ECM1408 Programming for Science
Continuous Assessment 2

Date set: Tuesday 6th November, 2012
Hand-in date: **12:00 Wednesday 21st November, 2012**

This continuous assessment (CA) comprises 25% 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 in the first few weeks of term, particularly the writing and testing of slightly more complex programs.

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

Exercises

1. **Shuffling.** It's often useful to be able to shuffle the items in a list. For example, in the next exercise it you will need to shuffle a list of integers representing a deck of cards. Pseudo-code for the Durstenfeld algorithm to shuffle the list `L` of N elements in place is:

   ```
   for i from N-1 downto 1 do:
       j := random integer with 0 <= j <= i
       exchange L[i] and L[j]
   ```

   To read more about how this works see, for example, `http://en.wikipedia.org/wiki/FisherYates_shuffle`.

   Write a function `shuffle(L)` which uses the Durstenfeld algorithm to shuffle the list `L`.

   Demonstrate your code by using it to shuffle the list of integers $1, \ldots, 20$ and printing the result. Also print the result of shuffling the list of Greek letter names:

   ```
   ["alpha", "beta", "gamma", "delta", "epsilon", "zeta", "eta",
    "theta", "iota", "kappa", "lambda", "mu"]
   ```

   Clearly, all the elements in the original list should be in the shuffled list, and the length of the shuffled and original lists should be the same. Write a function `check_shuffle` that checks that your `shuffle` function respects these conditions.

   It's a bit harder to test the *quality* of the shuffle. One way to check that the shuffle works is to define the quality of a shuffle as the fraction of times the second element of two adjacent elements is larger than the first. Thus the sorted list `[0, 1, 2, 3, 4, 5, 6]` has a quality $6/6 = 1$ because every element is greater than the previous one, whereas the list `[1, 4, 2, 3, 6, 5, 0]` has a quality $3/6 = 0.5$ because the pairs $(1, 4), (2, 3), (3, 6)$ have the second

element greater than the first, but the other pairs $(4, 2), (6, 5), (5, 0)$ do not. The list `[6, 5, 4, 3, 2, 1, 0]` has quality 0. A well-shuffled list will have a quality close to 0.5.

Write a function `quality(L)` that evaluates how well the list `L` is shuffled. Since the shuffling is random, to get a good idea of the quality we should take the average quality over several shuffles of the original list. Therefore write a function `average_quality(L, trials)` that finds the average quality of `trials` shuffles of `L`. Use this code to verify that the average quality produced by your `shuffle` function is close to 0.5.

Your code should be in a file named `shuffle.py`. Organise the code so that the functions can be imported into other programs and the average quality over 1000 trials of shuffling the list `range(100)` is printed out when the module is run as `python shuffle.py`.

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 result of shuffling a list of 20 integers and the list of Greek letter names.
- An electronic copy of your program in a file named `shuffle.py` (electronic submission).

[20 marks]

2. Patience. A variation of the card game *patience* is played as follows.[1] Starting with a shuffled deck of cards, the top two cards are placed face up so that their numbers are visible. Then, if their values sum to 11 (i.e., Ace and 10; 2 and 9; 3 and 8; 4 and 7; 5 and 6) then they are covered by two new cards drawn from the deck. If their values do not sum to 11 then a new card is drawn from the deck and placed face up to form a third visible card. If any pair of these sum to 11, then they are covered by new cards drawn from the deck; if not a new card is drawn from the deck to form a fourth visible card. Play continues this way until either the player wins by having no more cards in the deck or their are more than nine piles of visible cards. In addition to the "two cards summing to 11" rule, if a Jack and a Queen and a King are **all** visible then, each of them is covered by a new card from the deck.

