ECM1408 Programming for Science
Continuous Assessment 3

Date set: Friday 28th November, 2014
Hand-in date: **12:00 Friday 12th December, 2014**

This continuous assessment (CA) comprises 35% of the overall module assessment.

This is an **individual** exercise and your attention is drawn to the College and University guidelines on collaboration and plagiarism, which are available from the College website.

Note that both paper (BART) and electronic submissions are required.

This CA tests your knowledge of the programming in Python that we have covered so far and particularly the writing and testing of slightly more complex programs, involving dictionaries, exceptions and recursion.

Make sure that you lay your code out so that it is readable and you comment the code appropriately.

Exercises

1 Alternades

Quoting from Wikipedia,[1] an *alternade* is a word in which its letters, taken alternatively in a strict sequence, and used in the same order as the original word, make up at least two other words. All letters must be used, but the smaller words are not necessarily of the same length. For example, a word with seven letters where every second letter is used will produce a four-letter word and a three-letter word. For example:

- SCHOOLED: makes SHOE and COLD
- WAIST: makes WIT and AS

A variant is to take every third letter to make three words. For example:

- LACERATED: the 1st, 4th and 7th letters make LET; the 2nd, 5th and 8th letters make ARE; and the 3rd, 6th and 9th letters make CAD.

Write a program, `alternade.py`, which will find and print all the alternades of words longer than four letters in a given file of words. Your program should be run as follows:

```
$ python alternade.py words-file degree
```

where `words-file` is the name of the file containing possible words, one per line and the `degree` specified whether every second, third, etc letter is to be used. If the words file is not specified, then your program should use the file `words.txt` (available from the study resources site) and if the degree is not specified it should default to 2. The printed output from your program should be one line per word of the form:

```
SCHOOLED: makes SHOE and COLD
```

[1] `http://en.wikipedia.org/wiki/Alternade`

The design of the program is up to you, but you should note that a solution based on using lists is likely to be inefficient and a solution using dictionaries is to be preferred.

You should submit:

- Copies of your code in the files `alternade.py`, (electronic submission).
- Hardcopies of your code in the file `alternade.py` (paper via BART).
- ~~Hardcopies of your the output of your programs showing the degree 2 and 3 alternades in the file `words.txt` (paper via BART).~~

[25 marks]

2 Dictionaries

The goal of this exercise is to implement a simple `dictionary` in Python without using the built-in dictionaries. Recall that a way of implementing dictionaries is to store a list of tuples `(h, key, value)` where `h = hash(key)` is the hash of the key. In order to allow rapid look up of the key-value pair, the list should be stored ordered by the hash values, so that binary search can be used to rapidly locate the relevant tuple. For example, if

```
>>> hash('a key')
290179560907415603
>>> hash('another key')
8547278478136085765
>>> hash(678)
678
```

the key-value pairs, (`'a key'`, `100`), (`'another key'`, `101`) and (`678`, `'value'`) would be stored in a list:

```
[ (678, 678, 'value'), (290179560907415603, 'a key', 100),
  (8547278478136085765, 'another key', 101)  ]
```

You should write a module, `dictionary.py`, that implements the following functions:

- `insert(key, value, D, hasher=hash)` – insert the (`key`,`value`) pair into the dictionary `D`.
- `get(key, D, hasher=hash)` – return the value in dictionary `D` corresponding to the given `key`. Raise a `KeyError` if the key is not in `D`.
- `pop(key, D, hasher=hash)` – return the value in dictionary `D` corresponding to the given `key` and remove the key-value pair from `D`. Raise a `KeyError` if the key is not in `D`.
- `keys(D)` – return a list of the keys in `D`.
- `values(D)` – return a list of the values in `D`.
- `items(D)` – return a list of tuples of key-value pairs in `D`.

The functions `insert`, `get` and `pop` should all take an additional argument `hasher`, which is the hash function used by your dictionary. By default this should be the system hash function `hash`. The system hash function is designed so that collisions (two values having the same hash) are unlikely, but you can arrange a poorer hash function in which collisions are likely as follows:

```
>>> def poorhash(x):
...     return hash(x)%10
```

Here there are only 10 possible hash values, so collisions are frequent:

```
>>> poorhash('dog')
7
>>> poorhash('cat')
7
>>> poorhash('table')
7
```

Your code should cope with collisions by storing all the key-value pairs with the same hash in adjacent entries in the list, for example,

```
[ (5, 'flower', 'value'), (7, 'dog', dog_value), (7, 'cat', cat_value),
  (7, 'table', table_value), (8, 'angel', angel_value) ]
```

Devise suitable test functions for each of your dictionary functions. Arrange your code so that these functions, demonstrating that each function works with the `poorhash` hash function, are run when the module run as a program (`python dictionary.py`).

