Storing Data Using Other Collection Types

So far,
We manage to store collections of data using lists via *List.sort()*
Now, you will learn how to sort :

Four different sort options:

- Storing Data Using Sets
- Storing Data Using Tuples
- Storing Data Using Dictionaries

It is up to the programmers to pick the one that best matches their problem in order to keep the code as simple and efficient as possible.

Storing Data Using Sets

- What is SET: A *set* is an unordered collection of distinct items.
- *Unordered* means that items aren't stored in any particular order.
- *Distinct* means that any item appears in a set at most once (no duplicates).

Python has a type called set that allows us to store mutable collections of unordered, distinct items.

Storing Data Using Sets

EX:

```
>>> vowels = {'a', 'e', 'i', 'o', 'u'}
>>> vowels                  ?

>>> vowels = {'a', 'e', 'a', 'a', 'i', 'o', 'u', 'u'}
>>> vowels                  ?

{'a', 'e', 'i', 'o', 'u'} == {'a', 'e', 'a', 'a', 'i', 'o', 'u', 'u'}          ?
```

Variable vowels refers to an object of type set:

```
>>> type(vowels)
<class 'set'>
>>> type({1, 2, 3})                ?
```

Storing Data Using Sets

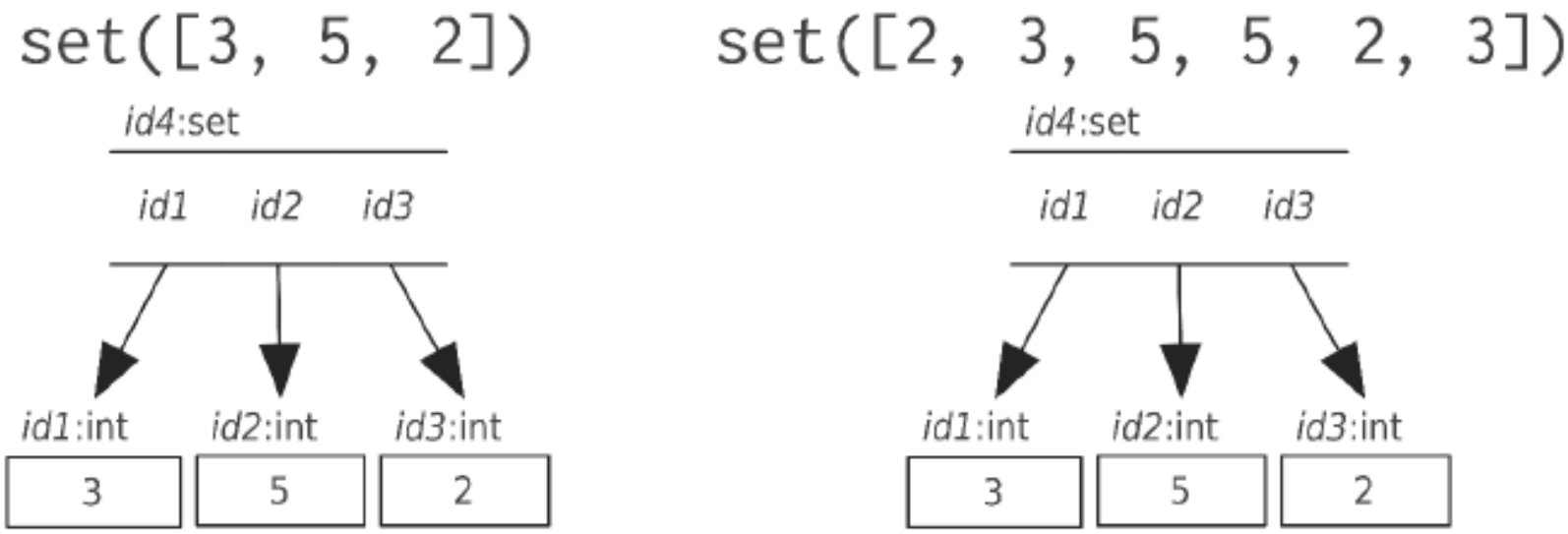
Converting list to set

Set could be created from a list:

Ex:

>>> set([2, 3, 2, 5]) **?**

Here how the set is represented in the memory:



Set Operations

```
>>> ten = set(range(10))                    ?
>>> lows = {0, 1, 2, 3, 4}                  ?
>>> odds = {1, 3, 5, 7, 9}                  ?
>>> lows.add(9)                             ?
>>> lows.difference(odds)                    ?
>>> lows.intersection(odds)                 ?
>>> lows.issubset(ten)                      ?
>>> lows.issuperset(odds)                   ?
>>> lows.remove(0)                           ?
>>> lows.symmetric_difference(odds)         ?
>>> lows.union(odds)                        ?
>>> lows.clear()                            ?
```

Set Operations

Tasks on set performed by methods can also be accomplished using operators.