Here is an example in which the player wins. The cards are represented by the numbers 1, ..., 13, with 1, 11, 12 and 13 meaning Ace, Jack, Queen and King respectively. The deck, written as a Python list, is:

```
[10, 4, 9, 8, 5, 1, 2, 12, 9, 11, 2, 12, 1, 3, 12, 10, 6, 13, 7, 6, 10,
4, 7, 5, 8, 2, 4, 1, 3, 9, 5, 4, 6, 9, 11, 10, 13, 11, 11, 1, 12, 3, 13,
2, 5, 13, 7, 3, 7, 6, 8, 8]
```

[1] We use the standard British deck of 52 cards in four suites: Ace (=1), 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen and King.

The game, which starts with the first two elements in the deck (10 and 4), is:

10	4								*Cards don't add to 11; start a new pile*
10	4	9							*Cards don't add to 11; start a new pile*
10	4	9	8						*Cards don't add to 11; start a new pile*
10	4	9	8	5					*Cards don't add to 11; start a new pile*
10	4	9	8	5	1				*10 and 1 add to 11; cover with new cards*
2	4	9	8	5	12				*2 and 9 add to 11; cover with new cards*
9	4	11	8	5	12				*Cards don't add to 11; start a new pile*
9	4	11	8	5	12	2			*9 and 2 add to 11; cover with new cards*
12	4	11	8	5	12	1			*Cards don't add to 11; start a new pile*
12	4	11	8	5	12	1	3		*8 and 3 add to 11*
12	4	11	12	5	12	1	10		*1 and 10 add to 11*
12	4	11	12	5	12	6	13		*J, Q, K (11, 12, 13) visible; cover with new cards*
6	4	7	12	5	12	6	10		*6 and 5 add to 11*
4	4	7	12	7	12	6	10		*4 and 7 add to 11*
5	4	8	12	7	12	6	10		*5 and 6 add to 11*
2	4	8	12	7	12	4	10		*7 and 4 add to 11*
2	1	8	12	3	12	4	10		*1 and 10 add to 11*
2	9	8	12	3	12	4	5		*2 and 9 add to 11*
4	6	8	12	3	12	4	5		*6 and 5 and to 11*
4	9	8	12	3	12	4	11		*8 and 3 add to 11*
4	9	10	12	13	12	4	11		*Q (12), K (13), J (11) visible; new cards: 11, 1 and 11*
4	9	10	11	1	12	4	11		*10 and 1 add to 11*
4	9	12	11	3	12	4	11		*Start a, 9th and last, new pile*
4	9	12	11	3	12	4	11	13	*Q, J, K visible; new cards: 5, 2, 13*
4	9	5	2	3	12	4	11	13	*9 and 2 add to 11*
4	7	5	3	3	12	4	11	13	*Q, K, K visible; new cards: 7, 6, 8*
4	7	5	3	3	6	4	7	8	*4 and 7 add to 11, but 8 is the only card left in deck*

Note that in some cases more than one pair adds to 11 or a pair adds to 11 and the Jack, Queen and King are all visible. In these cases it depends on the player (or program!) which order the rules are applied. Most games end with the player losing because she needs to add a tenth pile of cards. A short, unannotated example is:

3	2								
3	2	1							
3	2	1	5						
3	2	1	5	13					
3	2	1	5	13	5				
3	2	1	5	13	5	3			
3	2	1	5	13	5	3	2		
3	2	1	5	13	5	3	2	6	
3	2	1	7	13	5	3	2	12	
3	2	1	7	13	5	3	2	12	6

At the end there are 9 piles of cards, but no pair adds to 11 and, although the Queen and King are visible, the Jack is not.

The majority of games end with some cards left in the deck. The object of this question is to estimate the probability of winning a game. To do that you are asked to write a program that simulates many games and thus to calculate the fraction of games that are won.

I recommend that you represent the cards as the integers 1 to 13. Represent the deck of cards and the visible cards as lists of these integers.

You should write two functions `add_to_11(visible)` and `jqk(visible)` that take as their arguments the list of visible cards and return tuples of indices indicating which cards either add to 11 or are the Jack, Queen and King. If neither rules applies, return an empty tuple.

