Pair Exercise: Sets

HD Sheets July 2024 updated 11/13/2024, 6/27/2025

For DSE5002

Sources https://docs.python.org/2/library/sets.html

Think Python

https://allendowney.github.io/ThinkPython/chap18.html#sets

Sets are unordered collections of unique objects

They are mutable (meaning we can change them after they are created)

Sets are also iterable, but the order can be unpredictable

sets are defined within curly brackets {}

In [2]:
```
a={1,3,5,7,9,1,1}
a
```

Out[2]:
```
{1, 3, 5, 7, 9}
```

I entered 1 several times, but it only appears once in the set

Values are in the set or not, there is no need to list them more than once

In [3]:
```
#we can iterate a set, but the order may be random

# notice that we are iterating this set

for val in a:
    print(val)
```

```
1
3
5
7
9
```

In [4]:
```
dir(a)
```

Out[4]:
```
['__and__',
 '__class__',
 '__class_getitem__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__iand__',
 '__init__',
 '__init_subclass__',
 '__ior__',
 '__isub__',
 '__iter__',
 '__ixor__',
 '__le__',
 '__len__',
 '__lt__',
 '__ne__',
 '__new__',
 '__or__',
 '__rand__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__ror__',
 '__rsub__',
 '__rxor__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__sub__',
 '__subclasshook__',
 '__xor__',
 'add',
 'clear',
 'copy',
 'difference',
 'difference_update',
 'discard',
 'intersection',
 'intersection_update',
 'isdisjoint',
 'issubset',
 'issuperset',
 'pop',
 'remove',
 'symmetric_difference',
 'symmetric_difference_update',
 'union',
 'update']
```

In [5]:
```
#adding to a set
a.add(11)
a
```

Out[5]:
```
{1, 3, 5, 7, 9, 11}
```

In [6]:
```
# there is a pop function, it removes an arbitrary element
z=a.pop()
print(a)
print(z)
```

```
{3, 5, 7, 9, 11}
1
```

In [7]:
```
# add z back!
a.add(z)
```

In [8]:
```
#merging sets
b={2,4,6,8}
a.update(b)
a
```

Out[8]:
```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 11}
```

In [9]:
```
# adding an alias, aka a shallow copy

c=b
c.pop()
print(b)
print(c)

# note that the c.pop() changed b and c
```

```
{2, 4, 6}
{2, 4, 6}
```

In [11]:
```
#making a copy  or a deep copy
d=a.copy()
d.pop()
print(d)
print(a)

#did the d.pop() change both a and d?
```

```
{2, 3, 4, 5, 6, 7, 8, 9, 11}
{1, 2, 3, 4, 5, 6, 7, 8, 9, 11}
```

In [12]:
```
#find the length of a set

len(a)
```

Out[12]:
```
10
```

Set Operations

```
In [13]: # testing for membership

         y=21
         y in a
```

```
Out[13]: False
```

```
In [14]: y not in a
```

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

```
In [16]: ## classic set operations

         t={"apple","orange","banana"}
         c={"red","green","orange"}

         #union
         t|c
```

```
Out[16]: {'apple', 'banana', 'green', 'orange', 'red'}
```

```
In [17]: #intersection
         t&c
```

```
Out[17]: {'orange'}
```

```
In [18]: #set differences
         t-c
```

```
Out[18]: {'apple', 'banana'}
```

```
In [19]: c-t
```

```
Out[19]: {'green', 'red'}
```

Question/Action

Why is c-t not equal to c-t? What is going on here?

look up symmetric _difference for python sets

Add a markdown cell and explain this

I believe you meant to say "why is t-c not equal to c-t?" the reason for this is because t-c will equal everything in set t that is not in set c and c-t will equal everything in set c not in set t

Symmetric difference returns elements that are in either set but not in both (basically (t - c) ∪ (c - t))

What are sets good for?

Testing for membership

Sets work much faster for the "in" test, since sets are hashed.

the "in" test can be done using a list, but this is not hashed and thus 50 to 100 times slower

Not a big deal for a small project, a huge deal for a bit data set that is repeatedly re-analyzed

What is going on under the hood with sets?

Python sets use *Hashed Storage* which means that the elements in a set are *hashed* by feeding them into a *hash function* which converts them into large integers which are used as the "key" that tells the software where the corresponding value is stored.

Hash storage has roughly constant access time, since it is relying on the hash to find values stored in the data. It does not have to sort through a sequential storage to find things.

It is sort of like a card catalog in a library, you can look up the call number of a book (this is the *key*, the card catalog has in a way hashed the title, author and date of the book to create the call number). If you have the call number, you can then go to the appropriate section of the library and pull the book off the shelf. The hash has allowed you to find the book quickly.

Imagine how long it would take if you had to start at the first shelf in the library and read titles off the shelf until you found your book. This process is akin to searching an unordered list.

If we tried to look up our desired book in the card catalog, and it wasn't there, that would mean our book is not available.
We know this immediately, without looking at the whole library.

So finding out if a value is in a set or not is vastly faster than determining if a value is in a list or not.

For those of you who don't remember card catalogs...

Suppose we want to go to Lowes and find bathtub caulk (not window caulk, or driveway or roof, but bathtub caulk).

We could wander around Lowes, looking at likely departments and looking for our caulk on the shelves. Lowes is big, this will take a while.

Alternatively, we could go to the Lowes website on our phone, set the store to the one we are in and then look up bathtub caulk. This will tell us whether this store has any, what brands it has, and it will give us the row and shelf number of where to find it.

This is again a hash, we picked a specific caulk and the website gave us a code that tells use where to find it. If the website has no location for bathtub caulk, we will know we are out of luck.

```
In [ ]:
```