Lab 8

S2 2017

Lab 8

Note: If you do not have time to finish all exercises (in particular, the programming problems) during the lab time, you should continue working on them later. You can do this at home (if you have a computer with python set up), in the CSIT lab rooms outside teaching hours, or on one of the computers available in the university libraries or other teaching spaces.

If you have any questions about or difficulties with any of the material in this lab, or any of the material covered in the course so far, ask your tutor for help during the lab.

Objectives

The purpose of this week's lab is to:

- Practice working with dictionaries and sets.

Dictionaries and sets

The `dict` type in python implements a *mapping*, which is also, in computer science, known as a *dictionary*, or sometimes an *associative container type*.

- A dictionary stores *key-value pairs*. Each key that is stored in the dictionary has exactly one associated value. The same value can be associated with several keys, but a key cannot have more (or less) than one value.
- Dictionaries are mutable objects; they can be modified by adding and removing keys, and replacing the value associated with a key.
- Keys must be immutable values (such as integers, strings or tuples). Keys in a dictionary do not all have to be of the same type. However, mixing key types in a dictionary will give rise to some limitations; for example, you will not be able to sort the keys.

- The values stored in a dictionary can be of any type, including mutable objects.
- Given a key, it is easy to find out if the key is stored in the dictionary, and if so what is its value. The opposite - finding the key given a value - is not so easy (in computational terms).
- A dictionary is a collection, but it is NOT a sequence.

  (Contrast this with lists and tuples, which are sequences. An implication of this is that saying "the third element" of a dictionary has no meaning. As we will see in the exercises below, the order in which you create the elements of a dictionary is not necessarily the order in which they show up when enumerated. We can "index" the elements of the dictionary, but we do this using the key.)
- The operators `[]`, `len(.)` and `in` can be applied to dictionaries. Note that `x in adict` is `True` if `x` appears as a key in the dictionary; the operator does not check if `x` appears as a value.
- Dictionaries have some additional methods. The most important are `items()`, `keys()` and `values()`, which allow iterating over the dictionary content.

The `set` type in python implements the mathematical notion of a *set*. A set is an unordered collection of values (called the elements or members of the set), without duplicates.

- A python `set` can contain objects of any immutable type, including a mixture of different types. Thus, lists cannot be elements of a set, but tuples can.
- There is only one copy of any element in a set. If you add an element that appears already, it is not added again.
- There is no order to the elements of the set. The `set` type is iterable, but it is not a sequence (so it is not indexable).
- The `len()` and `in` operators can be applied to sets. The "length" of a set is the number of elements in it.
- Sets are mutable: methods `set.add(element)` and `set.remove(element)` can be used to modify a set.
- The binary operators `&` (intersection), `|` (union), `-` (difference) and `^` (symmetric difference) can be applied to set; these return their result as a new set, without modifying the operands.

Exercise 0

You can create a dictionary by writing a comma-separated sequence of `key : value` pairs within curly brackets, `{` and `}`. You can similarly create a set by writing a comma-separated sequence of elements (without the `:` and associated value) in curly brackets. However, an empty pair of curly brackets creates an empty dictionary, not an empty set. Try the following:

```
In [1]: a = { 0 : 'none', 1 : 'one', 2 : 'two', 3 : 'many' }

In [2]: type(a)

Out [2]: ...

In [3]: b = { 'one', 'two', 'many' }

In [4]: type(b)

Out [4]: ...

In [5]: c = { }

In [6]: type(c)

Out [6]: ...
```

Values stored in a dictionary are accessed by writing the key in square brackets. What is the output of the following?

```
In [7]: a[3] + " hundred " + a[1] + " ten " + a[2]

Out [7]: ...

In [8]: a[7] + " ten " + a[3]

Out [8]: ...
```

Trying to access the value of a key that is not in the dictionary causes an error.

Dictionaries are mutable: you can add keys, change the value associated with a key already in the dictionary, and remove keys. Changing the value associated with a dictionary is done by assigning to the dictionary indexed with the key using the same syntax. Adding keys to a dictionary is done by simply assigning a value to the new key:

```
In [9]: a
Out [9]: ...

In [10]: a[3] = 'three'

In [11]: a[7] = 'a lot of'

In [12]: a
Out [12]: ...
```

Retry the two string expressions (inputs 7 and 8) above with the dictionary after reassigning the keys `3` and `7`.

Exercise 1: Creating dictionaries and sets.

Writing every key-value pair in a dictionary explicitly will quickly get tedious, so let's look at ways to create dictionaries programmatically (which is how you will use them most of the time anyway).

