Objectives

At the end of this unit you should be able to

• Store and access data in a tuple, set, frozenset or dictionary
• Modify data in a set or dictionary
• Identify and explain the difference between an alias and a copy of a set or dictionary
• Read, trace, summarize and write Python code which can manipulate unordered data in a set
• Read, trace, summarize and write Python code which can manipulate (key, value) pairs in a dictionary
• List and recreate some common design patterns for manipulating data in sets and dictionaries
• List the properties and features of each of Python's collection classes
Collections of Data

• So far:
  – Lists: mutable, ordered, indexed collection of mutable objects; indexes are always integers 0 to \(\text{len}(\text{collection}) - 1\)
  – Strings: immutable, ordered, indexed collection of characters; indexes are always integers 0 to \(\text{len}(\text{collection}) - 1\)

• Other built-in Python collections
  – Tuples: immutable lists of mutable data
  – Sets: mutable, unordered, unindexed collection of distinct immutable data
  – Frozensets: immutable sets
  – Dictionaries: mutable, unordered, indexed collection of data; indexes can be (almost) anything

• All built-in Python collections allow heterogeneous data (except strings)
  – The type of one data element does not constrain the type of another
Tuples: An Immutable Sequence of Data

- A tuple is essentially a list that cannot be modified
  - Created by comma separated entries without []
  - By convention enclosed in ( ) to make them easier to read
  - Special cases: Empty tuple is (), one element (5,)
- Tuples can be indexed, sliced and looped over
  - Use same indexing / slicing notation as lists or strings
- Useful in immutable contexts
  - If value is not supposed to mutate, use a tuple to guarantee that it will not mutate
- Values referenced by the tuple may be mutable (eg: a tuple may contain a list), but the references themselves are not
- Often used to assign multiple variables at once
  - Swap variables:
    \[ v1, v2 = v2, v1 \]
- Also used in functions which need to return more than one value
- We can perform reduce, map and filter actions on tuples
Sets

• A set is a mutable collection of immutable items without duplicates
  – set items are not in any order; i.e. there is no first, second, …, last item
  – each item appears in the set at most once
  – items can be added or deleted
  – Set items have no index
• Set items must be hashable
  – For this class, this essentially means items cannot be mutable
  – Therefore, we cannot add Lists or Dictionaries to Sets
• We typically use sets in the cases in which the collection of data has to be duplicate free
• A sets normally starts as empty and grows by adding an element at a time
• An empty set can be defined as
  \[ s = \text{set}() \]
• A set with given values can be defined as
  \[ s1 = \{1,2,3,4,5\} \text{ or } s1 = \text{set}((1,2,3,4,5)) \text{ or } s1 = \text{set}([1,2,3,4,5]) \]
Set Methods and Operations

• To add an element to a set $s$ we use:
  $$s.add(item)$$
  – if an equal item is already in the set the new item is not added

• To remove an item from a set:
  $$s.remove(item)$$

• To check if an item is in a set:
  $$item \in s$$

• To empty a set:
  $$s.clear()$$

• Typical operations on sets:
  – intersection: returns a set with the common elements
    $$s1.intersection(s2)$$ or $$s1 \& s2$$
  – union: returns a set with all the elements of the two sets
    $$s1.union(s2)$$ or $$s1 | s2$$
  – difference: returns a set with the elements in the first set and not in the second
    $$s1.difference(s2)$$ or $$s1 - s2$$
Set Methods and Operations (cont')

• Python collection functions will also work on sets (eg: `max()`)
• Key property: very efficient `in` operator
• Order of elements is not maintained by Python
  – Display (eg: `print()`) or `for` loop generates elements in an arbitrary order
  – Reordering occurs as a side-effect of efficient `in` operator
  – If order matters, convert to a list or tuple
• If `s1` and `s2` are sets
  ```python
  s1 == s2
  ```
  returns `True` if `s1` and `s2` contain the same elements. Returns `False` otherwise.
Design Patterns for Sets

• Can apply map, filter and/or reduce patterns to a set
  – For example, printing a set:
    ```python
    v = { 'a', 'e', 'i', 'o', 'u', 'y' }
    for c in v:
        print(c)
    ```

• Additional patterns that use efficiencies of sets
  – Test for elements have already been seen
    ```python
    s = set()
    while True:
        n = input('Enter a new number:')
        if n in s:
            print('You entered that before!')
            break
        else:
            s.add(n)
    ```
  – Remove duplicate elements

• Example: `towns_set.py`
Example using Sets

• Suppose we want to keep a list of the towns from which the UBC students come this year. When students registered we created a file towns.txt that contains the student id and the town for each student. We want to print out the towns without repeating them.

