Assignment 2: List, Tuples and Sets

Objectives

- Students will be able to understand list, tuples, sets and dictionary data type and its operations
- Students will be to use list, tuples, sets and dictionary data type and its functions
- Students will be able to apply suitable list, tuples, sets and dictionary functions to solve given programming problem

Reading

You should read the following topics before starting this exercise

Python Lists: Concept, creating and accessing elements, updating & deleting lists, traversing a List, reverse Built-in List Operators, Concatenation, Repetition, In Operator, Built-in List functions and methods.

Tuples, Accessing values in Tuples, Tuple Assignment, Tuples as return values, Variable-length argument tuples, and Basic tuples operations, Concatenation, Repetition, in Operator, Iteration, Built-in tuple functions, indexing, slicing and matrices.

Ready Reference and Self Activity

Background

for in Loop: For loops are used for sequential traversal. For example: traversing a list or string or array etc. In Python, there is no C style for loop, i.e., for (i=0; i<n; i++). There is "for in" loop which is similar to for_each loop in other languages. Let us learn how to use for in loop for sequential traversals.

Syntax:

```
for iterator_var in sequence:
    statements(s)
```

It can be used to iterate over a range and iterators.

```
# Python program to illustrate # Iterating over range 0 to n-1
n = 3 for i in range(0, n):
    print(i)
```

Output-

```
0
1
2
```

```
#Python program to illustrate #Iterating over a list
print("List Iteration")
l = ["cmcs", "for","cmcs"]
for i in l:
   print(i)
# Iterating over a tuple(immutable)
print("\nTuple Iteration")
t = ("cmcs", "for", "cmcs")
for i in t:
    print(i)
# Iterating over a String
print ("\nString Iteration")
s = "Cmcs"
for i in s :
   print(i)
# Iterating over dictionary
print("\nDictionary Iteration")
d = dict()
d['xyz'] = 123
d['abc'] = 345
for i in d :
print("%s  %d" %(i, d[i]))
```

Output-

```
List Iteration
cmcs
for
cmcs
Tuple Iteration cmcs
for
cmcs
String Iteration
C
```

m

c

s

Dictionary Iteration

xyz 123

abc 345

How for loop in Python works internally?

Before proceeding to this section, you should have a prior understanding of Python Iterators.

Firstly, lets see how a simple for loop looks like.

```
# A simple for loop example
fruits = ["apple", "orange", "kiwi"]
for fruit in fruits:
    print(fruit)
```

Output-

apple

orange

kiwi

Python List

Here we can see the for loops iterates over a iterable object fruits which is a list. Lists, sets, dictionary these are few iterable objects while an integer object is not an iterable object.

For loops can iterate over any iterable object (example: List, Set, Dictionary, Tuple or String).

Now with the help of above example lets dive deep and see what happens internally here.

1. Make the list (iterable) an iterable object with help of iter() function.
2. Run a infinite while loop and break only if the StopIteration is raised.
3. In the try block we fetch the next element of fruits with next() function.
4. After fetching the element we did the operation to be performed in with the element. (i.e print(fruit))

```
list1 =['Maharashtra','Gujrat',1998,1999];
list2 =[1,2,3,4,5,6,7]; print"list1[0]: ",
list1[0] print"list2[1:5]: ", list2[1:5]
```

Output-

```
list1[0]:  Maharashtra list2[1:5]:
[2, 3, 4, 5]
```

Updating Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. For example −

```
list =['Maharashtra','Gujrat',1998,1999];
print"Value available at index 2 : "
print list[2] list[2]=2001;
```

```
print"New value available at index 2 : " print list[2]
```

Output-

```
Value available at index 2 :
1998
New value available at index 2 :
2001
```

Note − append() method is discussed in subsequent section.

Delete List Elements

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know. For example −

```
list1 =['Maharashtra','Gujrat',1998,1999];
print list1 del list1[2]; print"After
deleting value at index 2 : " print list1
```

Output-

```
 ['Maharashtra', 'Gujrat', 1998, 1999]
After deleting value at index 2 :
['Maharashtra', 'Gujrat', 1999]
```

Note − remove() method is discussed in subsequent section.

Basic List Operations

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

Indexing, Slicing, and Matrixes

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input − L
= ['spam', 'Spam', 'SPAM!']

Python Expression	Results	Description
L[2]	SPAM!	Offsets start at zero
L[-2]	Spam	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

Built-in List Functions & Methods

Python includes the following list functions −

Function	Use
cmp(list1, list2)	Compares elements of both lists.
len(list)	Gives the total length of the list.
max(list)	Returns item from the list with max value.

min(list)	Returns item from the list with min value.
list(seq)	Converts a tuple into list.

