Programming Exercices

Series 2

Nuno D. Mendes

Exercices

1. **Euromillions** [EASY]

   Write a program that gives you a possible outcome of a Euromillions lottery.

   Hint: Check the `random` package and the `set` data type.

2. **Symbol Generator** [EASY]

   Write a program that, given a dictionary whose keys are symbols (characters) and whose values are probabilities, generates a string of a given length with each symbol occurring with the predetermined probability.

   If the keys of the dictionary happen to be strings with more than one character, ignore the remaining characters and consider only the first one. Ignore all dictionary entries whose keys are not strings or if the string is empty. If the values of the dictionary don't add up to one or are, in effect, greater than one, make the necessary adjustments. Ignore dictionary entries whose values are not non-negative real numbers.

3. **Rational numbers** [MEDIUM]

   Create a new class `Rational` representing rational numbers. Rational numbers should be represented by a quotient, so you should keep a numerator and a denominator. You should be able to perform all basic arithmetic operations with these numbers by overloading the usual operators ($+$, $-$, $*$ and $/$). You should also be able to compare rational numbers using the usual comparison operators. The string representation of a rational number should yield the irreducible version of the corresponding quotient. (e.g. a rational number specified as $\frac{4}{6}$ should be readily converted to $\frac{2}{3}$). In order to do this you can use one of the oldest algorithms known, Euclid's algorithm (300 BC), to determine the greatest common divisor between the numerator and the denominator:

```
def gcd(p,q):
    while q:
        p, q = q, p % q
    return p
```

With the class `Rational` implemented one should be able to write the following expressions:

```
p = Rational(4,6)
q = Rational(9,10)

if p < q:
    a = p + q
elif p > 2 * q:
    b = p * q
elif p > q:
    c = p / q
```

4. **Brownian motion simulation** [MEDIUM]

Brownian motion consists of the random trajectories described by minute particles immersed in or floating on the surface of a fluid. The conditional probability distribution of the position of one of these particles at time $t + dt$ is a distribution $N(p + \mu dt, \sigma^2 dt)$, if p is the position of the particule at time t. μ is called the drift velocity and σ the noise.

One way to approximate this result when $\mu = 0$ and $\sigma = 1$, is to model the trajectory of each particle in the 3D space by successive observations of discrete uniformly distributed random variables $X, Y, Z \frown \mathit{Uniform}(-1, 1)$ so that the next position of a particule is a triple of random variables $(p_x + X, p_y + Y, p_z + Z)$ given the current positions (p_x, p_y, p_z).

The goal of this exercise is to make a movie of N particles engaging on Brownian motion in a closed volume. If a particle happens to hit the wall of your volume, make it appear on the opposite side.

You can adopt the following strategy:

- Given a parameter N for the number of particles, make your program write an R script with a series of coordinate lists. These coordinate lists should be the coordinates of your particles at each time step. This R script should also generate as many PNG files as the number of time steps S you are considering (consider at least 100 time steps);

- Invoke R from your Python program to run the script you have automatically generated;
- You have now S PNG files representing various moments in time. Make sure your files are named in such a way that lexicographic order (alphabetic order) reflects chronology. You can now invoke a shell command from your program to make a movie from all your PNG files: `convert -delay DELAY *.png your_movie.mng`, where `DELAY` is the number of milliseconds each PNG image will occupy in the movie.
- You can now watch your movie using `konqueror`.

Make sure that your program cleans up after you, ie, any file you may have created must be deleted before exiting (except, of course, your movie). Moreover, every time you create a new (temporary) file, try to devise a strategy to avoid possible name conflicts (you should not accidently overwrite any pre-existing file).

5. **Sequence alignment** [MEDIUM]

Implement the Needleman-Wunsch alignment algorithm. Your program should read a FASTA file and align the first two sequences found therein. You can assume you will only find DNA sequences. In the end, your program should report the alignment to a file showing the matches and the gaps. For example:

```
Alignment score:  α
ATTTTGGgGT-----GGGGAAAAGGTTG
ATTTTGGcGTaaaaaGGGGAAAAGGTTG
```

Where α is the score of your alignment, which will depend on the way you decide to score your matches, mismatches and gaps.

6. **Pairwise shortest-paths in Graphs** [ADVANCED]

A directed graph $G = (V, E)$ is a pair where V is a set of vertices and E is a set of ordered pairs (v_1, v_2) with $v_1, v_2 \in V$. A sequence of vertices $\Gamma = v_1 v_2 \ldots v_n$ is a path in G from v_1 to v_n iff $\forall_{1 \leq i \leq n} v_i \in V$ and $\forall_{1 \leq i < n} (v_i, v_{i+1}) \in E$. The number n of vertices in the path Γ is said to be the lenght of the path and is denoted by $|\Gamma|$.

(a) Choose an appropriate representation for graphs, considering that, in general, they are sparse, ie, in most cases for any two vertices $u, v \in V$, $(u, v), (v, u) \notin E$.

(b) Given parameters n and p your program should generate a graph G with n vertices and such that for any two vertices $u, v \in V$, $(u, v) \in E$ with probability p.

(c) A second program should, given a graph $G = (V, E)$ and two vertices $u, v \in V$, determine a path Γ from u to v such that $|\Gamma| \leq |\Gamma'|$, for any other path Γ' from u to v.

For the sake of simplicity you are advised to identify vertices with integers.

Notes

1. Check with Python documentation if you have any doubts. You can find the language reference manual at http://docs.python.org/ref/ref.html and the Python Tutorial at http://docs.python.org/tut/tut.html. Remeber to use the `help` function on interactive mode, and the `pydoc` command in the command-line

2. Make sure your code is elegant and readable and that the appropriate error messages are printed every time the assumptions about program arguments are violated

3. Be lazy = be smart! Try to produce programs that require you to write the least amount of code

4. You are expected to complete this series of exercises until November, 11

5. To obtain comments, suggestions, to dissipate any doubts and to deliver your code you should write to `ndm@algos.inesc-id.pt`