SETS

Concept: A set contains a collection of unique values and works like a mathematical set. A set is an object that stores a collection of data in the same way as mathematical sets.

Here are some important things to know about sets:
• All the elements in a set must be unique. No two elements can have the same value.
• Sets are unordered, which means that the elements in a set are not stored in any particular order.
• The elements that are stored in a set can be of different data types.

CREATING A SET

To create a set, you have to call the built-in set function.
Here is an example of how you create an empty set:

```
myset = set()
```

After this statement executes, the *myset* variable will reference an empty set. You can also pass one argument to the set function. The argument that you pass must be an object that contains iterable elements, such as a list, a tuple, or a string. The individual elements of the object that you pass as an argument become elements of the set.
Here is an example:

```
myset = set(['a', 'b', 'c'])
```

In this example we are passing a list as an argument to the set function. After this statement executes, the myset variable references a set containing the elements 'a', 'b', and 'c'.

If you pass a string as an argument to the set function, each individual character in the string becomes a member of the set. Here is an example:

```
myset = set('abc')
```

After this statement executes, the myset variable will reference a set containing the elements 'a', 'b', and 'c'.

Sets cannot contain duplicate elements. If you pass an argument containing duplicate elements to the set function, only one of the duplicated elements will appear in the set.
Here is an example:

```
myset = set('aaabc')
```

The character 'a' appears multiple times in the string, but it will appear only once in the set. After this statement executes, the myset variable will reference a set containing the elements 'a', 'b', and 'c'.

What if you want to create a set in which each element is a string containing more than one character?

For example, how would you create a set containing the elements 'one', 'two', and 'three'? The following code does not accomplish the task because you can pass no more than one argument to the set function:

```
# This is an ERROR!
myset = set('one', 'two', 'three')
```

The following does not accomplish the task either:

```
# This does not do what we intend.
myset = set('one two three')
```

After this statement executes, the myset variable will reference a set containing the elements 'o', 'n', 'e', ' ', 't', 'w', 'h', and 'r'. To create the set that we want, we have to pass a list containing the strings ‘one’, ‘two’, and ‘three’ as an argument to the set function.
Here is an example:

```
# OK, this works.
myset = set(['one', 'two', 'three'])
```

After this statement executes, the myset variable will reference a set containing the elements 'one', 'two', and 'three'.

GETTING THE NUMBER OF ELEMENTS IN A SET

As with lists, tuples, and dictionaries, you can use the len function to get the number of elements in a set. The following interactive session demonstrates:

```
 >>> myset = set([1, 2, 3, 4, 5])
>>> len(myset)
5
>>>
```

ADDING AND REMOVING ELEMENTS

Sets are mutable objects, so you can add items to them and remove items from them. You use the add method to add an element to a set. The following interactive session demonstrates:

```
>>> myset = set()
>>> myset.add(1)
>>> myset.add(2)
>>> myset.add(3)
>>> myset
{1, 2, 3}
>>> myset.add(2)
>>> myset
{1, 2, 3}
```

The statement in line 1 creates an empty set and assigns it to the myset variable.
The statements in lines 2 through 4 add the values 1, 2, and 3 to the set.
Line 5 displays the contents of the set, which is shown in line 6.
The statement in line 7 attempts to add the value 2 to the set.
The value 2 is already in the set, however. If you try to add a duplicate item to a set with the add method, the method does not raise an exception. It simply does not add the item.

You can add a group of elements to a set all at one time with the update method. When you call the update method as an argument, you pass an object that contains iterable elements, such as a list, a tuple, string, or another set. The individual elements of the object that you pass as an argument become elements of the set.

The following interactive session demonstrates:

```
>>> myset = set([1, 2, 3])
>>> myset.update([4, 5, 6])
>>> myset
{1, 2, 3, 4, 5, 6}
>>>
```

The statement in line 1 creates a set containing the values 1, 2, and 3. Line 2 adds the values 4, 5, and 6.
The following session shows another example:

```
>>> set1 = set([1, 2, 3])
>>> set2 = set([8, 9, 10])
>>> set1.update(set2)
>>> set1
{1, 2, 3, 8, 9, 10}
>>> set2
{8, 9, 10}
>>>
```

Line 1 creates a set containing the values 1, 2, and 3 and assigns it to the set1 variable.
Line 2 creates a set containing the values 8, 9, and 10 and assigns it to the set2 variable.
Line 3 calls the set1.update method, passing set2 as an argument.
This causes the element of set2 to be added to set1. Notice that set2 remains unchanged.
The following session shows another example:

```
>>> myset = set([1, 2, 3])
>>> myset.update('abc')
>>> myset
{'a', 1, 2, 3, 'c', 'b'}
>>>
```

The statement in line 1 creates a set containing the values 1, 2, and 3.