Python includes following list methods

Methods	**Use**
list.append(obj)	Appends object obj to list
list.count(obj)	Returns count of how many times obj occurs in list
list.extend(seq)	Appends the contents of seq to list
list.index(obj)	Returns the lowest index in list that obj appears
list.insert(index, obj)	Inserts object obj into list at offset index
list.pop(obj=list[-1])	Removes and returns last object or obj from list
list.remove(obj)	Removes object obj from list
list.reverse()	Reverses objects of list in place
list.sort([func])	Sorts objects of list, use compare func if given

Python Tuples

A tuple is a collection of objects which ordered and immutable. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these commaseparated values between parentheses also. For example −

```
tup1 = ('Maharashtra', 'Gujrat', 1998, 1999);
tup2 = (1, 2, 3, 4, 5 ); tup3 = "a", "b", "c",
"d";
```

The empty tuple is written as two parentheses containing nothing − tup1 = ();

To write a tuple containing a single value you have to include a comma, even though there is only one value − tup1 = (50,);

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

Accessing Values in Tuples

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

```
tup1 =('Maharashtra','Gujrat',1998,1999);
tup2 =(1,2,3,4,5,6,7); print"tup1[0]: ",
tup1[0]; print"tup2[1:5]: ", tup2[1:5];
```

Output- tup1[0]: Maharashtra tup2[1:5]: [2, 3, 4, 5]

Updating Tuples

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates −

```
tup1 =(12,34.56);
tup2 =('abc','xyz');
# Following action is not valid for tuples
# tup1[0] = 100;
# So let's create a new tuple as follows tup3
= tup1 + tup2;
```

```
print tup3;
```

Output-

(12, 34.56, 'abc', 'xyz')

Delete Tuple Elements

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the **del** statement. For example −

```
tup =('Maharashtra','Gujrat',1998,1999);
print tup; del tup; print"After deleting
tup : "; print tup;
```

Output-

```
('Maharashtra', 'Gujrat', 1998, 1999) After
deleting tup :
Traceback (most recent call last):
File "test.py", line 9, in <module>
print tup;
NameError: name 'tup' is not defined
#Note an exception raised, this is because after del tup tuple does not exist anymore −
```

Basic Tuples Operations

Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

In fact, tuples respond to all of the general sequence operations we used on strings in the prior chapter −

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

Indexing, Slicing, and Matrixes

Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input −

```
L =('spam','Spam','SPAM!')
```

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

No Enclosing Delimiters

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples −

```
print'abc',-4.24e93,18+6.6j,'xyz'; x, y =1,2;
print"Value of x , y : ", x,y; Output-
abc -4.24e+93 (18+6.6j) xyz
Value of x , y : 1 2
```

Built-in Tuple Functions

Python includes the following tuple functions −

Function	Use
cmp(tuple1, tuple2)	Compares elements of both tuples.
len(tuple)	Gives the total length of the tuple.
max(tuple)	Returns item from the tuple with max value.
min(tuple)	Returns item from the tuple with min value.
tuple(seq)	Converts a list into tuple.

Python Set

Python's built-in set type has the following characteristics:

- Sets are unordered.
- Set elements are unique. Duplicate elements are not allowed.
- A set itself may be modified, but the elements contained in the set must be of an immutable type.

Let's see what all that means, and how you can work with sets in Python.

A set can be created in two ways. First, you can define a set with the built-in set() function:

```
x=set(<iter>)
```

In this case, the argument <iter> is an iterable—again, for the moment, think list or tuple—that generates the list of objects to be included in the set. This is analogous to the <iter> argument given to the .extend() list method:

```
>>>x=set(['zoo','cat','jaz','zoo','box'])
>>>x
{'box', 'zoo', 'cat', 'jaz'}

>>>x=set(('zoo','cat','jaz','zoo','box'))
>>>x
{'box', 'zoo', 'cat', 'jaz'}
```

Strings are also iterable, so a string can be passed to set() as well. You have already seen that list(s) generates a list of the characters in the string s. Similarly, set(s) generates a set of the characters in s:

```
>>>s='quux'
>>>list(s)
['q', 'u', 'u', 'x']
>>>set(s)
{'x', 'u', 'q'}
```

You can see that the resulting sets are unordered: the original order, as specified in the definition, is not necessarily preserved. Additionally, duplicate values are only represented in the set once, as with the string 'zoo' in the first two examples and the letter 'u' in the third.

