DM 536 Introduction to Programming

Fall 2012 Project (Part 2)

Department of Mathematics and Computer Science
University of Southern Denmark

October 4, 2012

Introduction
The purpose of the project for DM536 is to try in practice the use of programming techniques and knowledge about the programming language Python on small but interesting examples.

There are two possible projects. You have to pick one of these. Each project consists of two parts. You may choose to do the first part of one project and the second part of the other project.

Please make sure to read this entire note before starting your work on this part of the project. Pay close attention to the sections on deadlines, deliverables, and exam rules.

Exam Rules
This second part of the project is a part of the final exam. Both parts of the project have to be passed to pass the overall project.

Thus, the project must be done individually, and no cooperation is allowed beyond what is explicitly stated in this document.

Deliverables
A short project report (at least 4 pages without appendix) has to be delivered. This report has to contain the following 7 sections:

- **front page** (course number, name, section, date of birth)
- **specification** (what the program is supposed to do)
- **design** (how the program was planned)
- **implementation** (how the program was written)
- **testing** (what tests you performed)
- **conclusion** (how satisfying the result is)
- **appendix** (complete source code)

The report has to be delivered as a single PDF file electronically using Blackboard's SDU Assignment functionality.

Deadline

October 31, 11:00

Project "Fractals and the Beauty of Nature"

Many fractals can be generated from descriptions in the Fractal Description Language (.fdl). This language describes a start state of a fractal and a set of rules how to progress to larger depths.

For example, to generate a Koch curve, we start with the initial (depth 0) state `F` signifying a forward move, i.e., a straight line. The rule for expanding to the next depth is given as a replacement rule.

```
F -> F L F R F L F
```

where `L` is a 60 degree turn to the left and `R` is a 120 degree turn to the right. That is, we replace a straight line by a straight line, a left-turn, a straight line, a sharp right-turn, a straight line, a left-turn, and a fourth and final straight line.

Thus, the state for depth 1 is `F L F R F L F`. To get to depth 2, we have to apply the rule again to all positions of the state where it is possible, i.e., we have to replace each of the four `F` by `F L F R F L F`. The result is F L F R F L F L F L F R F L F R F L F L F L F R F L F R F L F L F L F R F L F where the new sections are underlined to aid your understanding.

To get to depth 3, we would have to replace each of the 16 `F`s by the right side of the rule. For the sake of brevity, I leave this exercise to you.

Let us take a look at the file `koch.fdl` available from the project section of the course home page.

```
start F
rule F -> F L F R F L F
length 2
depth 5
cmd F fd
cmd L lt 60
cmd R rt 120
```

The first line give the start state, i.e., the state `F` for depth 0. The second line give the only rule needed for the Koch curve, i.e., to replace `F` by `F L F R F L F`. The third line specifies the length of each segment, i.e., each straight line will be 2 units long. The fourth line specifies that states should be expanded to depth 5 before drawing the fractal. Finally, the Lines 5 to 7 specify that `F` is a straight line, `L` is a 60 degree left-turn and `R` is a 120 degree right-turn.

Task 0 – Preparation

Download all the .fdl files from the course homepage (at least `dragon.fdl`, `fern.fdl`, `koch.fdl`, `sierpinski.fdl`, `sierpinski2.fdl`, `snowflake.fdl`, and `tree.fdl`).

Try to understand how these generate their respective fractals. If you are new to the fractals project, have a look at the project description for the first part to get some inspiration. While most of the fractals are known, `dragon.fdl` represents a dragon curve and `sierpinski.fdl` represents a Sierpinski curve.

Task 1 – Representing and Applying Rules
A rule consists of a single letter for the left side and a list of letters for the right side. Your task is to create a user-defined type (= Python class) "Rule" that represents such a rule. This class should have attributes for at least the left side and the right side.

You also need to write a function that applies a rule to each (matching) element of a list, i.e., that from the list `["F","L","F","R","F","L","F"]` for depth 1 of the Koch curve will generate the list `[["F","L","F","R","F","L","F"],"L",["F","L","F","R","F","L","F"],"R",["F","L","F","R","F","L","F"],"L",["F","L","F","R","F","L","F"]]`.

Task 2 – Representing and Executing Commands
A command consists of a command string (such as `fd`, `lt`, `rt`, `scale`) and a list of arguments. Your task is to create a user-defined type (= Python class) "Command" that represents a command. This class should have attributes for at least the command string and the arguments.

You also need to write a function for executing a command. This function gets as an argument a turtle object and a length. The command with command string `lt` and argument list ["60"] should execute the Python statement `lt(turtle, 60)`. The command `scale` multiplies the length with the given float. Finally, the command `nop` simply does nothing (no operation).

Task 3 – Loading a Fractal Description Language File
A fractal consists of a start state represented by a list, a list of rules, a mapping from single letters to commands, a length, and a depth.

Your task is to create a user-defined type (= Python class) "Fractal" that has at least the attributes described above. In addition, you should write a function or method that executes all commands of a given state, i.e., goes through a state list, uses the mapping from single letters to commands, and executes these.

You also need to write a function that reads an .fdl file and creates a Fractal object from it.

Task 4 – Generating Fractals
Now you are left with computing a new state from an old state. Your task is to write a (recursive or iterative) function or method that for a given start state, set of rules, and depth computes the state at that depth.