You can create a dictionary from a list (or any sequence) of keys and a value expression, much like you can create a list using list comprehension. For example, copy the following statement into the python shell:

```
wordlist = ['test', 'your', 'function', 'with', 'a', 'diverse',
 'range', 'of', 'examples', 'your', 'tests', 'should', 'cover',
 'all', 'cases', 'for', 'example', 'test', 'words', 'beginning',
 'with', 'a', 'vowel', 'and', 'word', 'beginning', 'with', 'a',
 'consonant', 'pay', 'particular', 'attention', 'to', 'edge',
 'cases', 'for', 'example', 'what', 'happens', 'if', 'the',
 'word', 'consists', 'of', 'just', 'one', 'vowel', 'like',
 'a', 'what', 'happens', 'if', 'the', 'string', 'is', 'empty']
```

This sets `wordlist` to a list of strings. Now, the expression

```
In [1]: wordlen = { word : len(word) for word in wordlist }
```

creates a dictionary, called `wordlen` in which every word in the list is mapped to its length. To check that it has worked, print the dictionary, and look up some sample words.

But wait! The list contains some words multiple times (for example, both "for" and "example" appear twice). Haven't we said that in a dictionary a key can only appear once? When you create a dictionary from an iterable (such as a sequence), key-value pairs are assigned in order. For example,

```
In [2]: { 'a' : 1, 'b' : 2, 'a' : 3 }

Out [2]: {'a': 3, 'b': 2}
```

That is, the second occurrence of the `'a'` in a pair in the list overwrites the first.

In a similar way, you can create a set from any iterable collection of values (as long as all values are immutable). For example,

```
In [3]: wordset = set(wordlist)
```

creates a set of strings. Here also, duplicates in the list are ignored.

A very common use of a dictionary is to count things. (An example of this was shown in last week's lecture.) The following function (from section "Dictionary as a Collection of Counters", Chapter 11, in Downey's book), creates a dictionary that maps values in a sequence (string, list, etc) to the number of times they appear:

```
def histogram(seq):
    count = dict()
    for elem in seq:
        if elem not in count:
            count[elem] = 1
        else:
            count[elem] += 1
    return count
```

Copy it into a python file and run the file so that you can test the function. You can test it on strings,

```
In [1]: histogram("neverending")
Out [1]: ...

In [2]: histogram("mississippi")
Out [2]: ...
```

on lists of numbers,

```
In [3]: histogram([2, 1, 4, 6, 5, 5, 3, 4, 4, 6])
Out [3]: ...
```

on lists of strings,

```
In [4]: histogram(wordlist)

Out [4]: ...
```

and other sequences. What happens if you try

```
In [5]: histogram([[1,2], [2,1], [1,2], [1], [2]])
```

Exercise 3: Iterating and type conversions

The python `dict` type provides three methods that return an iterable view into the contents of a dictionary: `keys()` enumerates the keys stored in the dictionary (without their associated values), `values()` enumerates the values (without the keys), and `items()` enumerates the key-value pairs. What this means is that the value returned by these methods is something that you can iterate over using a `for` loop, but they are not copies of the values in the dictionary. For example, running the following python code creates a dictionary `numbers` and prints the key-value pairs in it, one per line:

```
numbers = { 0 : 'none', 1 : 'one', 2 : 'two', 3 : 'many', 4 : 'many',
            5 : 'many', 6 : 'a few more than many', 7 : 'a lot of',
            8 : 'really many', 9 : 'too many to count' }

for key in numbers.keys():
    print(key, ':', numbers[key])
```

Another way to do the same thing is with this loop:

```
for item in numbers.items():
    key, value = item
    print(key, ':', value)
```

The expression `key, value = item` is a short-hand for writing

```
key = item[0]
value = item[1]
```

but also checks that `item` is a sequence of length exactly two (as it will be in this case). However, what happens if you change the loop to the following?

```
for item in numbers.items():
    key, value = item
    print(key, ':', value)
    numbers[key + 1] = value
```

The assignment modifies the dictionary as the loop iterates through it. The assignment to key `9 + 1` adds a new key to the dictionary, which makes the current enumeration invalid and should result in a runtime error.

Most of python's sequence types can be created directly from any kind of iterable value. Thus, for example, the following are all valid:

```
key_list = list(numbers.keys())
value_list = list(numbers.values())
item_list = list(numbers.items())
key_tuple = tuple(numbers.keys())
value_tuple = tuple(numbers.values())
item_tuple = tuple(numbers.items())
key_set = set(numbers.keys())
value_set = set(numbers.values())
item_set = set(numbers.items())
```

Check the type and content of each of these values; are they what you expect?

Exercise 4: Inverting a dictionary.

A dictionary stores one value for each key, and we can easily look up the value given the key. But finding the key that maps to a given value ("reverse look-up") is more complicated: it requires a loop over the dictionary items (key-value pairs). Furthermore, there can be more than one key that maps to the same value, so the reverse look-up must return a sequence (or a set). To speed up the process, we can build an inverse dictionary, which maps values in the original dictionary to the keys that they were associated with.

1. Write a function, `invert_dictionary`, that takes a dictionary, call it `d`, and returns a new dictionary, `inverse_d`, such that `inverse_d[x] == y` if `d[y] == x`.

   This is only possible if every key in `d` has a different value. If that is not the case, your function should detect it and print an error message.