Alternately, a set can be defined with curly braces ({}):

```
x={<obj>,<obj>,...,<obj>}
```

When a set is defined this way, each <obj> becomes a distinct element of the set, even if it is an iterable. This behavior is similar to that of the .append() list method. Thus, the sets shown above can also be defined like this:

```
>>>x={'zoo','cat','jaz','zoo','box'}
>>>x
{'box', 'zoo', 'cat', 'jaz'}
>>>x={'b','o','o','x'}
>>>x
{'x', 'b', 'o'}
```

To recap:

- The argument to set() is an iterable. It generates a list of elements to be placed into the set.

- The objects in curly braces are placed into the set intact, even if they are iterable.

Observe the difference between these two set definitions:

```
>>>{'zoo'}
{'zoo'}
>>>set('zoo')
{'o', 'z'}
```

A set can be empty. However, recall that Python interprets empty curly braces ({}) as an empty dictionary, so the only way to define an empty set is with the set() function:

```
>>>x=set()
>>>type(x)
<class 'set'>
>>>x
set()
>>>x={}
>>>type(x)
<class 'dict'>
```

An empty set is falsy in a Boolean context:

```
>>>x=set()
>>>bool(x)
False
>>>xor1
1
>>>xand1
set()
```

You might think the most intuitive sets would contain similar objects—for example, even numbers or surnames:

```
>>>s1={2,4,6,8,10}
>>>s2={'Smith','McArthur','Wilson','Johansson'}
```

Python does not require this, though. The elements in a set can be objects of different types: >>>x={42,'zoo',3.14159,None}

```
>>>x
{None, 'zoo', 42, 3.14159}
```

Don’t forget that set elements must be immutable. For example, a tuple may be included in a set:

```
>>>x={42,'zoo',(1,2,3),3.14159}
```

```
>>>x
{42, 'zoo', 3.14159, (1, 2, 3)}
```

But lists and dictionaries are mutable, so they can’t be set elements:

```
>>>a=[1,2,3]
>>>{a}
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
{a}
TypeError: unhashable type: 'list'
>>>d={'a':1,'b':2}
>>>{d}
Traceback (most recent call last):
  File "<pyshell#72>", line 1, in <module>
{d}
TypeError: unhashable type: 'dict'
```

Set Size and Membership

The len() function returns the number of elements in a set, and the in and not in operators can be used to test for membership:

```
>>>x={'zoo','cat','jaz'}
>>>len(x)
3
>>>'cat' in x
True
>>>'box' in x
False
```

Operating on a Set

Many of the operations that can be used for Python’s other composite data types don’t make sense for sets. For example, sets can’t be indexed or sliced. However, Python provides a whole host of operations on set objects that generally mimic the operations that are defined for mathematical sets.

Operators vs. Methods

Most, though not quite all, set operations in Python can be performed in two different ways: by operator or by method.

Let’s take a look at how these operators and methods work, using set union as an example.

Given two sets, x1 and x2, the union of x1 and x2 is a set consisting of all elements in either set.

Consider these two sets:

```
x1={'zoo','cat','jaz'} x2={'jaz','box','quux'}
```

The union of x1 and x2 is {'zoo', 'cat', 'jaz', 'box', 'quux'}.

Note: Notice that the element 'jaz', which appears in both x1 and x2, appears only once in the union. Sets never contain duplicate values.

In Python, set union can be performed with the | operator:

```
>>>x1={'zoo','cat','jaz'}
>>>x2={'jaz','box','quux'}
>>>x1|x2
{'jaz', 'quux', 'box', 'cat', 'zoo'}
```

Set union can also be obtained with the .union() method. The method is invoked on one of the sets, and the other is passed as an argument:

```
>>>x1.union(x2)
{'jaz', 'quux', 'box', 'cat', 'zoo'}
```

The way they are used in the examples above, the operator and method behave identically. But there is a subtle difference between them. When you use the | operator, both operands must be sets. The .union() method, on the other hand, will take any iterable as an argument, convert it to a set, and then perform the union.

Observe the difference between these two statements:

```
>>>x1|('jaz','box','quux')
Traceback (most recent call last):
  File "<pyshell#43>", line 1, in <module> x1|('jaz','box','quux')
TypeError: unsupported operand type(s) for |: 'set' and 'tuple'
>>>x1.union(('jaz','box','quux'))
{'jaz', 'quux', 'box', 'cat', 'zoo'}
```

Both attempt to compute the union of x1 and the tuple ('jaz', 'box', 'quux'). This fails with the | operator but succeeds with the .union() method.

