Sets

A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference. Sets can also have important efficiency benefits.

One Motivation -- Lists can be slooooooow....

One motivation for using sets is that several important operations (adding an element, determining whether an element is in the set) take *constant time* regardless of the size of the set, rather than linear time in the size of the list.

In [1]:
```
big_num = 10000000 # ten million
big_num_list = list(range(big_num))
big_num_set = set(big_num_list)
```

In [2]:
```
small_num = 100
small_num_list = list(range(big_num - small_num, big_num))

# how many of small_num_list elements are in big_num_list?
import time
start = time.time()
count = 0
print("counting...")
for i in small_num_list:
    count = count + 1 if i in big_num_list else 0
end = time.time()
print("count using list:", count, "time:", end-start, "sec")
```

```
counting...
count using list: 100 time: 26.44185996055603 sec
```

In [3]:
```
# how many of small_num_list elements are in big_num_set?
start = time.time()
count = 0
## small_num_list = big_num_list
print("counting...")
for i in small_num_list:
    count = count + 1 if i in big_num_set else 0
end = time.time()
print("count using set:", count, "time:", end-start, "sec")
count_intersection = len(big_num_set.intersection(set(small_num_list)))
end2 = time.time()
print("count using set intersection:", count_intersection, "time:", end2-end, "sec")
```

```
counting...
count using set: 100 time: 0.0015420913696289062 sec
count using set intersection: 100 time: 0.004267692565917969 sec
```

Another Motivation -- Conceptual clarity with set operations

In [4]:
```python
# Lists can have duplicate elements, and lists are ordered
basket = ['apple', 'orange', 'apple', 'pear', 'orange']

# Creating a set from a list results in a set without duplicate elements
fruit1 = set(basket)
print(fruit1)
```

```
{'pear', 'orange', 'apple'}
```

In [5]:
```python
# Adding the same element again to a set doesn't change the set
fruit1.add('apple')
print(fruit1)
```

```
{'pear', 'orange', 'apple'}
```

In [6]:
```python
# But adding a different element does change (mutate) the set...
fruit1.add('banana')
print(fruit1)
```

```
{'pear', 'orange', 'banana', 'apple'}
```

In [7]:
```python
# Can discard/remove elements
fruit1.discard('grape')  #no exception if element not in set
fruit1.remove('apple') #exception if element not in set
print(fruit1)
```

```
{'pear', 'orange', 'banana'}
```

In [8]:
```python
# Sets are unordered: cannot index or slice into a set
fruit1[0:]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-573d82862ca7> in <module>()
      1 # Sets are unordered: cannot index or slice into a set
----> 2 fruit1[0:]

TypeError: 'set' object is not subscriptable
```

In [9]:
```python
# Can still iterate over the elements in a set, in loops or comprehensions
for elt in fruit1:
    if 'n' in elt:
        print(elt)

print([elt for elt in fruit1 if 'n' in elt])
print({elt for elt in fruit1 if 'n' in elt})
```

```
orange
banana
['orange', 'banana']
{'orange', 'banana'}
```

Basic set operations

In [10]:
```python
fruit2 = {'orange', 'apple', 'berry', 'grape', 'orange'}
print("fruit1 =", fruit1)
print("fruit2 =", fruit2)
```

```
fruit1 = {'pear', 'orange', 'banana'}
fruit2 = {'grape', 'orange', 'berry', 'apple'}
```

In [11]:
```
#Intersection
print("Intersection:", fruit1 & fruit2)

#Union
print("Union:", fruit1 | fruit2)

#Difference
print("Difference, fruit1 - fruit2:", fruit1 - fruit2)
print("Difference, fruit2 - fruit1:", fruit2 - fruit1)

#Symmetric Difference
print("Symmetric Difference:", fruit1 ^ fruit2)
```

```
Intersection: {'orange'}
Union: {'grape', 'orange', 'berry', 'pear', 'banana', 'apple'}
Difference, fruit1 - fruit2: {'pear', 'banana'}
Difference, fruit2 - fruit1: {'grape', 'berry', 'apple'}
Symmetric Difference: {'grape', 'berry', 'pear', 'banana', 'apple'}
```

