ECM1408 Programming for Science
Continuous Assessment 3

Date set: Thursday, 29th November, 2012
Hand-in date: **12:00 Thursday, 13th December 2012**

This continuous assessment (CA) comprises 30% 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 some of the more recent programming ideas that we have covered in the last few weeks. As for CA2, the description of some of these problems is quite protracted, but the problems themselves are not especially complex. Please make sure you read the entire question and think about all of it before starting to code it.

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

1 Mastermind

In this exercise you will write a version of the game "Mastermind" in which the master (your program) thinks of a number consisting of, say, 5 digits and it is the player's job to guess the number. The player enters a number and the program should then tell the player how many of the digits are correct and in the correct position, and how many are correct but not in the correct position. Thus if the hidden number is 34478, the start of the game might look like:

```
python mastermind.py
How many digits would you like to guess? 5
You have 10 guesses!

What is your next guess? 12468
Your guess has 2 digits in the correct position and 0 out of position

What is your next guess? 12448
Your guess has 2 digits in the correct position and 1 out of position
```

Here the player's responses are shown in ***blue bold italics***.

The game should ask the player how many digits to play with. Also the program should set the maximum number of guesses the player can make. If the player wins by guessing correctly in fewer tries, the program should congratulate them and report how many guesses it took. If the player loses by running out of tries, the program should tell the player what the hidden number was.

Make sure that your program is robust against unexpected input by the user. You may import code from other CAs (for example from `dice.py`).

You should submit:

- A paper copy of the code (via BART).
- A paper copy of the output of your program, showing a run in which the player wins and another in which the program wins.
- An electronic copy of your program in a file named `mastermind.py` (electronic submission).

[20 marks]

2 Inverting dictionaries

A dictionary defines a *map* between the keys and values, essentially a function which has domain equal to the set of the keys and range equal to the set of the values. Sometimes it is useful to be able to *invert* the mapping, that is construct a new dictionary whose keys are the values of the original dictionary and whose values are the keys of the original.

Write a function `invert` to invert a dictionary. Initially, assume that the values are unique (that is, each key maps to exactly one value, so that the mapping is invertible). Test `invert`. Note that if `D` is a dictionary then `invert(invert(D))` should be a dictionary identical to `D`. Put your code in a file `invert.py`.

The dictionary

```
multi = {'a' : 21, 'b' : 34, 'c' : 21, 'd' : 56, 'e', 56}
```

has the same value for more than one key, so inverting it is troublesome. Write a new function, also called `invert`, that inverts a dictionary so values that correspond to multiple keys in the original become keys (in the inverted dictionary) associated with lists of the original keys. Thus:

```
>>> multi = {'a' : 21, 'b' : 34, 'c' : 21, 'd' : 56, 'e', 56}
>>> I = invert(multi)
>>> I
{56: ['e', 'd'], 34: ['b'], 21: ['a', 'c']}
```

Your code for this function should be in a file called `minvert.py`, together with code to demonstrate that it works correctly.

Telephone directories contain the names of people as keys and their telephone numbers as values. The file `phones.txt` on the ELE page contains a list of the names of people taking this module and others, together with *fictitious* telephone numbers for them; some phone numbers are shared by more than one person (only recent British Prime Ministers share phones in this directory).

Write a program `directory.py` that reads the `phone.txt` data and then prompts the user for a surname. If there are people in the directory with that surname then your program should print their full names and telephone numbers. Sometimes (particularly if you are a spy) it's useful to be able to look up the name that corresponds to a phone number. Enhance your program so that if a number is entered it prints the name(s) of the people that use that phone.

You should submit:

- A paper copy of the code (via BART).

- A paper copy of the output of `invert` and `minvert` demonstrating that they work correctly (via BART).
- A paper copy demonstrating that your `directory.py` program works in both directions (via BART).
- An electronic copy of your programs in a files named `invert.py`, `minvert.py` and `directory.py` (electronic submission).

[20 marks]

3 String rewriting

In the last CA you were asked to iteratively rewrite strings according to transformation rules and the strings were then interpreted as instructions to a TurtleGraphics turtle.

String rewriting systems (SRSs) are naturally recursive, as each character in the initial string is rewritten according to the rules, and then the rules are used to transform each of those characters in turn.

Using the example algae string rewriting function discussed in lectures as a basis, write a recursive function `sierpinksi_print(S, n)` which will print the `n`th rewriting of the string `S`. The rules for the Sierpinski rewriting are:

Symbols `E`, `F`, `L` and `R`;

Axiom `E`;

Rules `E` → `FLELF` and `F` → `ERFRE`.

Write a second function `sierpinski(S, n, t, step=2, angle=60)` which, rather than printing the string, uses the turtle `t` to interpret the string symbols as:

`F` move forwards a fixed distance given by the `step` argument;

`E` move forwards by the same fixed distance as for `F`;

`L` turn left by a fixed angle, given by the `angle` argument;

`R` turn right by the fixed angle.

Any other symbols are ignored by the turtle. You should either use your `follow` function from the last CA or the version provided on the ELE site.

Use your function to draw the Sierpinski curve (starting with the string `E`) with a recursion depth of 12. Both these functions should be written in a file `sierpinski.py` and when python is used to run the file (`python sierpinski.py`) your program should draw the curve.

The repertoire of string rewriting systems can be increased by introducing a pair of new symbols: `[` and `]`. These symbols are just copied without transformation by the SRS. However, when the string is interpreted, a `[` means save the current location and heading of the turtle, while the matching `]` means restore the turtle to the saved location and heading.