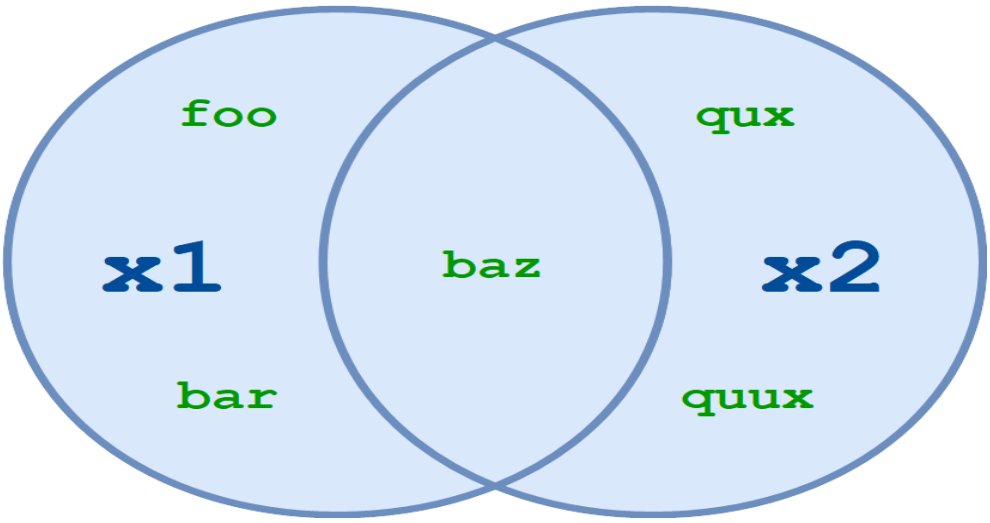
Available Operators and Methods

Below is a list of the set operations available in Python. Some are performed by operator, some by method, and some by both. The principle outlined above generally applies: where a set is expected, methods will typically accept any iterable as an argument, but operators require actual sets as operands.

Compute the union of two or more sets.

x1.union(x2[, x3 ...]) x1

| x2 [| x3 ...]



Set Union

x1.union(x2) and x1 | x2 both return the set of all elements in either x1 or x2:

```
>>>
>>>x1={'zoo','cat','jaz'}
>>>x2={'jaz','box','quux'}


>>>x1.union(x2)
{'zoo', 'box', 'quux', 'jaz', 'cat'}


>>>x1|x2
{'zoo', 'box', 'quux', 'jaz', 'cat'}
```

More than two sets may be specified with either the operator or the method:

```
>>>a={1,2,3,4}
>>>b={2,3,4,5}
```

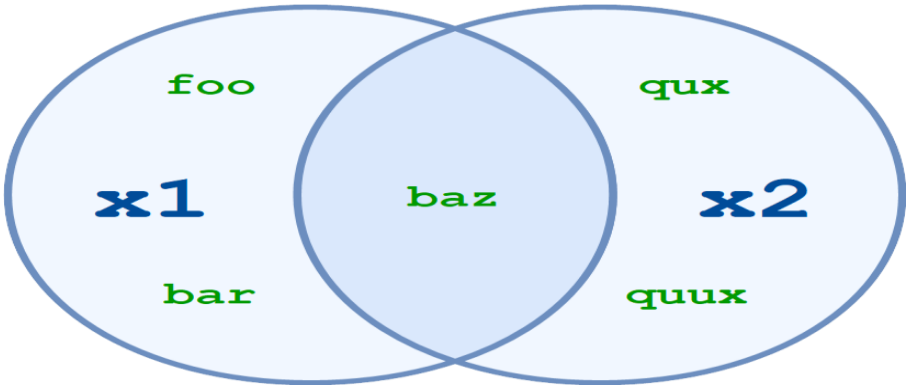
```
>>>c={3,4,5,6}
>>>d={4,5,6,7}
>>>a.union(b,c,d)
{1, 2, 3, 4, 5, 6, 7}
>>>a|b|c|d
{1, 2, 3, 4, 5, 6, 7}
```

The resulting set contains all elements that are present in any of the specified sets.

Compute the intersection of two or more sets.

x1.intersection(x2[, x3 ...]) x1

& x2 [& x3 ...]



Set Intersection

x1.intersection(x2) and x1 & x2 return the set of elements common to both x1 and x2:

```
>>>x1={'zoo','cat','jaz'}
>>>x2={'jaz','box','quux'}
>>>x1.intersection(x2)
{'jaz'}
>>>x1&x2
{'jaz'}
```

You can specify multiple sets with the intersection method and operator, just like you can with set union:

```
>>>a={1,2,3,4}
>>>b={2,3,4,5}
>>>c={3,4,5,6}
>>>d={4,5,6,7}
>>>a.intersection(b,c,d)
{4}
>>>a&b&c&d
```