Demonstrate your code, by implementing the word frequency program discussed in lectures using your dictionary module and extending it to print the 10 most frequently used words in the file `dracula.txt`.

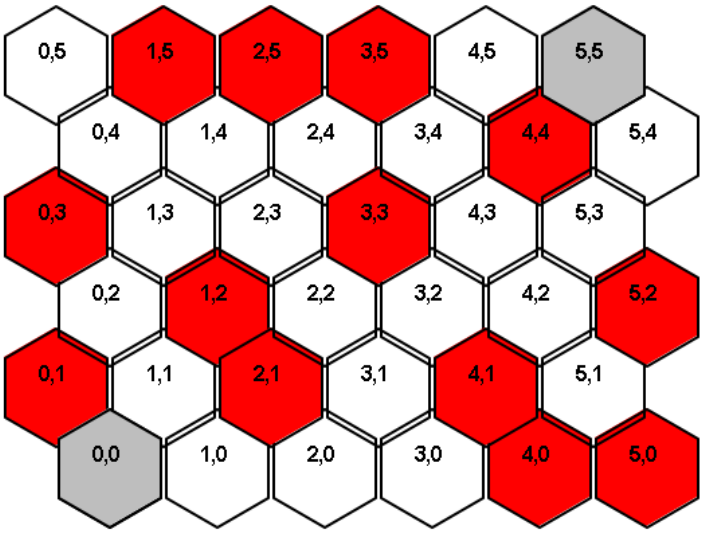
You should submit:

- Copies of your code in the files `dictionary.py` and `wordfreq.py` (electronic submission).
- Hardcopies of your code in the files `dictionary.py` and `wordfreq.py` (paper via BART).
- Hardcopies of your the output of your programs the tests of your functions ten most frequently used words in `dracula.txt` (paper via BART).

[35 marks]

3 Hexagonal mazes

The goal of this exercise is to write a program that will find paths through a maze of hexagons, such as the one below. As shown in the diagram, the maze consists of N rows and N columns of hexagons, numbered as shown. The aim is to find paths through the maze starting at the hexagon $(0, 0)$ in the bottom left hand corner and finishing at hexagon $(N-1, N-1)$ in the top right hand corner. In this maze you can move from one hexagon to any of the adjacent hexagons, except the forbidden hexagons which are coloured red.



A maze is specified by a file in the following format. The first line specifies N, the number of rows and columns and the remaining lines specify the forbidden hexagons. For example, the file `small.txt` (on the study resources site) contains:

```
6
4 0
5 0
0 1
2 1
4 1
1 2
5 2
0 3
3 3
4 4
1 5
2 5
```

describing the $N = 6$ maze shown in the diagram.

A function to draw a maze is given in the module `drawmaze.py` on the study resources size. The signature of the function is `draw(turtle, N, forbidden, scale=40)` and the arguments are:

`turtle` An `exturtle` turtle for drawing

`N` The size of the maze

`forbidden` A dictionary specifying which cells are forbidden. If the dictionary has a key for a tuple specifying the cell's coordinates, then that cell is forbidden. For example, the dictionary

```
forbidden = { (4, 0) : 1, (3, 3) : 1 }
```

specifies that the cells with coordinates $(4, 0)$ and $(3, 3)$ are forbidden. Forbidden hexagons are drawn in red.

`scale` This determines the width of the drawn hexagons. 40 is a good value.

Write a function `read(filename)` to read a maze file. Your function should return the size of the maze and a dictionary specifying the forbidden cells. Use your function and the `draw` function to draw the maze specified by `small.txt`.