Line 2 calls the myset.update method, passing the string 'abc' as an argument. This causes the each character of the string to be added as an element to myset.

You can remove an item from a set with either the remove method or the discard method.

You pass the item that you want to remove as an argument to either method, and that item is removed from the set.

The only difference between the two methods is how they behave when the specified item is not found in the set.

The remove method raises a KeyError exception, but the discard method does not raise an exception.

The following interactive session demonstrates:

```
>>> myset = set([1, 2, 3, 4, 5])
>>> myset
{1, 2, 3, 4, 5}
>>> myset.remove(1)
>>> myset
{2, 3, 4, 5}
>>> myset.discard(5)
>>> myset
{2, 3, 4}
>>> myset.discard(99)
>>> myset.remove(99)
Traceback (most recent call last):
File "<pyshell#12>", line 1, in <module>
myset.remove(99)
KeyError: 99
>>>
```

Line 1 creates a set with the elements 1, 2, 3, 4, and 5.
Line 2 displays the contents of the set, which is shown in line 3.
Line 4 calls the remove method to remove the value 1 from the set.
You can see in the output shown in line 6 that the value 1 is no longer in the set.
Line 7 calls the discard method to remove the value 5 from the set.
You can see in the output in line 9 that the value 5 is no longer in the set.
Line 10 calls the discard method to remove the value 99 from the set. The value is not found in the set, but the discard method does not raise an exception.
Line 11 calls the remove method to remove the value 99 from the set.
Because the value is not in the set, a KeyError exception is raised, as shown in lines 12 through 15.

You can clear all the elements of a set by calling the clear method. The following interactive session demonstrates:

```
>>> myset = set([1, 2, 3, 4, 5])
>>> myset
{1, 2, 3, 4, 5}
>>> myset.clear()
>>> myset
set()
```

The statement in line 4 calls the clear method to clear the set.
Notice in line 6 that when we display the contents of an empty set, the interpreter displays set().

USING THE FOR LOOP TO ITERATE OVER A SET
You can use the for loop in the following general format to iterate over all the elements in a set:

```
for var in set:
        statement
        statement
etc.
```

In the general format, var is the name of a variable and set is the name of a set. This loop iterates once for each element in the set. Each time the loop iterates, var is assigned an element.
The following interactive session demonstrates:

```
>>> myset = set(['a', 'b', 'c'])
>>> for val in myset:
        print(val)
a
c
b
```

Lines 2 through 3 contain a for loop that iterates once for each element of the myset set.
Each time the loop iterates, an element of the set is assigned to the val variable.
Line 3 prints the value of the val variable.
Lines 5 through 7 show the output of the loop.

USING THE IN AND NOT IN OPERATORS TO TEST FOR A VALUE IN A SET
You can use the in operator to determine whether a value exists in a set. The following interactive session demonstrates:

```
>>> myset = set([1, 2, 3])
>>> if 1 in myset:
        print('The value 1 is in the set.')
The value 1 is in the set.
```

The if statement in line 2 determines whether the value 1 is in the myset set.
If it is, the statement in line 3 displays a message.
You can also use the not in operator to determine if a value does not exist in a set, as demonstrated in the following session:

```
>>> myset = set([1, 2, 3])
>>> if 99 not in myset:
        print('The value 99 is not in the set.')
The value 99 is not in the set.
```

FINDING THE UNION OF SETS
The union of two sets is a set that contains all the elements of both sets. In Python, you can call the union method to get the union of two sets. Here is the general format:

set1.union(set2)

In the general format, set1 and set2 are sets. The method returns a set that contains the elements of both set1 and set2.
The following interactive session demonstrates:

```
>>> set1 = set([1, 2, 3, 4])
>>> set2 = set([3, 4, 5, 6])
 >>> set3 = set1.union(set2)
>>> set3
{1, 2, 3, 4, 5, 6}
>>>
```

The statement in line 3 calls the set1 object’s union method, passing set2 as an argument.
The method returns a set that contains all the elements of set1 and set2 (without duplicates, of course). The resulting set is assigned to the set3 variable.

