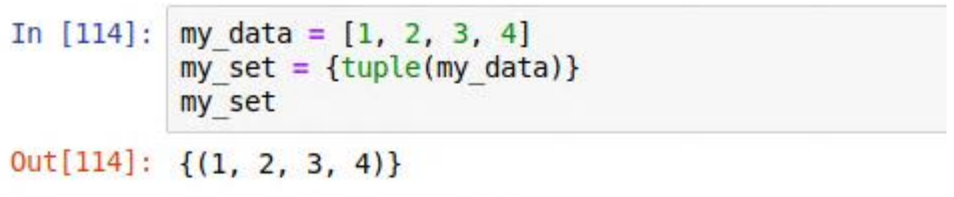
```
In [110]: a
Out[110]: {1, 2, 3, 4, 5}

In [111]: b
Out[111]: {3, 4, 5, 6, 7, 8}

In [112]: c = a.copy()
          c |= b
          c
Out[112]: {1, 2, 3, 4, 5, 6, 7, 8}

In [113]: d = a.copy()
          d &= b
          d
Out[113]: {3, 4, 5}
```

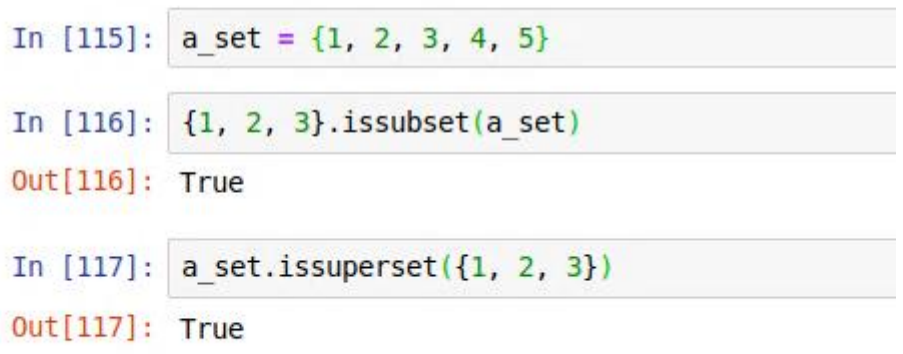
- Like dicts, set elements generally must be immutable.
- To have list-like elements, you must convert it to a tuple:



```
In [114]: my_data = [1, 2, 3, 4]
          my_set = {tuple(my_data)}
          my_set

Out[114]: {(1, 2, 3, 4)}
```

- You can also check if a set is a subset of (is contained in) or a superset of (contains all elements of) another set:

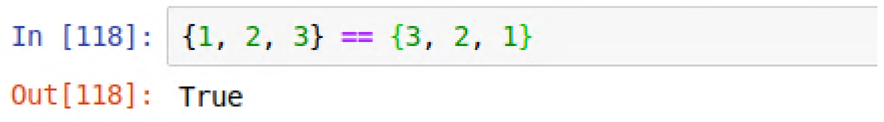


```
In [115]: a_set = {1, 2, 3, 4, 5}

In [116]: {1, 2, 3}.issubset(a_set)
Out[116]: True

In [117]: a_set.issuperset({1, 2, 3})
Out[117]: True
```

- Sets are equal if and only if their contents are equal:



```
In [118]: {1, 2, 3} == {3, 2, 1}
Out[118]: True
```

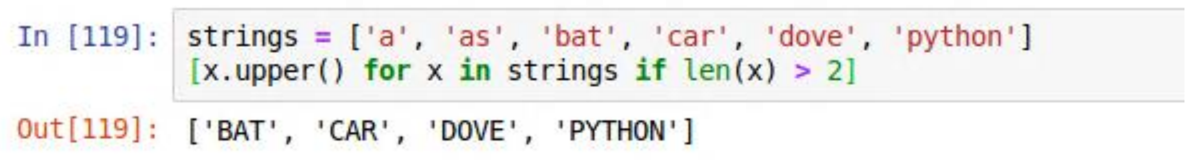
List, Set, and Dict Comprehensions

- 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.
- They take the basic form:

  -
    ```
    [expr for val in collection if condition]
    ```
- This is equivalent to the following for loop:

  -
    ```
    result = []
    for val in collection:
        if condition:
            result.append(expr)
    ```
- The filter condition can be omitted, leaving only the expression.