   Test your function on some of the dictionaries you created in the last exercise. (You may find most of them cannot be inverted, because several keys have the same value.)

2. Change the function so that instead of storing only one key for each value, it stores the set of keys that map to the value in the original dictionary.

   For example, the dictionary returned by `histogram("mississippi")` is

   ```
   {'i': 4, 's': 4, 'm': 1, 'p': 2}
   ```

   Inverting this should return the dictionary

   ```
   {4: {'i', 's'}, 1: {'m'}, 2: {'p'}}
   ```

   (The order may differ, of course.)

If you get stuck, there is a solution to this exercise in section "Dictionaries and Lists", Chapter 11, in Downey's book.

Programming problems

Note: We don't expect everyone to finish all these problems during the lab time. If you do not have time to finish these programming problems in the lab, you should continue working on them later (at home, in the CSIT labs after teaching hours, or on one of the computers available in the university libraries or other teaching spaces).

Wordplay

Here is a file that contains 113,809 words, one word per line: wordlist.txt. Using this list, write programs to solve the following problems.

Note: In every case, you should read the file only once and store the data that you need in a suitable data structure.

- What are the 10 (20, 50, etc) most frequent word lengths?
- Find a word that has three consecutive double letters. (Exercise 9.7 from Chapter 9 in Downey's book.) To illustrate, the word "committee", "c-o-m-m-i-t-t-e-e", almost qualifies, except for the 'i'. (In other words, if "commttee" was a word, it would be an answer to this question).
- Counting bi-grams.

  A bi-gram is a pair of consecutive letters in a word. For example, the word "reverend" contains the bi-grams "re", "ev", "ve", "er", "re" (again), "en" and "nd". Bi-gram frequency (or, more generally, n-gram frequency) is used for various kinds of statistical analysis of text, for example in automatic document classification. Questions:

  - What are the 10 (20, 50, etc) most common bi-grams across the entire word list?
  - Using the letters of the English alphabet, there are `26 ** 2` possible bi-grams. How many (and which!) of these do not appear in any word?
  - How many words contain repeated bi-grams?
  - What is the highest number of repetions of any bi-gram in a word, and which words have that number of repetitions?

Word frequency

For a larger example of counting the frequency of items, we can consider the frequency of different words in text. To compute this, we need some (larger) examples of text. Project Gutenberg is an open source effort to digitise and make available on the web books whose copyright has expired. It currently has over 50,000 books.

You can browse the book catalog or by categories and download books. Make sure you select the "Plain Text UTF-8" format when you download.

As an alternative to downloading and reading text files, you can, with a relatively small change, make your program read the text directly from the web:

```
from urllib.request import urlopen
fileobj = urlopen("http://www.gutenberg.org/ebooks/42671.txt.utf-8")
for byteseq in fileobj:
    line = byteseq.decode()
    # process line of text
fileobj.close()
```

The two differences are that (1) you use `urlopen` instead of `open` to create the file object, and (2), each line read is not a string but a byte sequence, which must be decoded to produce a string. After this, however, you can treat the string just the same as you would if you had read it from a text file.

To count the frequency of words in the text, you need to split lines of text into words, removing whitespace and punctuation characters. Be careful with certain non-letter characters. For example, "haven't" should (arguably) be treated as one word, including the apostrophe, while for a word or phrase that appears

in single quotation marks, like "'so she said'", the same character (') should be removed. Look up the documentation of the `str.strip` method; this method can do more than just remove whitespace.

Word frequency can be used to compare different texts, or different authors. For example, are the most frequent words in books by Jane Austen the same as in 1950's pulp scifi? How many different words do different authors use? Does any of the texts use words that do not appear in the wordlist you used for the previous programming problem?

Permutations

A permutation is a mapping from a set of elements to itself, such that no two elements are mapped to the same value. A permutation (on a finite set) can be represented with a dictionary. For example, here is one

```
p1 = { 'alice' : 'carol', 'bob' : 'bob', 'carol' : 'eve',
       'dave' : 'dave', 'eve' : 'alice' }
```

We can think of it as musical chairs: in permutation `p1`, Alice moves to Carol's chair, Bob stays in his own chair, Carol moves to Eve's chair, and so on.

A *closed set* in a permutation is a subset of the elements such that the permutation maps all elements in the subset to elements in the subset. For example, in the permutation above Bob is a closed set by himself, as is Dave, while Alice, Carol and Eve form the last closed set. (Every element must belong to exactly one closed set.) Here is another example of a permutation over the same set of elements:

```
p2 = { 'alice' : 'bob', 'bob' : 'carol', 'carol' : 'dave',
       'dave' : 'eve', 'eve' : 'alice' }
```

This permutation has just one closed set, comprising all the elements.

Write a function that takes as argument a dictionary representing a permutation, and returns the list of closed sets of the permutation. (If you want, you can also check that the contents of the dictionary is actually a permutation, i.e., that the set of values is the same as the set of keys.) To test your function, create as many permutations of the set of five names as you can (or write code to generate all possible permutations).