Paths through the maze can be found using a recursive function. Suppose that a partial path has been constructed that reaches the cell (i, j). Then a possible new path can be constructed that consists of the original path and one of the neighbours of (i, j); this new path is passed in turn to the recursive function to extend the path. Searching terminates when a path reaches the target $(N - 1, N - 1)$ (the base case). Pseudo-code for a function that searches the maze to find the shortest paths is:

```
1: function SEARCH(path, forbidden, shortest)
2:     ▷ Recursively return the shortest path in the maze
3:     if last element of path is the target then                    ▷ Base case
4:         if path is shorter than shortest  then
5:             shortest ← path
6:             return shortest
7:     ▷ General case
8:     where ← last element of path                                  ▷ Current hexagon
9:     for neighbour in NEIGHBOURS(where) do
10:        if neighbour is not forbidden and neighbour is not in path then
11:            shortest ← SEARCH(path + neighbour, forbidden, shortest)
12:    return shortest
```

Here the function NEIGHBOURS(where) returns a list of the hexagons which are neighbours to the hexagon where. The search starts by calling the SEARCH function with the path which is just the starting hexagon.

Write a function `search(path, N, forbidden, shortest, longest)` which searches the maze to find the shortest and longest paths from $(0, 0)$ to $(N - 1, N - 1)$. Your function should return *lists* of the shortest and longest paths in case there is more than one shortest or longest path. Use your function to find and print the longest and shortest paths for the maze described by `small.txt`. Also make your program draw one of the shortest paths and one of the longest paths found. These should be on different drawings of the maze and you may want to use the `hexagon` function in `drawmaze.py`. Your code should be in a file named `maze.py`.

Hint: Remember that list are mutable and so can be modified in place by a function. You may need to copy lists that you do not want modified.

You should submit:

- A copy of your program in the file `maze.py` (electronic submission).
- Hardcopy of your program `maze.py` (paper via BART).

- Hardcopy of the output of your program showing the shortest and the longest paths for the `small.txt` maze and screenshots of the drawings of a shortest and longest path (paper via BART).

[40 marks]

[Total 100 marks]

Submitting your work

The CA requires both paper and electronic submissions.

Paper You should submit and paper copies of the code and any output for **all** the other questions to the Harrison Student Services Office by the deadline of **12:00 Friday 12th December, 2014**. Markers will not be able to give feedback if you do not submit hardcopies of your code and marks will be deducted if you fail to do so.

Paper submissions should have the BART cover sheet securely attached to the front and should be anonymous (that is, the marker should not be able to tell you are from the submission). If this is the first time you have used BART, please make sure that you understand the procedure beforehand and leave plenty of time as there are often queues close to the deadline.

Where you are asked for paper copies of the output of your code, please copy and paste the output from the terminal rather than taking a screenshot, because the screenshot is often illegible after printing. To cut and paste from a Windows Command window, highlight the text you want to paste, right click in the Command window, click on "Select all", press Enter, and past into your document (control-V or from the menu).

Electronic You should submit the files containing the code for each question via the electronic submission system at `http://empslocal.ex.ac.uk/submit/`. Make sure that your code is in files with the names specified in the questions. Then use `zip` or `rar` or `tar` to compress these into a single file, and upload this file using the submit system. You must do this by the deadline.

You will be sent an email by the submit system asking you to confirm your submission by following a link. Your submission is not confirmed until you do this. It is best to do it straightaway, but there is a few hours leeway after the deadline has passed. It is possible to unsubmit and resubmit electronic coursework — follow the instructions on the submission website.

Marking criteria

Work will be marked against the following criteria. Although it varies a bit from question to question the criteria all have approximately equal weight.

- **Does your algorithm correctly solve the problem?**
  In most of these exercises the algorithm has been described in the question, but not always in complete detail and some decisions are left to you.
- **Does the code correctly implement the algorithm?**
  Have you written correct code?

- **Is the code syntactically correct?**
  Is your program a legal Python program regardless of whether it implements the algorithm?
- **Is the code beautiful or ugly?**
  Is the implementation clear and efficient or is it unclear and inefficient? Is the code well structured? Have you made good use of functions?
- **Is the code well laid out and commented?**
  Is there a comment describing what the code does? Are the comments describing the major portions of the code or particularly tricky bits? Do functions have a docstring? Although Python insists that you use indentation to show the structure of your code, have you used space to make the code clear to human readers?

There are 10% penalties for:

- Not submitting hardcopies of your programs.
- Not naming files as instructed in the questions.