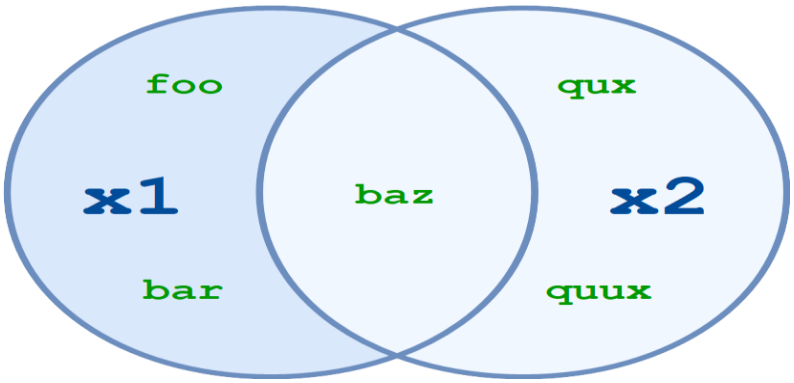
```
{4}
```

The resulting set contains only elements that are present in all of the specified sets.

Compute the difference between two or more sets.

x1.difference(x2[, x3 ...]) x1

- x2 [- x3 ...]



Set Difference

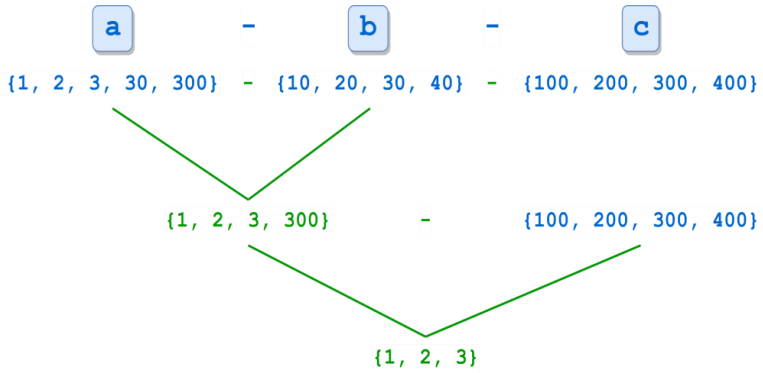
x1.difference(x2) and x1 - x2 return the set of all elements that are in x1 but not in x2:

```
>>>
>>>x1={'zoo','cat','jaz'}
>>>x2={'jaz','box','quux'}
>>>x1.difference(x2)
{'zoo', 'cat'}
>>>x1-x2
{'zoo', 'cat'}
```

Another way to think of this is that x1.difference(x2) and x1 - x2 return the set that results when any elements in x2 are removed or subtracted from x1. Once again, you can specify more than two sets:

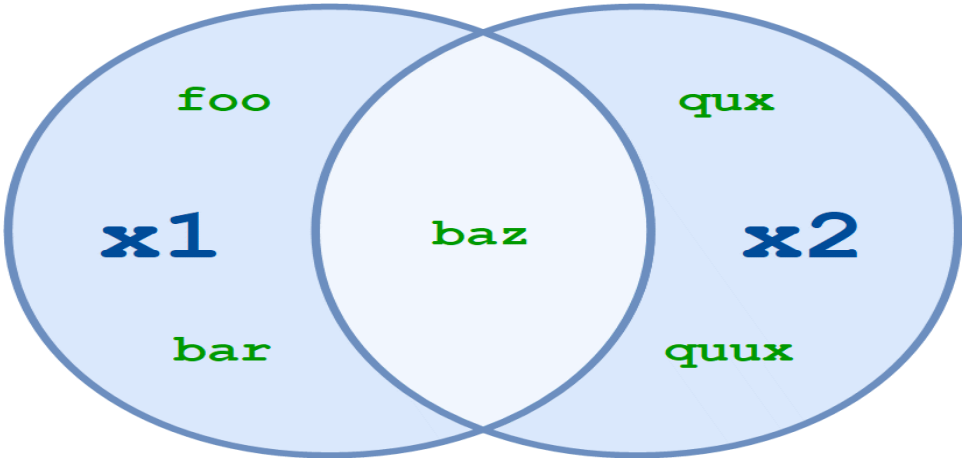
```
>>>a={1,2,3,30,300}
>>>b={10,20,30,40}
>>>c={100,200,300,400}
>>>a.difference(b,c)
{1, 2, 3}
>>>a-b-c
{1, 2, 3}
```

When multiple sets are specified, the operation is performed from left to right. In the example above, a - b is computed first, resulting in {1, 2, 3, 300}. Then c is subtracted from that set, leaving {1, 2, 3}:



Compute the symmetric difference between sets.

x1.symmetric_difference(x2) x1 ^ x2 [^ x3 ...]