Method Call	Operator
set1.difference(set2)	set1 - set2
set1.intersection(set2)	set1 & set2
set1.issubset(set2)	set1 <= set2
set1.issuperset(set2)	set1 >= set2
set1.union(set2)	set1 | set2
set1.symmetric_difference(set2)	set1 ^ set2

The following code shows the set operations in action:

```
>>> lows = set([0, 1, 2, 3, 4])
>>> odds = set([1, 3, 5, 7, 9])
>>> lows - odds
>>> lows & odds
>>> lows <= odds
>>> lows >= odds
>>> lows | odds
>>> lows ^ odds
```

Create set from data file

reads each line of the file, strips off the leading and trailing whitespace, and adds the species on that line to the set:

```
>>> observations_file = open('observations.txt')
>>> birds_observed = set()
>>> for line in observations_file:
... bird = line.strip()
... birds_observed.add(bird)
...
>>> birds_observed
```

Storing Data Using Tuples

A **tuple** is a sequence of immutable Python objects. **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.

EX:

```
>>> bases = ('A', 'C', 'G', 'T')
>>> for base in bases:
... print(base)

>>>type(bases)
```

Storing Data Using Tuples

Unlike lists, once a tuple is created, it cannot be mutated:

```
>>> life = (['Canada', 76.5], ['United States', 75.5], ['Mexico', 72.0])
>>> life[0] = life[1]                    ?
```

However, the objects inside it *can* still be mutated:

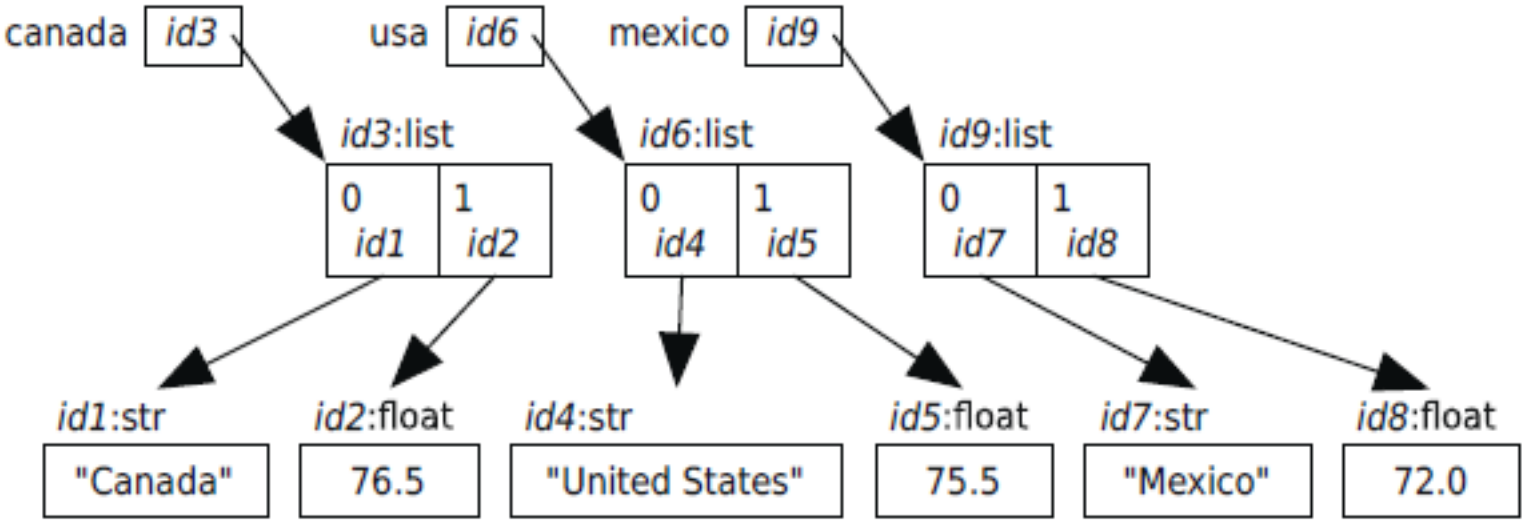
```
>>> life = (['Canada', 76.5], ['United States', 75.5], ['Mexico', 72.0])
>>> life[0][1] = 80.0
>>> life                ?
```

Rather than saying that a tuple cannot change, it is more accurate to say this: the references contained in a tuple cannot be changed after the tuple has been created

Storing Data Using Tuples

```
>>> canada = ['Canada', 76.5]
>>> usa = ['United States', 75.5]
>>> mexico = ['Mexico', 72.0]
>>> life = (canada, usa, mexico)
```

