Lecture 4.2 : Sets

Introduction

- A *set* is a *collection* of objects of arbitrary type. The objects in a set are called its *members*. There is no order to the members of a set (like a dictionary). Sets are iterable. A key property of a set is that it may contain only one copy of a particular object: duplicates are not allowed. A set with no elements is *the empty set*.

Python sets

- A set is created by calling the `set()` constructor or by using curly brackets. (Note a `dictionary` is also created using curly brackets but while a dictionary consists of *key-value* pairs separated by a colon, the members of a `set` are separated by commas.)
- The `set()` constructor requires an iterable be passed to it. Each object in the iterable becomes a member of the set. No duplicates are allowed so a given object will only be added once. For example:

```
>>> empty_set = set()
>>> type(empty_set)
<class 'set'>
>>> my_set = set('aeiou')
>>> my_set
{'a', 'i', 'e', 'u', 'o'}
>>> another_set = set([1,2,3,1,2,7])
>>> another_set # Note how duplicates are removed
{1, 2, 3, 7}
>>> new_set = set('Some characters')
>>> new_set # Note how duplicates are removed
{'h', 'm', 'o', 'a', ' ', 'c', 'e', 'S', 'r', 's', 't'}
>>> len('Some characters')
15
>>> len(new_set)
11
```

- Note how duplicates in the original iterable are absent in the corresponding `set`.

Set operators and functions

- The number of members in a `set` can be determined with the `len()` function.
- We can use the `in` operator to check for membership of a `set`.
- We can use a `for` loop to iterate over the members of a `set`. Since a `set` is unordered, the order in which members are visited by a `for` loop is unknown.

```
>>> my_set = set(['fred', 'joe', 23, 'mary'])
>>> len(my_set)
4
```

```
>>> 'joe' in my_set
True
>>> 'vicky' in my_set
False
>>> for m in my_set:
...     print(m)
...
joe
mary
fred
23
```

Set methods

- We can find the *intersection* of two sets using the `intersection()` method. The intersection of sets A and B is the set of elements that are in both A and B. For example:

```
>>> a_set = set('abcd')
>>> b_set = set('cdef')
>>> a_set
{'a', 'c', 'b', 'd'}
>>> b_set
{'c', 'e', 'd', 'f'}
>>> a_set.intersection(b_set)
{'c', 'd'}
>>> b_set.intersection(a_set)
{'c', 'd'}
```

- We can find the *union* of two sets using the `union()` method. The union of sets A and B is the set of elements that are in A or B. For example:

```
>>> a_set
{'a', 'c', 'b', 'd'}
>>> b_set
{'c', 'e', 'd', 'f'}
>>> a_set.union(b_set)
{'a', 'c', 'b', 'e', 'd', 'f'}
>>> b_set.union(a_set)
{'a', 'c', 'b', 'e', 'd', 'f'}
```

- We can find the *set difference* between two sets using the `difference()` method. A set difference B is the set of elements in A but not in B. B set difference A is the set of elements in B but not in A. For example:

```
>>> a_set
{'a', 'c', 'b', 'd'}
>>> b_set
{'c', 'e', 'd', 'f'}
>>> a_set.difference(b_set)
{'a', 'b'}
>>> b_set.difference(a_set)
{'e', 'f'}
>>> b_set.difference(b_set)
set()
```

- We can check whether one set is a *subset* of another using the `issubset()` method. Set A is a subset of set B if every member of A is also a member of B. We can check whether one set is a

superset of another using the `issuperset()` method. Set A is a superset of set B if every member of B is also a member of A. For example:

```
>>> a_set
{'a', 'c', 'b', 'd'}
>>> b_set
{'c', 'e', 'd', 'f'}
>>> a_set.issubset(b_set)
False
>>> b_set.issuperset(a_set)
False
>>> a_set.issuperset(b_set)
False
>>> c_set = set('ad')
>>> c_set
{'a', 'd'}
>>> c_set.issubset(a_set)
True
>>> c_set.issuperset(a_set)
False
>>> a_set.issuperset(c_set)
True
```

Set applications

- One common set application is collecting unique elements of a file, list, etc. Simply add each element of the file, list, etc. to a set and let the set look after the removal of duplicates.
- Intersection can be used to find objects that share certain properties i.e. they are members of two distinct sets.