CPSC 301: Computing in the Life Sciences
Lecture Notes 12:
Tuples, Sets and Dictionaries

2017-03-14

Jessica Dawson
jqdawson@cs.ubc.ca
http://www.cs.ubc.ca/~jqdawson

University of British Columbia
Department of Computer Science

2016 W2
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 \( \text{len(collection)} - 1 \)
  – Strings: immutable, ordered, indexed collection of characters; indexes are always integers 0 to \( \text{len(collection)} - 1 \)

• 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
  \[ s1 = \{1,2,3,4,5\} \quad \text{or} \quad s1 = \text{set}((1,2,3,4,5)) \quad \text{or} \quad s1 = \text{set}([1,2,3,4,5]) \]
Set Methods and Operations

• To add an element to a set s we use:
  ```python
  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:
  ```python
  s.remove(item)
  ```
• To check if an item is in a set:
  ```python
  item in s
  ```
• To empty a set:
  ```python
  s.clear()
  ```
• Typical operations on sets (see others in Table 13 of text)
  – intersection: returns a set with the common elements
    ```python
    s1.intersection(s2) or s1 & s2
    ```
  – union: returns a set with all the elements of the two sets
    ```python
    s1.union(s2) or s1 | s2
    ```
  – difference: returns a set with the elements in the first set and not in the second
    ```python
    s1.difference(s2) or s1 - s2
    ```
Set Methods and Operations (cont')

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

Notes 12: Sets & Dictionaries