Write a function `play(deck, verbose)` that plays a single game of patience with a deck `deck`. The Boolean argument `verbose` should control whether the visible cards are printed at each stage of the game (as above). Your `play` function should return the number of cards left in the deck at the end of a game; thus 0 means the player has won and a positive number means that the player has lost. Test your function, producing hardcopy of a game in which the player wins and another in which she loses.

Write a function `many_plays(N)` which plays `N` games of patience. The function should return a list of length 53 showing the number of times a game ended with 0, ..., 52 cards left in the deck. Thus if the list is called `remaining`, and if 20 games end with 6 cards remaining, then `remaining[6] = 20`.

Write a function `histogram(x, y, width)` which draws a simple histogram of the numbers in `y` versus those in `x`. So, for example,

```
>>> x = range(6)
>>> H = [12.5, 6.4, 10, 7.6, 8, 13]
>>> histogram(x, H, width=30)
0 ******************************      12.5
1 ***************      6.4
2 ************************      10
3 ******************      7.6
4 ******************      8
5 ******************************      13
```

Here the `width` argument controls the width (in characters) of the longest bar of stars; in the above example the bar corresponding to `H[5] = 13` comprises 30 stars. Your histogram function should be in a separate file named `histogram.py`, together with a short function to demonstrate its use.

Use your `histogram` function (suitably `import`ed) to plot a histogram of the percentage of times that games ended with 0, ..., 52 cards left in the deck. That is, plot a histogram of the list returned by your `many_plays` function. Your program should be fast enough that you can choose `N = 10000` and therefore obtain a reasonable estimate for the probability of any number of cards remaining in the deck. In particular, state what the probability of winning a game is.

You should submit:

- Copies of your `patience.py` and `histogram.py` programs (electronic submission).
- Hardcopies of your `patience.py` and `histogram.py` programs (paper submission, via BART).
- Hardcopy of the output of your `patience.py` program, showing a game in which the player wins, a game in which the play loses and the histogram of the percentage of times that games ended with 0, ..., 52 cards left in the deck (paper via BART).
- Hardcopy of the output of your histogram test function (paper via BART).

Hints:

- Use your `shuffle` function from exercise 1 to shuffle the deck of cards. If you are not confident that your function works, you can use the `shuffle` provided on the ELE page. This is supplied as `shuffle.pyc`, which is the Python code *byte-compiled* to an intermediate code which can be read by the interpreter, but not easily by humans. You can import it exactly like any other module, for example:

```
>>> import exshuffle
>>> L = range(10)
>>> exshuffle.shuffle(L)
>>> L
[6, 5, 7, 1, 4, 0, 3, 8, 2, 9]
```

- If you represent the deck by a list, then you can get the top "card" on the deck with `deck.pop(0)` which returns the first item in the list and removes it from the list.
- Recall the multiplying a string by a positive integer `n` concatenates the string with itself `n` times: thus `"X"*10 = "XXXXXXXXXX"`.

[40 marks]

3. String rewriting systems. String rewriting systems (SRS) are a beguilingly simple way of producing some very complex objects. They were popularised by Astrid Lindenmayer's work on modelling biological systems and a prototypical SRS is a simple model for the growth of algae. An SRS is defined in terms of some *symbols*, a starting string or *axiom* and one or more *rules* for transforming the symbols. Given the initial string or axiom, each symbol in the axiom is transformed according to the rules; the resulting string now takes the place of the axiom and is itself transformed. For example, in Lindenmayer's algae system the elements are:

Symbols `A` and `B`

Axiom `A`

Rules `A` $\rightarrow$ `AB` and `B` $\rightarrow$ `A`

To produce longer strings the axiom is rewritten according to the first rule `A` $\rightarrow$ `AB`, to yield the string `AB`. The first symbol of this new string (`A`) is rewritten according to the first rule, while the second symbol (`B`) is rewritten according to the second rule, to yield `ABA`. These steps and the next few are shown below:

$n = 0$	`A`
$n = 1$	`AB`
$n = 2$	`ABA`
$n = 3$	`ABAAB`
$n = 4$	`ABAABABA`
$n = 5$	`ABAABABAABAAB`
$n = 6$	`ABAABABAABAABABAABABA`
$n = 7$	`ABAABABAABAABABAABABAABAABABAABAAB`

Write a function `algae(S)` which takes as its argument a Python string and returns the string rewritten according to the algae rules above. Thus if your function is given the string `"ABAAB"` it should return the string `"ABAABABA"` (c.f., $n = 3$ and $n = 4$ above). Use your function to write a program that prints the axiom and the first 10 rewritings of the algae sequences, and finds the length of the string corresponding to $n = 30$. This program should be in a file named `algae.py`.

The string rewriting systems come to life when the symbols are interpreted as instructions to a line drawing turtle. In the rest of this exercise we will interpret the symbols `E`, `F`, `L` and `R` to mean:

`F` move forwards a fixed distance, say 10 pixels;

`E` move forwards by the same fixed distance as for `F` (it will become clear why two symbols are needed);

`L` turn left by a fixed angle;

`R` turn right by a fixed angle.

Any other symbols are ignored by the turtle.

Write and test a function `follow(t, S, step=10, angle=90)` which causes the Turtle-Graphics turtle, `t`, to draw the path described by the string `S`. The arguments `step` and `angle` should specify the amount by which the turtle draws forward for an `E` or `F` symbol and the angle through which it turns for `L` and `R` symbols. You may want to speed up the rate at which the turtle draws, which can be done with the statement `world.delay = 0` after getting the `TurtleWorld`:

```
world = TurtleWorld()
world.delay = 0
```

Your `follow` function should be in a file `follow.py`

Write a function `dragon(S)` (in a file `dragon.py`) that rewrites the string `S` according to the following *Heighway dragon* SRS:[2]

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

Axiom `FA`;

Rules `A` → `ARBF` and `B` → `FALB`.

(The symbols `F`, `L` and `R` are not rewritten, but just copied, so `FA` → `F ARBF` → `F ARBF R FALB F` where the spaces are just to make it easier to read.)

Importing the `follow` function, write a program in the `dragon.py` file that uses the `dragon` function to draw the result of the rewritings of the Heighway dragon SRS. The turning angle should be 90°. Experiment with different numbers of rewritings, but beware that very long strings—that take a long time to draw—are produced by more rewritings (the string from 20 rewrites comprises 3145727 symbols). You should make a screenshot for the $n = 12$ case

[2] See `http://en.wikipedia.org/wiki/Dragon_curve` and references there for more information on the Heighway dragon.

Write a new function `sierpinski(S)` (in a file `sierpinski.py`) that rewrites `S` according to the SRS:

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

Axiom `E`;

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

Use your `follow` function to draw the result of 8 rewrites according to these rules, with `step=2` and `angle=60`. This should produce an approximation to the famous Sierpinski curve, which — like the Heighway dragon and the Koch curve — is a fractal having a dimension between 1 (a simple curve) and 2 (the plane). Fractals are extensively used in describing and modelling natural objects with structure on many length scales, such as the coastline of Great Britain, the shape of clouds, the distribution of stars in the universe, the structure of intestinal walls, plants and so on. They are also used in computer graphics to make realistic models of terrain and natural scenery. The classic reference for fractals is Mandelbrot's book, *The Fractal Geometry of Nature*, which is in the library; there is lots of recent information on the web.

You should submit:

- Copies of your code in the files `algae.py`, `follow.py`, `dragon.py` and `sierpinski.py` (electronic submission).
- Hardcopies of your code in the files `algae.py`, `follow.py`, `dragon.py` and `sierpinski.py` (paper via BART).
- Hardcopies of the output of your code: the first 10 rewritings of the algae sequences and the length of the $n = 30$ algae string; a screenshot of the test for your `follow` function; a screenshot of your Heighway dragon and a screenshot of your Sierpinski curve. (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 Wednesday 21st November, 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 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.