That builds this memory model:



You can not change the reference id3 to id6, i.e. :
Life[0] = life[1] is not possible, but you may change the items ex: 76.5

Assigning to Multiple Variables Using Tuples

You can assign to multiple variables at the same time:

```
>>> (x, y) = (10, 20)
>>> x          ?
>>> y          ?
```

multiple assignment will work with lists and sets as well. Python will happily pull apart information out of any collection:

```
>>> [[w, x], [[y], z]] = [{10, 20}, [(30,), 40]]
>>> w          ?
>>> x          ?
>>> y          ?
>>> z          ?
```

Storing Data Using Dictionaries

What is a dictionary

Dictionaries – like lists – are collections of objects. Unlike lists, dictionaries are *unordered* collections.

They are not indexed by sequential numbers, but by *keys*:

Ex:

```
>>> mydict = {"apples": 42, "oranges": 999}
 >>> mydict['oranges']                ?
```

Valid keys

The **keys** of a dictionary can be any kind of **immutable** type, which includes: strings, numbers, and tuples:

EX:

```
mydict = {"hello": "world",
           0: "a",
           1:  "b",
          "2": "not a number" }
>>>print (mydict['hello'])                    ?
```

Looping Over Dictionaries

Like the other collections you've seen, you can loop over dictionaries. The
general form of a for loop over a dictionary is as follows:

for *«variable»* **in** *«dictionary»*:
«block»

For dictionaries, the loop variable is assigned each key from the dictionary

Ex:

```
mydict = {'name': 'John', 'Age':54, 'city': 'Ottawa', 'Faculty': 'SITE'}
for x in mydict:
    print(x, ':  ',mydict[x])
```

Dictionary Operations

- Like lists, tuples, and sets, dictionaries are objects.
- Each object has operation and methods

Ex:

```
mydict = {'name': 'John', 'Age':54, 'city': 'Ottawa', 'Faculty': 'SITE'}
>>>mydict.keys()                    ?
>>>mydict.values()                  ?
```

loop over the keys and values in a dictionary:

```
for key, value in dictionary.items():
# Do something with the key and value
```

Ex:

```
>>> scientist_to_birthdate = {'Newton' : 1642, 'Darwin' : 1809, 'Turing' : 1912}
>>> for scientist, birthdate in scientist_to_birthdate.items():
... print(scientist, 'was born in', birthdate)          ?
```

Dictionary Example

Let us consider this file

canada goose
canada goose
long-tailed jaeger
canada goose
snow goose
canada goose
long-tailed jaeger
canada goose
northern fulmar

- Code to find how often each species was seen.
- uses a list of lists, each inner list has two items.
- index 0 contains the species, item at index 1 contains the occurrence
- For each line, we iterate over the list, looking for the species on that line.
- If we find that the species occurs in the list, we add one to the occurrence

Dictionary Example

Now let us see how can we solve the same problem using dictionary

To do this:

- Create a dictionary that is initially empty.
- Check the read observation from the file if repeated i.e
- If the bird is already a key in our dictionary.
- If it is, we add 1 to the value associated with it.
- If it isn't, we add the bird as a key to the dictionary with the value 1.

Here is the program that does this:

Dictionary Example

```
>>> observations_file = open('observations.txt')
>>> bird_to_observations = {}
>>> for line in observations_file:
... bird = line.strip()
... if bird in bird_to_observations:
... bird_to_observations[bird] = bird_to_observations[bird] + 1
... else:
... bird_to_observations[bird] = 1
...
>>> observations_file.close()
>>>
>>> # Print each bird and the number of times it was seen.
... for bird, observations in bird_to_observations.items():
... print(bird, observations)
```

Comparing Collections

Collection	Mutable?	Ordered?	Use When...
str	No	Yes	You want to keep track of text.
list	Yes	Yes	You want to keep track of an ordered sequence that you want to update.
tuple	No	Yes	You want to build an ordered sequence that you know won't change or that you want to use as a key in a dictionary or as a value in a set.
set	Yes	No	You want to keep track of values, but order doesn't matter, and you don't want to keep duplicates. The values must be immutable.
dictionary	Yes	No	You want to keep a mapping of keys to values. The keys must be immutable.

Summery

In these lecture you learned the following:

- Sets are used in Python to store unordered collections of unique values.
- They support the same operations as sets in mathematics.
- Tuples are another kind of Python sequence. Tuples are ordered sequences like lists, except they are immutable.
- Dictionaries are used to store unordered collections of key/value pairs.
- The keys must be immutable, but the values need not be.
- Dictionaries is much faster in searching than the other data collection