- For example, given a list of strings, we could filter out strings with length 2 or less and also convert them to uppercase like this:

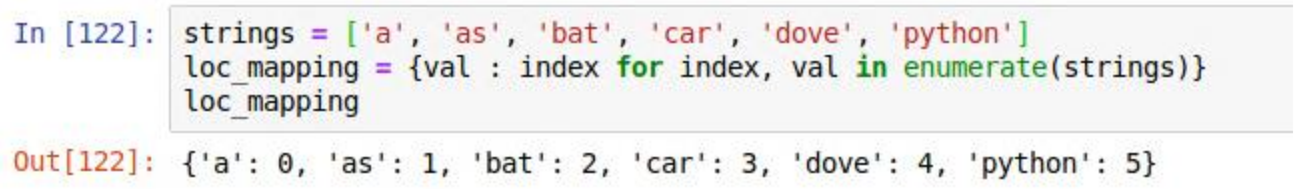


```
In [119]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
          [x.upper() for x in strings if len(x) > 2]

Out[119]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

- Set and dict comprehensions are a natural extension, producing sets and dicts in an idiomatically similar way instead of lists.
- A dict comprehension looks like this:
  - `dict_comp = {`***`key-expr : value-expr`***` for value in collection`
    `if `***`condition`***`}`
- A set comprehension looks like the equivalent list comprehension except with curly braces instead of square brackets:
  - `set_comp = {`***`expr`***` for value in collection if `***`condition`***`}`

- As a simple dict comprehension example, we could create a lookup map of these strings to their locations in the list:

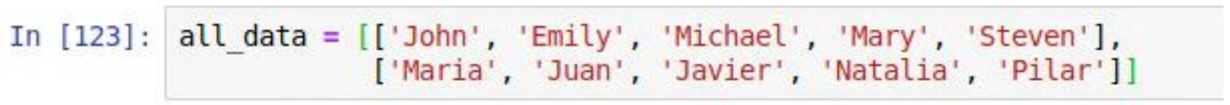


```
In [122]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
          loc_mapping = {val : index for index, val in enumerate(strings)}
          loc_mapping

Out[122]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

Nested list comprehensions

- Suppose we have a list of lists containing some English and Spanish names:

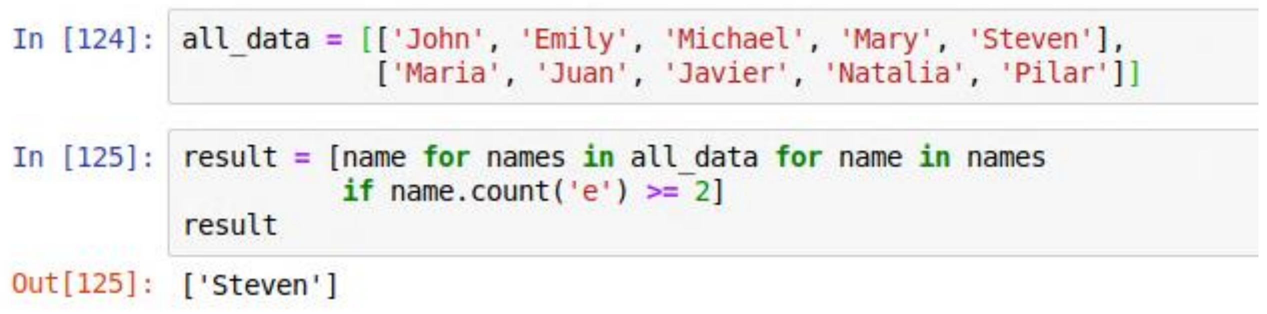


```
In [123]: all_data = [['John', 'Emily', 'Michael', 'Mary', 'Steven'],
                      ['Maria', 'Juan', 'Javier', 'Natalia', 'Pilar']]
```