Set Symmetric Difference

x1.symmetric_difference(x2) and x1 ^ x2 return the set of all elements in either x1 or x2, but not both:

```
>>>
>>>x1={'zoo','cat','jaz'}
>>>x2={'jaz','box','quux'}
>>>x1.symmetric_difference(x2)
{'zoo', 'box', 'quux', 'cat'}
>>>x1^x2
{'zoo', 'box', 'quux', 'cat'}
The ^ operator also allows more than two sets:
>>>
>>>a={1,2,3,4,5}
>>>b={10,2,3,4,50}
>>>c={1,50,100}
```

```
>>>a^b^c
{100, 5, 10}
```

As with the difference operator, when multiple sets are specified, the operation is performed from left to right.

Curiously, although the ^ operator allows multiple sets, the .symmetric_difference() method doesn't:

```
>>>
>>>a={1,2,3,4,5}
>>>b={10,2,3,4,50}
>>>c={1,50,100}
>>>a.symmetric_difference(b,c)
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module> a.symmetric_difference(b,c)
TypeError: symmetric_difference() takes exactly one argument (2 given) x1.isdisjoint(x2)
```

Determines whether or not two sets have any elements in common. x1.isdisjoint(x2) returns True if x1 and x2 have no elements in common:

```
>>>x1={'zoo','cat','jaz'}
>>>x2={'jaz','box','quux'}

>>>x1.isdisjoint(x2)
False
>>>x2-{'jaz'}
{'quux', 'box'}
>>>x1.isdisjoint(x2-{'jaz'})
True
```

If x1.isdisjoint(x2) is True, then x1 & x2 is the empty set:

```
>>>x1={1,3,5}
>>>x2={2,4,6}
>>>x1.isdisjoint(x2)
True
>>>x1&x2
set()
```

Note: There is no operator that corresponds to the .isdisjoint() method.

Determine whether one set is a subset of the other.

x1.issubset(x)

x1 <= x2

In set theory, a set x1 is considered a subset of another set x2 if every element of x1 is in x2. x1.issubset(x2) and x1 <= x2 return True if x1 is a subset of x2:

```
>>>x1={'zoo','cat','jaz'}
>>>x1.issubset({'zoo','cat','jaz','box','quux'})
True
>>>x2={'jaz','box','quux'}
>>>x1<=x2
False
```

A set is considered to be a subset of itself:

```
>>>x={1,2,3,4,5}
>>>x.issubset(x)
True
>>>x<=x True
```

It seems strange, perhaps. But it fits the definition—every element of x is in x.

x1 < x2

Determines whether one set is a proper subset of the other.

A proper subset is the same as a subset, except that the sets can't be identical. A set x1 is considered a proper subset of another set x2 if every element of x1 is in x2, and x1 and x2 are not equal.

x1 < x2 returns True if x1 is a proper subset of x2:

```
>>>x1={'zoo','cat'}
>>>x2={'zoo','cat','jaz'}
>>>x1<x2
True
>>>x1={'zoo','cat','jaz'}
>>>x2={'zoo','cat','jaz'}
>>>x1<x2
False
```

While a set is considered a subset of itself, it is not a proper subset of itself:

```
>>>x={1,2,3,4,5}
>>>x<=x
True
>>>x<x
```

```
False
```

Note: The < operator is the only way to test whether a set is a proper subset. There is no corresponding method.

Determine whether one set is a superset of the other.

x1.issuperset(x2) x1 >= x2

A superset is the reverse of a subset. A set x1 is considered a superset of another set x2 if x1 contains every element of x2. x1.issuperset(x2) and x1 >= x2 return True if x1 is a superset of x2:

```
>>>x1={'zoo','cat','jaz'}
>>>x1.issuperset({'zoo','cat'})
True
>>>x2={'jaz','box','quux'}
>>>x1>=x2
False
```

You have already seen that a set is considered a subset of itself. A set is also considered a superset of itself:

```
>>>x={1,2,3,4,5}
>>>x.issuperset(x)
True
>>>x>=x
True x1
> x2
```

Determines whether one set is a proper superset of the other.

A proper superset is the same as a superset, except that the sets can't be identical. A set x1 is considered a proper superset of another set x2 if x1 contains every element of x2, and x1 and x2 are not equal.

x1 > x2 returns True if x1 is a proper superset of x2:

```
>>>x1={'zoo','cat','jaz'}
>>>x2={'zoo','cat'}
>>>x1>x2
True
>>>x1={'zoo','cat','jaz'}
>>>x2={'zoo','cat','jaz'}
>>>x1>x2
False
A set is not a proper superset of itself:
>>>x={1,2,3,4,5}
>>>x>x
False
```

Note: The > operator is the only way to test whether a set is a proper superset. There is no corresponding method.