We will use the following *fern* SRS:

Symbols `A, F`, `L` and `R`;

Axiom `F`;

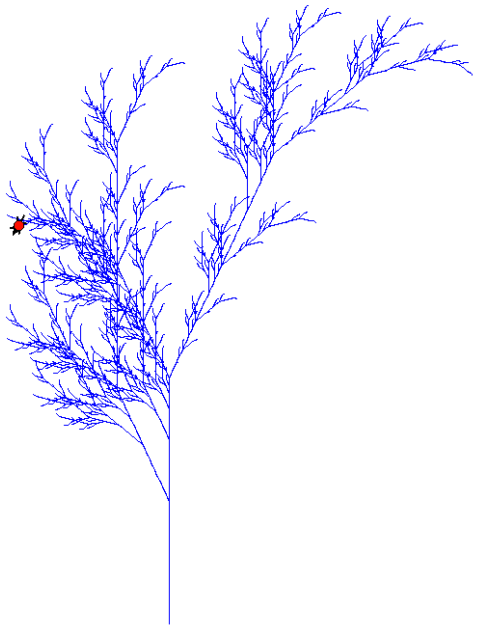
Rules `A` → `FL[[A]RA]RF[RFA]LA` and `F` → `FF`.

Enhance your `follow` function so that it has the signature `follow(t, S, stack, step=10, angle=90)`. Here `stack` is a list which you should use storing and retrieving the location and heading of the turtle when the `[` and `]` symbols are encountered. In essence, add the location and heading to the stack when a `[` is encountered and retrieve them when a `]` is found.

Write a function `fern(S, n, turtle, stack=[], step=4, angle=25)` to recursively rewrite and draw the string `S`. Draw a fern with this function, using a recursion depth of 6. With an angle of 25°, the result should be similar to the picture below.

Hints:

- Recall that you can get the location of the turtle with the `where(turtle)` function. Its heading is returned by the `get_heading(turtle)` function. The turtle can be repositioned to a new location and heading with the `goto(x, y, heading)` function which places the turtle at `(x, y)` with the given `heading`. To use the `get_heading` and `goto` functions you will need to **download a new version** of `exswampy` from the ELE site.
- Recall that `list.pop(index)` returns `list[index]` and removes that item from the list. Think about how you could use `list.append(item)` and `list.pop(-1)`.
- Note that lists are mutable variables, so that if `stack` is modified inside `follow`, the changes will affect `stack` in the function that called `follow`. This is useful, not a problem!
- You may need to move the turtle's starting position.
- Do not use global variables!



You should submit:

- A paper copy of the code (via BART).
- A paper copy of screenshots of the Sierpinski curve and the fern drawn by your program (via BART).
- An electronic copy of your programs in a files named `sierpinski.py`, `fern.py` and `follow.py` (electronic submission).

[30 marks]

4 Sets

The aim of this exercise is to write and test some functions that implement operations on sets. We will represent the sets by lists, but recall that an item can only occur once in a set, so `S = [1, 45.6, 'element', 'zero', math.pi]` is a legal set, but `T = [1, 2, 1]` is not.

Write and test a function `legalise(L)` a function that returns a list that is legal set constructed from the list `L`, which may contain repeats. Make sure your test checks the result for a list without repeated elements, with repeated elements, with a single element and an empty list.

If `A` and `B` are two sets, write functions that implement the following operations on sets:

- `union(A, B)` returns the union $A \cup B$ of the sets.
- `intersection(A, B)` returns the intersection $A \cap B$ of the sets.
- `remove(A, e)` returns A without the element e. If $e \notin A$ then this function should just return A.
- `difference(A, B)` returns the set difference $A \setminus B$ of the sets.

Write a function `test` that tests and prints demonstrations of these functions on useful test cases.

Each of these functions will be quite short. I know that Python provides a set object, but there are no marks for using it! You could verify your functions by checking them against the Python built-in set methods.

The *power set* of a set A, denoted $\mathcal{P}(A)$, is the set of all subsets of A. For example, if $A = \{a, b, c\}$ then, the subsets of A are:

$$\{\},\ \{a\},\ \{b\},\ \{c\},\ \{a, b\},\ \{a, c\},\ \{b, c\},\ \{a, b, c\}$$

where $\{\}$ means the empty set. Then

$$\mathcal{P}(A) = \{\{\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$$

If A has n elements, then the power set has 2^n elements.

A recursive algorithm to generate the power set of A can be written in terms of the function

$$F(e, T) = \{X \cup \{e\} \,|\, X \in T\}$$

This function returns the set with the element e added to each set X in T.

The algorithm then consists of the base and general cases as follows:

- If $S = \{\}$, then return $\mathcal{P}(S) = \{\{\}\}$
- Otherwise:
  - Let e be any single element of S
  - Let $T = S \setminus \{e\}$
  - Return $\mathcal{P}(S) = \mathcal{P}(T) \cup F(e, \mathcal{P}(T))$

For the base case the power set of an empty set is the set containing the empty set. In the general case the power set of a set is all the subsets of the set containing a particular element (e) and all the subsets of the set not containing that particular element.

Write a function `powerset(S)` that recursively finds the power set of `S`. Demonstrate your function by printing the power set of the set `[ 'alpha', 'beta', 'gamma', 'delta', 'epsilon']`.

You should submit:

- A paper copy of the code (via BART).
- A paper copy of the output of a run of your program, showing the demonstations of your functions and the power set of the set of Greek letters.
- An electronic copy of your program in a file named `sets.py` (electronic submission).

[30 marks]

[Total 100 marks]

Submitting your work

The CA requires both paper and electronic submissions.

Paper You should submit 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 Thursday, 13th December 2012**. 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 who 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 paste 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.

5 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.