- You might have gotten these names from a couple of files and decided to organize them by language.
- Now, suppose we wanted to get a single list containing all names with two or more `e`’s in them.
- We could certainly do this with a simple `for` loop:

  -
    ```
    names_of_interest = []
    for names in all_data:
        enough_es = [name for name in names if name.count('e') >= 2]
        names_of_interest.extend(enough_es)
    ```

- You can actually wrap this whole operation up in a single nested list comprehension, which will look like:

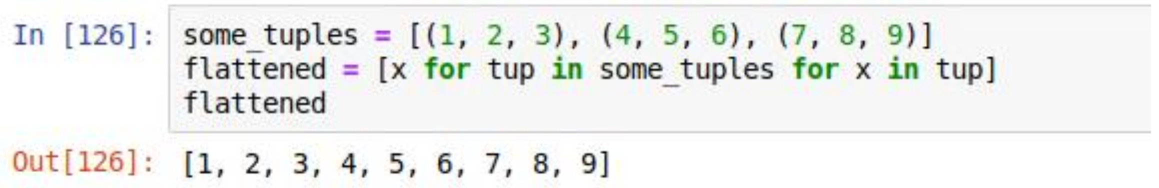


```
In [124]: all_data = [['John', 'Emily', 'Michael', 'Mary', 'Steven'],
                      ['Maria', 'Juan', 'Javier', 'Natalia', 'Pilar']]

In [125]: result = [name for names in all_data for name in names
                    if name.count('e') >= 2]
          result

Out[125]: ['Steven']
```

- Here is another example where we “flatten” a list of tuples of integers into a simple list of integers:

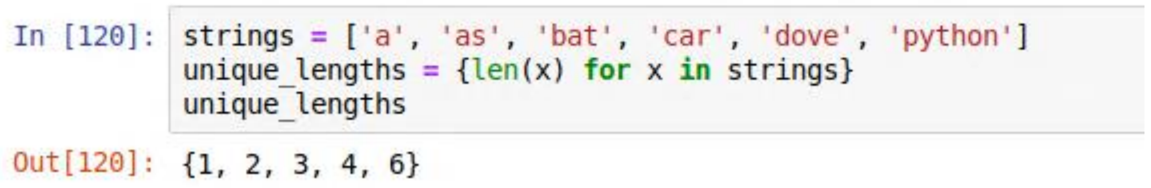


```
In [126]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
          flattened = [x for tup in some_tuples for x in tup]
          flattened

Out[126]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

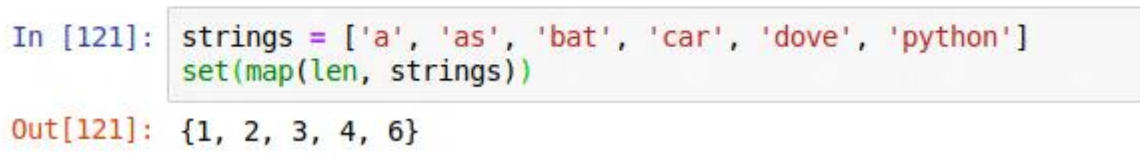
- You can have arbitrarily many levels of nesting, though if you have more than two or three levels of nesting you should probably start to question whether this makes sense from a code readability standpoint.

- Like list comprehensions, set and dict comprehensions are mostly conveniences, but they similarly can make code both easier to write and read.
- Consider the list of strings from before.
- Suppose we wanted a set containing just the lengths of the strings contained in the collection; we could easily compute this using a set comprehension:



```
In [120]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
          unique_lengths = {len(x) for x in strings}
          unique_lengths
Out[120]: {1, 2, 3, 4, 6}
```

- We could also express this more functionally using the `map` function:



```
In [121]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
          set(map(len, strings))

Out[121]: {1, 2, 3, 4, 6}
```