• A solution:

```python
infile = open("towns.txt", "r")
towns = set()
for line in infile :
    words = line.split()
    towns.add(words[1])

print "UBC Students come from the following towns:"
for town in towns :
    print town
```

<table>
<thead>
<tr>
<th>towns.txt</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12312314</td>
<td>Vancouver</td>
</tr>
<tr>
<td>23433444</td>
<td>Calgary</td>
</tr>
<tr>
<td>23444555</td>
<td>Vancouver</td>
</tr>
<tr>
<td>21456345</td>
<td>Shanghai</td>
</tr>
<tr>
<td>43555663</td>
<td>Mumbai</td>
</tr>
<tr>
<td>34566667</td>
<td>Vancouver</td>
</tr>
<tr>
<td>56768236</td>
<td>Calgary</td>
</tr>
<tr>
<td>94356002</td>
<td>Calgary</td>
</tr>
<tr>
<td>73390234</td>
<td>Shanghai</td>
</tr>
<tr>
<td>89345672</td>
<td>Shanghai</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Summary of Facts about Sets and Frozensets

- Sets are used to store an unordered collection of distinct data
  - Each data item must be immutable
  - Supports common set operations: union, intersection, difference, element of (in), etc.
  - Data a and b are distinct if a is b evaluates to false
- A set is mutable
- A **frozenset** is immutable
- A frozenset is created by
  
  ```python
  frozenset(collection)
  ```
  where collection is a list, a tuple, or another set
- A frozenset can be an element of another set or frozenset
- There is no shorthand notation for creating sets or frozensets
Dictionaries

• Suppose we want to extend the last problem to keep track of how many UBC students come from each town
  – we need to keep pairs (town, count) not single values
  – a set will not work in this case
• The right structure for that is a dictionary
• A dictionary is a collection of pairs of the form (key, value), i.e. they associate a value with a key
  – phone book: key ==> name, value ==> phone number
  – address book: key ==> name, value ==> email address
• Each key appears at most once (is unique) but a value can be associated with many keys
• Keys are immutable, but the values are not
• You can view a dictionary as a table with keys in column 1 and values in column 2
  – You may look up keys in column 1 and find out the value in column 2, but not vice-versa
Creating a Dictionary

• Usually we start with an empty dictionary:
  ```python
country_codes = {}
```
  or with some key/value pairs:
  ```python
country_codes = { 'Canada':'CAN', 'China':'CNN', 'Russia':'RUS'}
```

• Can add a new pair:
  ```python
country_codes['United States'] = 'USA'
```
  it associates the value 'USA' with the key 'United States' in `country_codes`

• Can change the value for a key:
  ```python
country_codes['China'] = 'CHN'
```

• Can retrieve an entry:
  ```python
country_codes['Canada'] will return 'CAN'
country_codes.get('Canada') will also return 'CAN'
```

• Can check if a key is in a dictionary:
  ```python
'Canada' in country_codes will return True
```

• Can remove an entry (key/value):
  ```python
del country_codes['United States'] removes the entry for United States
```
Dictionary Methods

Dictionaries are objects and have their own methods. Their most important methods are:

- **dictionary.clear()**
  - empties the dictionary

- **dictionary.keys()**
  - returns the dictionary's keys as a list; this list has no duplicates as keys are unique

- **dictionary.items()**
  - returns a list of (key, value) pairs with all the items in the dictionary

- **dictionary.values()**
  - returns the dictionary's values as a list; the list may contain duplicates

- **dictionary.update(other_dictionary)**
  - adds to dictionary the pairs from the other dictionary
Looping Over a Dictionary

• We can access all the items of a dictionary by looping over its keys:

```python
for k in country_codes:
    print(k, ":", country_codes[k])
```

or

```python
for k in country_codes.keys():
    print(k, ":", country_codes[k])
```

• We can access all the items of a dictionary by looping over its items:

```python
for (k,v) in country_codes.items():
    print(k, ":", v)
```
Example

Here is a program that reads the file towns.txt and prints a list of the towns in alphabetical order and the number of students that come from each town:

```python
infile = open("towns.txt", "r")

# Create a dictionary with towns and number
# of students from them
towns = {}
for line in infile:
    words = line.split()
    town = words[1]
    if town in towns:
        towns[town] += 1
    else:
        towns[town] = 1

# Print towns and their count in alpha order
print "UBC Students come from the following towns:",
for town in sorted(towns):
    print town, ":", towns[town]
```

Note: The built-in function `sorted()` gets the dictionary keys and sorts them...
Summary of Facts About Dictionaries

- Any immutable data can be a key
  - For example: Booleans, any number type, strings, ...
- Any data can be a value
  - For example: Other dictionaries, files, numbers, strings, ...
- Dictionaries are mutable
  - The value associated with a key can change, but not the key
  - Keys can be added or deleted
- Many different mechanisms to access dictionary data
  - Get list of keys, list of values, loop over keys, test for key, ...
- Dictionary is not ordered
  - If you print a dictionary or iterate over a dictionary, Python will pick some arbitrary ordering (potentially different each time)
  - If you want the keys and/or values in a particular order, you need to create that ordering (eg: get the list of keys and sort it)
Summary

• Tuples are immutable lists
• Sets are mutable, unordered, unindexed collections of distinct heterogenous immutable data
• Frozensets are immutable sets
• Dictionaries are mutable, unordered collections of key-value pairs
  – The immutable, distinct keys serve as indexes
  – The values can be any data
• Sets and dictionary keys have very efficient tests for inclusion
• Ordering of sets and dictionary keys is not maintained
• Braces {} are used to create sets and dictionaries
• Brackets [ ] are used to access or modify values in a dictionary (the key is used as the index inside the brackets)
• Python has many functions, methods and operators which can create, examine and modify collections