Finally, you have to put everything together such that you can load, compute, and draw a fractal for a given file. To this end, you should import the `sys` module and use as a filename the first argument passed on the command line, i.e., `sys.argv[1]`. You also need to write a function to flatten the lists of lists obtained by rule applications into a list of elements those lists.

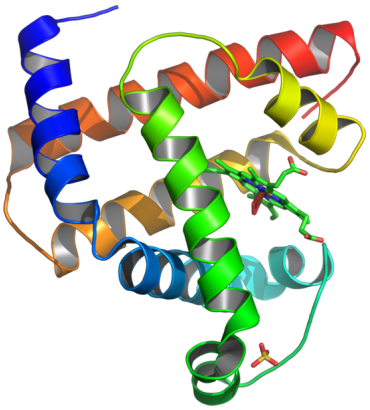
Task 5* – Support for Line Widths and Colors
The fractals look nice enough, but some colors and wider lines would make them more pretty. Your challenge task is to extend the Fractal Descripton language by the commands `color` and `width` where color gets a color name, a color code or "`random`" as an argument while width gets a float. Random colors can e.g. be generated by using format strings:
`"#%02x%02x%02x" % (randint(0,255),randint(0,255),randint(0,255))`

Here is an example for a nicer dragon curve:

```
start F X
rule X -> X R Y F
rule Y -> F X L Y
length 3
depth 13
color random
width 2.0
cmd F fd
cmd X nop
cmd Y nop
cmd L lt 90
cmd R rt 90
```

Note that this task is optional and does not have to be solved for this part of the project to be considered as passed.

Project "From DNA to Proteins"



In nature, deoxyribonucleic acid (short: DNA) is used to encode genetic information of living organisms as sequences of bases. There are four bases found in DNA: adenine (short: `A`), cytosine (short: `C`), guanine (short: `G`), and thymine (short: `T`). One of the main functions of DNA is to encode the sequence of amino acids used in the construction of proteins.

In the second part of the project, we will translate the base sequences identified in the first part of the project into proteins. You find examples of output on the course home page (`chr1_gl000191_random.pro` and `chrX.pro`).

Task 0 – Preparation

For those, who did the first part of the "From DNA to Proteins" project, you need to modify your program to produce the list of base sequences that you found in the first part of the project. This should be easy as you already compute the start and end index of theses substrings. Do not forget to skip the start codon!

For those, who did the first part of the "Fractals and the Beauty of Nature" project, you need to download the files `chr1_gl000191_random.pbs` and `chrX.pbs`. The files contain a base sequence encoding a protein in each line. Thus, you need to write a function that reads one of these files and outputs a list of the lines without any whitespace.

Task 1 – Representing Amino Acids

Download the file `dna-codons.cdl` from the course home page. It encodes the genetic codes for the twenty amino acids. The description for one amino acid starts with a "`>`" followed by its abbreviation, its short name, and its long name. In the following lines until the next amino acid, each line identifies a codon that is translated to this amino acid.

Your task is to create a user-defined type (= Python class) "Acid" that represents one amino acid. This class should have attributes for at least the abbreviation, the short name, the long name, and the codons that encode this amino acid. For example, for aspartic acid this would be the abbreviation "D", the short name "Asp", the long name "Aspartic acid", and a list of codons containing exactly "GAT" and "GAC". Write a function that will read `dan-codons.cdl` and produce a list of 20 amino acids.

Task 2 – Setting up the Translation
As we will translate a large number of base sequence to proteins, it is important to have an efficient translation from codons to amino acids.

Your task is to create a user-defined type (= Python class) "Ribosome" that builds and stores a mapping of codons to amino acids. To this end, your type should have at least two associated functions or methods. First, you need a function that takes an amino acid (represented as in Task 1) and adds mappings from its codons to the amino acid. Second, you need a function that takes a codon (a base sequence of length 3) and returns the appropriate amino acid, i.e., the amino acid that this codon is translated to.

Task 3 – Creating Proteins
A protein is represented by a sequence of amino acids.

Your task is to create a user-defined type (= Python class) "Protein" that builds and stores sequences of amino acids. To this end, your type should have at least two associated functions or methods. First, you need a function that takes a base sequence and uses the function associated with the "Ribosome" class to translate it into a sequence of amino acids represented by instances of the "Acid" class. Second, you need a function that will convert a protein into a string. This function should have a parameter "mode" where a value of "0" means that a string of one-letter abbreviations is returned (e.g. `ADDYF`), a value of "1" means that a string of comma-separated short names is returned (e.g. `Ala, Asp, Asp, Tyr, Phe`), and a value of "2" means that a string of new-line separated long names is returned, e.g.:

```
Alanine
Aspartic acid
Aspartic acid
Tyrosine
Phenylalanine
```

Task 4* – Representing Codons
Codons are base sequences of length 3 that are encoded into exactly one amino acid. Your challenge task is to create a user-defined type (= Python class) "Codon" that represents a base sequence of length 3 and to use it instead of using strings in Tasks 0–3. You have to write a function or method, that takes a base sequences from Task 0 and translates it into a list of instances of "Codon". You will also need to write your own `__hash__(self)` and `__cmp__` methods in order to be able to use codons as keys for a dictionary.

Note that this task is optional and does not have to be solved for this part of the project to be considered as passed.