Modifying a Set

Although the elements contained in a set must be of immutable type, sets themselves can be modified. Like the operations above, there are a mix of operators and methods that can be used to change the contents of a set.

Augmented Assignment Operators and Methods

Each of the union, intersection, difference, and symmetric difference operators listed above has an augmented assignment form that can be used to modify a set. For each, there is a corresponding method as well.

Modify a set by union. x1.update(x2[, x3 ...]) x1 |= x2 [| x3 ...] x1.update(x2) and x1 |= x2 add to x1 any elements in x2 that x1 does not already have:

```
>>>
>>>x1={'zoo','cat','jaz'}
>>>x2={'zoo','jaz','box'}
>>>x1|=x2
>>>x1
{'box', 'zoo', 'cat', 'jaz'}
>>>x1.update(['corge','garply'])
>>>x1
```

```
{'box', 'corge', 'garply', 'zoo', 'cat', 'jaz'}
```

Modify a set by intersection. x1.intersection_update(x2[, x3 ...]) x1 &= x2 [& x3 ...]

x1.intersection_update(x2) and x1 &= x2 update x1, retaining only elements found in both x1 and x2:

```
>>>x1={'zoo','cat','jaz'}
>>>x2={'zoo','jaz','box'}
>>>x1&=x2
>>>x1
{'zoo', 'jaz'}
>>>x1.intersection_update(['jaz','box'])
>>>x1
{'jaz'}
```

Modify a set by difference. x1.difference_update(x2[, x3 ...]) x1 -= x2 [| x3 ...]

x1.difference_update(x2) and x1 -= x2 update x1, removing elements found in x2:

```
>>>x1={'zoo','cat','jaz'}
>>>x2={'zoo','jaz','box'}
>>>x1-=x2
>>>x1
{'cat'}
>>>x1.difference_update(['zoo','cat','box'])
>>>x1
set()
```

Modify a set by symmetric difference. x1.symmetric_difference_update(x2) x1 ^= x2

x1.symmetric_difference_update(x2) and x1 ^= x2 update x1, retaining elements found in either x1 or x2, but not both:

```
>>>x1={'zoo','cat','jaz'}
>>>x2={'zoo','jaz','box'}
>>>x1^=x2
>>>x1
{'cat', 'box'}
>>>x1.symmetric_difference_update(['box','corge'])
```

```
>>>x1
{'cat', 'corge'}
```

Other Methods For Modifying Sets

Aside from the augmented operators above, Python supports several additional methods that modify sets.

Adds an element to a set. x.add(<elem>)

x.add(<elem>) adds <elem>, which must be a single immutable object, to x:

```
>>>x={'zoo','cat','jaz'}
>>>x.add('box')
>>>x
{'cat', 'jaz', 'zoo', 'box'}
```

Removes an element from a set.

x.remove(<elem>)

x.remove(<elem>) removes <elem> from x.

Python raises an exception if <elem> is not in x:

```
>>>x={'zoo','cat','jaz'}
>>>x.remove('jaz')
>>>x
{'cat', 'zoo'}
>>>x.remove('box')
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module> x.remove('box')
KeyError: 'box'
```

Removes an element from a set. x.discard(<elem>)

x.discard(<elem>) also removes <elem> from x. However, if <elem> is not in x, this method quietly does nothing instead of raising an exception:

```
>>>x={'zoo','cat','jaz'}
>>>x.discard('jaz')
>>>x
{'cat', 'zoo'}
>>>x.discard('box')
>>>x
```

```
{'cat', 'zoo'}
```

Removes a random element from a set. x.pop()

x.pop() removes and returns an arbitrarily chosen element from x. If x is empty, x.pop() raises an exception:

```
>>>x={'zoo','cat','jaz'}
>>>x.pop()
'cat'
>>>x
{'jaz', 'zoo'}
>>>x.pop()
'jaz'
>>>x
{'zoo'}
>>>x.pop()
'zoo'
>>>x
set()
>>>x.pop()
Traceback (most recent call last):
  File "<pyshell#82>", line 1, in <module> x.pop()
KeyError: 'pop from an empty set'
```

Clears a set.

x.clear()

x.clear() removes all elements from x:

```
>>>x={'zoo','cat','jaz'}
>>>x
{'zoo', 'cat', 'jaz'}
>>>x.clear()
>>>x
set()
```