Some set relations

In [12]:
```
fruit3 = set()  #Create an empty set with 'set()' NOT with '{}'
fruit3.add('banana')
fruit3.add('pear')

print("fruit1 =", fruit1)
print("fruit2 =", fruit2)
print("fruit3 =", fruit3)
```

```
fruit1 = {'pear', 'orange', 'banana'}
fruit2 = {'grape', 'orange', 'berry', 'apple'}
fruit3 = {'pear', 'banana'}
```

In [13]:
```
#Subset
fruit3.issubset(fruit1)
```

Out[13]:
```
True
```

In [14]:
```
#Disjoint
fruit3.isdisjoint(fruit2)
```

Out[14]:
```
True
```

In [15]:
```
#Superset
fruit1.issuperset(fruit3)
```

Out[15]:
```
True
```

What kind of objects can be in a set?

The elements of sets must be immutable hashable objects. Thus numbers, strings, tuples (as long as all elements of the tuple are also immutable/hashable objects) can be members of sets, but lists cannot be members of sets. And sets cannot be members of sets! (See frozensets if you're interested in an immutable/hashable variant of sets, that *can* be elements of a set.) The hashable restriction is what makes it possible to determine whether an element is in a set using constant time with respect to the size of the set; i.e., one does not need to iterate over all elements of a set to determine whether that element is in the set. (See 6.006 for more details on how this hashing works.)

Example: Is the number met before

This section takes advantage of sets to solve a simple problem. Here we input a list of integers, your job is to return a list of Booleans which gives True if the number is met earlier in the input list and False if not.

First we generate a random list of integers, ranging from 0 to max_num. You could customize these two parameters to make you own inputs.

In [16]:
```
import random
max_num = 10
size = 20

#Randomly generate an integer list
input = [random.randint(0,max_num) for i in range(size)]
print(input)
```

```
[4, 10, 3, 9, 1, 1, 6, 5, 8, 2, 10, 3, 7, 5, 5, 0, 9, 8, 1, 3]
```

Then, we use a set to store the numbers that we met before, and determine whether the next integer is met or not via the membership testing of set which takes constant time.

In [18]:
```
met = set() #Create an empty set
result = [False]*len(input) #Initialize a list with the same length of input
for index, number in enumerate(input):
    if number in met:
        result[index] = True #Remember that the number has been met again
    else:
        met.add(number) #If not, add it to the set
print("input:", input)
print("result:", result)
```

```
input: [4, 10, 3, 9, 1, 1, 6, 5, 8, 2, 10, 3, 7, 5, 5, 0, 9, 8, 1, 3]
result: [False, False, False, False, False, True, False, False, False, False, True, True, False, Tr
ue, True, False, True, True, True, True]
```

Note: different implementations are possible, and might have different efficiencies. For example, we could create the result list one item at a time, but at the cost of repeated appends:

In [19]:
```
met = set()
result = []
for index, number in enumerate(input):
    if number in met:
        result.append(True)
    else:
        met.add(number) #If not, add it to the set
        result.append(False)
print("input:", input)
print("result:", result)
```

```
input: [4, 10, 3, 9, 1, 1, 6, 5, 8, 2, 10, 3, 7, 5, 5, 0, 9, 8, 1, 3]
result: [False, False, False, False, False, True, False, False, False, False, True, True, False, Tr
ue, True, False, True, True, True, True]
```

In [20]:
```
#Even more (too?) pythonic; many frown upon mutating inside a comprehension
met = set()
result = [True if val in met else (False, met.add(val))[0] for val in input]
print("input:", input)
print("result:", result)
```

```
input: [4, 10, 3, 9, 1, 1, 6, 5, 8, 2, 10, 3, 7, 5, 5, 0, 9, 8, 1, 3]
result: [False, False, False, False, False, True, False, False, False, False, True, True, False, Tr
ue, True, False, True, True, True, True]
```