You can also use the | operator to find the union of two sets.
Here is the general format of an expression using the | operator with two sets:

set1 | set2

In the general format, set1 and set2 are sets. The expression returns a set that contains the elements of both set1 and set2.
The following interactive session demonstrates:

```
>>> set1 = set([1, 2, 3, 4])
>>> set2 = set([3, 4, 5, 6])
 >>> set3 = set1 | set2
>>> set3
{1, 2, 3, 4, 5, 6}
>>>
```

FINDING THE INTERSECTION OF SETS

The intersection of two sets is a set that contains only the elements that are found in both sets. In Python, you can call the intersection method to get the intersection of two sets.
Here is the general format:

```
set1.intersection(set2)
```

In the general format, set1 and set2 are sets. The method returns a set that contains the elements that are found in both set1 and set2. The following interactive session demonstrates:

```
>>> set1 = set([1, 2, 3, 4])
>>> set2 = set([3, 4, 5, 6])
 >>> set3 = set1.intersection(set2)
>>> set3
{3, 4}
>>>
```

The statement in line 3 calls the set1 object's intersection method, passing set2 as an argument. The method returns a set that contains the elements that are found in both set1 and set2. The resulting set is assigned to the set3 variable.
You can also use the & operator to find the intersection of two sets. Here is the general format of an expression using the & operator with two sets:

```
set1 & set2
```

In the general format, set1 and set2 are sets. The expression returns a set that contains the elements that are found in both set1 and set2.
The following interactive session demonstrates:

```
>>> set1 = set([1, 2, 3, 4])
>>> set2 = set([3, 4, 5, 6])
>>> set3 = set1 & set2
>>> set3
{3, 4}
>>>
```

FINDING THE DIFFERENCE OF SETS

The difference of set1 and set2 is the elements that appear in set1 but do not appear in set2. In Python, you can call the difference method to get the difference of two sets.
Here is the general format:

```
set1.difference(set2)
```

In the general format, set1 and set2 are sets. The method returns a set that contains the elements that are found in set1 but not in set2. The following interactive session demonstrates:

```
 >>> set1 = set([1, 2, 3, 4])
>>> set2 = set([3, 4, 5, 6])
>>> set3 = set1.difference(set2)
>>> set3
{1, 2}
>>>
```

You can also use the - operator to find the difference of two sets. Here is the general format of an expression using the - operator with two sets:

```
set1 − set2
```

In the general format, set1 and set2 are sets. The expression returns a set that contains the elements that are found in set1 but not in set2. The following interactive session demonstrates:

```
>>> set1 = set([1, 2, 3, 4])
>>> set2 = set([3, 4, 5, 6])
>>> set3 = set1 - set2
>>> set3
{1, 2}
>>>
```

FINDING THE SYMMETRIC DIFFERENCE OF SETS

The symmetric difference of two sets is a set that contains the elements that are not shared by the sets. In other words, it is the elements that are in one set but not in both. In Python, you can call the symmetric_difference method to get the symmetric difference of two sets. Here is the general format:

```
set1.symmetric_difference(set2)
```

In the general format, set1 and set2 are sets. The method returns a set that contains the elements that are found in either set1 or set2 but not both sets.
The following interactive session demonstrates:

```
>>> set1 = set([1, 2, 3, 4])
>>> set2 = set([3, 4, 5, 6])
>>> set3 = set1.symmetric_difference(set2)
>>> set3
{1, 2, 5, 6}
>>>
```

You can also use the ^ operator to find the symmetric difference of two sets. Here is the general format of an expression using the ^ operator with two sets:

```
set1 ^ set2
```

In the general format, set1 and set2 are sets. The expression returns a set that contains the elements that are found in either set1 or set2 but not both sets. The following interactive session demonstrates:

```
>>> set1 = set([1, 2, 3, 4])
>>> set2 = set([3, 4, 5, 6])
>>> set3 = set1 ^ set2
>>> set3
{1, 2, 5, 6}
>>>
```

FINDING SUBSETS AND SUPERSETS

Suppose you have two sets and one of those sets contains all of the elements of the other set. Here is an example:

```
set1 = set([1, 2, 3, 4])
set2 = set([2, 3])
```

In this example, set1 contains all the elements of set2, which means that set2 is a subset of set1. It also means that set1 is a superset of set2. In Python, you can call the issubset method to determine whether one set is a subset of another. Here is the general format:

```
set2.issubset(set1)
```

In the general format, set1 and set2 are sets. The method returns True if set2 is a subset of set1. Otherwise, it returns False. You can call the issuperset method to determine whether one set is a superset of another. Here is the general format:

```
set1.issuperset(set2)
```

In the general format, set1 and set2 are sets. The method returns True if set1 is a superset of set2. Otherwise, it returns False. The following interactive session demonstrates:

```
>>> set1 = set([1, 2, 3, 4])
>>> set2 = set([2, 3])
>>> set2.issubset(set1)
True
>>> set1.issuperset(set2)
True
>>>
```

You can also use the <= operator to determine whether one set is a subset of another and the >= operator to determine whether one set is a superset of another. Here is the general format of an expression using the <= operator with two sets:

```
set2 <= set1
```

In the general format, set1 and set2 are sets. The expression returns True if set2 is a subset of set1. Otherwise, it returns False. Here is the general format of an expression using the >= operator with two sets:

```
set1 >= set2
```