Frozen Sets

Python provides another built-in type called a **frozenset**, which is in all respects exactly like a set, except that a frozenset is immutable. You can perform non-modifying operations on a frozenset:

```
>>>x=frozenset(['zoo','cat','jaz'])
>>>x
frozenset({'zoo', 'jaz', 'cat'})
>>>len(x)
3
>>>x&{'jaz','box','quux'}
frozenset({'jaz'})
```

But methods that attempt to modify a frozenset fail:

```
>>>x=frozenset(['zoo','cat','jaz'])
>>>x.add('box')
Traceback (most recent call last):
  File "<pyshell#127>", line 1, in <module> x.add('box')
AttributeError: 'frozenset' object has no attribute 'add'

>>>x.pop()
Traceback (most recent call last):
  File "<pyshell#129>", line 1, in <module> x.pop()
AttributeError: 'frozenset' object has no attribute 'pop'

>>>x.clear()
Traceback (most recent call last):
  File "<pyshell#131>", line 1, in <module> x.clear()
AttributeError: 'frozenset' object has no attribute 'clear'

>>>x
frozenset({'zoo', 'cat', 'jaz'})
```

Deep Dive: Frozensets and Augmented Assignment

Since a frozenset is immutable, you might think it can't be the target of an augmented assignment operator. But observe:

```
>>>f=frozenset(['zoo','cat','jaz'])
>>>s={'jaz','box','quux'}


>>>f&=s
>>>f
frozenset({'jaz'})
```

What gives?

Python does not perform augmented assignments on frozensets in place. The statement x &= s is effectively equivalent to x = x & s. It isn't modifying the original x. It is reassigning x to a new object, and the object x originally referenced is gone.

You can verify this with the id() function:

```
>>>f=frozenset(['zoo','cat','jaz'])
>>>id(f)
56992872
>>>s={'jaz','box','quux'}


>>>f&=s
>>>f
frozenset({'jaz'})
>>>id(f)
56992152
```

f has a different integer identifier following the augmented assignment. It has been reassigned, not modified in place.

Some objects in Python are modified in place when they are the target of an augmented assignment operator. But frozensets aren't.

Frozensets are useful in situations where you want to use a set, but you need an immutable object. For example, you can't define a set whose elements are also sets, because set elements must be immutable:

```
>>>x1=set(['zoo'])
>>>x2=set(['cat'])
>>>x3=set(['jaz'])
```

```
>>>x={x1,x2,x3}
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module> x={x1,x2,x3}
TypeError: unhashable type: 'set'
```

If you really feel compelled to define a set of sets (hey, it could happen), you can do it if the elements are frozensets, because they are immutable:

```
>>>x1=frozenset(['zoo'])
>>>x2=frozenset(['cat'])
>>>x3=frozenset(['jaz'])
>>>x={x1,x2,x3}
>>>x
{frozenset({'cat'}), frozenset({'jaz'}), frozenset({'zoo'})}
```

Likewise, recall from the previous tutorial on dictionaries that a dictionary key must be immutable. You can't use the built-in set type as a dictionary key:

```
>>>x={1,2,3}
>>>y={'a','b','c'}
>>>
>>>d={x:'zoo',y:'cat'}
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module> d={x:'zoo',y:'cat'}
TypeError: unhashable type: 'set'
```

If you find yourself needing to use sets as dictionary keys, you can use frozensets:

```
>>>x=frozenset({1,2,3})
>>>y=frozenset({'a','b','c'})
>>>
>>>d={x:'zoo',y:'cat'}
>>>d
{frozenset({1, 2, 3}): 'zoo', frozenset({'c', 'a', 'b'}): 'cat'}
```

SET A

- **List**

1. Write a Python program to sum all the items in a list.

2. Write a Python program to multiplies all the items in a list.

- **Tuples**
    1. Write a Python program to create a tuple.

    2. Write a Python program to create a tuple with different data types.

    3. Write a Python program to check whether an element exists within a tuple.

- **Sets**
    1. Write a Python program to create a set.

    2. Write a Python program to iterate over sets.

3. Write a Python program to create set difference.

SET B

- **List**
  1. Write a Python program to remove duplicates from a list.

2. Write a Python program to check a list is empty or not.

- **Tuples**
  1. Write a Python program to convert a list to a tuple.

2. Write a Python program to remove an item from a tuple.

3. Write a Python program to slice a tuple.

4. Write a Python program to find the length of a tuple.

- **Sets**
  1. Write a Python program to check if a set is a subset of another set.

2. Write a Python program to find maximum and the minimum value in a set.

3. Write a Python program to find the length of a set.

Assignment Evaluation

0: Not Done [ ] 1 : Incomplete [ ] 2 : Late Complete[]

3: Needs Improvement [ ] 4 : Complete [ ] 5 : Well Done [ ]

Signature of Instructor