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 sloooooow....

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))
print(len(big_num_list))
```

```
10000000
```

How long do you think the following will take?

1. less than 1 second
2. longer than 1 second but less than 10 seconds
3. longer than 10 seconds but less than 1 minute
4. longer than 1 minute

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) #side question: parens needed?
end = time.time()
print("count using list:", count, "; time:", end-start, "sec")
```

```
counting...
count using list: 100 ; time: 23.41487193107605 sec
```

How long for the following different version?

In [3]:
```
# how many of small_num_list elements are in big_num_set?
count = 0
##small_num_list = big_num_list
print("counting...")
start = time.time()
big_num_set = set(big_num_list) #include the time to build this
end1 = time.time()
print("time to build big_num_set:", end1-start, "sec")
for i in small_num_list:
    count = count + (1 if i in big_num_set else 0)
end2 = time.time()
print("count using set:", count, "; time:", end2-end1, "sec")

start = time.time()
small_num_set = set(small_num_list)
count_intersection = len(big_num_set.intersection(small_num_set))
end = time.time()
print("count using set intersection:", count_intersection, "; time:", end-start, "sec")
```

```
counting...
time to build big_num_set: 0.7173233032226562 sec
count using set: 100 ; time: 0.0009999275207519531 sec
count using set intersection: 100 ; time: 0.0 sec
```

Another Motivation -- Conceptual clarity with set operations

In [4]:
```
# 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)
```

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

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

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

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

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

In [7]:
```
# 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)
```

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

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-c2d8b5207658> in <module>()
      1 # Sets are unordered: cannot index or slice into a set
----> 2 fruit1[0:]

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

In [9]:
```
# Can 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])  #list comprehension
print({elt for elt in fruit1 if 'n' in elt})  #set comprehension
```

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

Basic set operations

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

```
fruit1 = {'orange', 'pear', 'banana'}
fruit2 = {'orange', 'grape', '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) #elements in union but NOT in intersection
```

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

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 = {'orange', 'pear', 'banana'}
         fruit2 = {'orange', 'grape', '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 hashable objects. Python's primitive immutable data types are all hashable -- e.g., strings, numbers, booleans, `None`. The "values" associated with these types are unique and thus instances of these types can serve as unique members in a set. In contrast, Python's built-in list type is mutable: the "value" of a list instance or object (e.g., `[1, 2]`) can mutate and thus change, so lists are deemed not hashable -- a persistent hash computed on the "value" of the list might change if the list mutates -- and thus lists *cannot* be members of sets.

Tuples are an interesting case -- they can be members of sets *if* the elements of the tuple are themselves (recursively) hashable. Thus `(1, "foo")` can be in a set (it's "value" will never change). But the tuple `(1, [2])` has a second element that could be mutated, and thus this tuple is not hashable and cannot be in a set. By this reasoning, sets themselves 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.)

For those interested: instances of user-defined classes are hashable by default. But the user can control or change the hashable nature of their class, depending on including `__eq__` and `__hash__` methods in their class definition. Read more about that in the Python documentation if you're interested in that advanced concept.

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:

```
In [16]: data = [7, 4, 7, 3]
         #expected output: result = [False, False, True, False]
```

Approach: 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 [17]:
```
met = set() #Create an empty set
result = [False]*len(data) #Initialize a list with the same length of data
for index, number in enumerate(data):
    if number in met:
        result[index] = True #Indicate that the number has been met before
    else:
        met.add(number) #If not, add it to the set of previously met elements
print("data:", data)
print("result:", result)
print("met:", met)
```

```
data: [7, 4, 7, 3]
result: [False, False, True, False]
met: {3, 4, 7}
```

Side note and Caution! In the above, something like `[False]*3` would create the list `[False, False, False]`. Since `False` is immutable, everything is good and no confusion arises when we later change individual elements of the `result` list. But let's say we wanted a list of length 3, but with each element being its own empty list to which we'll add elements. Using `x = [[]]*3` creates a list that prints as `[[], [], []]` so might look good. But each of the elements are the **same** empty list! Thus `x[0].append(1)` results in `x` printings as `[[1], [1], [1]]`, and is a **very common aliasing bug**. You are much better off with a different approach to create a deeper data structure, e.g., using a list comprehension that ensures that each element of the list is it's own instance, such as `x = [[] for _ in range(3)]`.

In [18]:
```
x = [[]]*3
print("x:", x)
x[0].append(1)
print("x now:", x, "-- aliased!\n")

y = [[] for _ in range(3)]
print("y:", y)
y[0].append(1)
print("y now:", y, "-- not aliased")
```

```
x: [[], [], []]
x now: [[1], [1], [1]] -- aliased!

y: [[], [], []]
y now: [[1], [], []] -- not aliased
```

Alternative "is number met before": different implementations to our goal or problem above are possible, and might have different efficiencies. For example, we could create the result list one item at a time, but using repeated appends:

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

```
data: [7, 4, 7, 3]
result: [False, False, True, False]
met: {3, 4, 7}
```

In [20]:
```
#Even more (probably 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 data]
print("data:", data)
print("result:", result)
print("met:", met)
```

```
data: [7, 4, 7, 3]
result: [False, False, True, False]
met: {3, 4, 7}
```