In the general format, set1 and set2 are sets. The expression returns True if set1 is a subset of set2. Otherwise, it returns False. The following interactive session demonstrates:

```
>>> set1 = set([1, 2, 3, 4])
>>> set2 = set([2, 3])
>>> set2 <= set1
True
>>> set1 >= set2
True
>>> set1 <= set2
False
```

SETS

```
# This program demonstrates various set operations.
baseball = set(['Jodi', 'Carmen', 'Aida', 'Alicia'])
basketball = set(['Eva', 'Carmen', 'Alicia', 'Sarah'])

# Display members of the baseball set.
print('The following students are on the baseball team:')
for name in baseball:
    print(name)

# Display members of the basketball set.
print()
print('The following students are on the basketball team:')
for name in basketball:
    print(name)

# Demonstrate intersection
print()
print('The following students play both baseball and basketball:')
for name in baseball.intersection(basketball):
    print(name)

# Demonstrate union
print()
print('The following students play either baseball or basketball:')
for name in baseball.union(basketball):
    print(name)

# Demonstrate difference of baseball and basketball
print()
print('The following students play baseball, but not basketball:')
for name in baseball.difference(basketball):
    print(name)

# Demonstrate difference of basketball and baseball
print()
print('The following students play basketball, but not baseball:')
for name in basketball.difference(baseball):
    print(name)

# Demonstrate symmetric difference
print()
print('The following students play one sport, but not both:')
for name in baseball.symmetric_difference(basketball):
    print(name)
```

SETS

Python programmers also have access to the sets, which are much like the keys of a valueless dictionary; they don’t map keys to values, but can often be used like dictionaries for fast lookups when there is no associated value, especially in search routines:

```
>>> D = {}
>>> D['state1'] = True              # A visited-state dictionary
>>> 'state1' in D
True
>>> S = set()
>>> S.add('state1')                 # Same, but with sets
>>> 'state1' in S
True
```

DICTIONARY VIEWS AND SETS

- view objects returned by the keys method are setlike and support common set operations such *as intersection and union;*
- values views are not set-like, but items results are setlike if their (key, value) pairs are unique and hashable (immutable).
- Given that sets behave much like *valueless dictionaries* (and may even be coded in curly braces like dictionaries in 3.X and 2.7), this is a logical symmetry.
- Set items are unordered, unique, and immutable, just like dictionary keys.

Dictionary value views are *never set-like*, since *their items are not necessarily unique or immutable*. Here is what keys views look like when used in set operations.

```
>>> K, V
(dict_keys(['c', 'a']), dict_values([3, 1]))

>>> K | {'x': 4} # Keys (and some items) views are set-like --- union
{'c', 'x', 'a'}

>>> V & {'x': 4} # not set-like - intersection
TypeError: unsupported operand type(s) for &: 'dict_values' and 'dict'

>>> V & {'x': 4}.values()
TypeError: unsupported operand type(s) for &: 'dict_values' and 'dict_values'
```

In set operations, views may be mixed with other views, sets, and dictionaries; dictionaries are treated the same as their keys views in this context:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'a': 1, 'b': 2, 'c': 3}
>>> D.keys()
dict_keys(['a', 'b', 'c'])
>>> D.keys() & D.keys() # Intersect keys views
{'b', 'c', 'a'}
>>> D.keys() & {'b'} # Intersect keys and set
{'b'}
>>> D.keys() & {'b': 1} # Intersect keys and dictionary
{'b'}
>>> D.keys() | {'b', 'c', 'd'} # Union keys and set
{'b', 'c', 'a', 'd'}
>>> D.keys() | {'E': 9} # Union keys and dictionary
{'c', 'b', 'a', 'E'}
>>> D.keys() | D.keys() # Union keys views
{'a', 'b', 'c'}
>>> E={'X':9}
>>> D.keys() | E.keys()
{'X', 'a', 'b', 'c'}
>>> D.keys() & E.keys()
set()
```

Items views are set-like too if they are hashable—that is, if they contain only immutable objects:

```
>>> D = {'a': 1}
>>> list(D.items()) # Items set-like if hashable
[('a', 1)]
>>> D.items()
dict_items([('a', 1)])
>>> D.keys()
dict_keys(['a'])
>>> D.items() | D.keys() # Union view and view
{('a', 1), 'a'}
>>> D.items() | D # dict treated same as its keys
{('a', 1), 'a'}
>>> D.items() | {('c', 3), ('d', 4)} # Set of key/value pairs
{('d', 4), ('a', 1), ('c', 3)}
>>> dict(D.items() | {('c', 3), ('d', 4)}) # dict accepts iterable sets too
{'c': 3, 'a': 1, 'd': 4}
```