DM536/DM550/DM857
Introduction to Programming
Fall 2016 Project (Python)

Department of Mathematics and Computer Science
University of Southern Denmark

September 21, 2016

Introduction
The purpose of this project is to try in practice the use of programming techniques and knowledge about the programming language Python on small but interesting examples. There are three possible projects. You have to pick one of these. Please make sure to read this entire note before starting your work on this project. Pay close attention to the three sections below.

Exam Rules
This project is the final exam for DM536 and it is part of the final exam for DM550 and DM857. For DM550 and DM857 both this Python project and the Java project have to be passed in order to pass the overall exams. The project must be done individually, and no cooperation is allowed beyond what is explicitly stated in this document. You are allowed to discuss the project, problems, and solution ideas with other participants of the course.

Deliverables
As Part 1 of the Python project, the "Python driving license" test has to be passed with at least 50/100 points.

For Parts 2-4, at the end a short project report (at least 6 pages without appendix) has to be delivered. For Parts 2 and 3, you submit preliminary versions of this report covering the relevant tasks of the project.The final report has to contain the following 6 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)

The report has to be delivered as a PDF file (suffix .pdf) together with the Python source code files (suffix .py) electronically using Blackboard's SDU Assignment functionality. In addition, one printed copy has to be delivered to the "dueslag" of the respective teaching assistant. Do not forget to include the complete source code!

Deadlines
The description of the different projects indicates which tasks belong to which part (and which ones are optional).

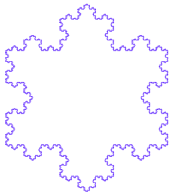
Part 1: Tuesday, September 27, 16:00
Part 2: Tuesday, October 4, 16:00
Part 3: Tuesday, October 18, 16:00
Part 4: Tuesday, November 1, 16:00

Project "Fractals and the Beauty of Nature"

Fractals are geometric objects that are similar to themselves on arbitrarily small scales. There are many examples of fractals in nature, and they form some of the most beautiful structures you can find. One example is snowflakes, but also lightning has a fractal structure. Another example are ferns, where each part is similar to the whole.

In this project, we will use the `turtle` module to generate fractals that are similar to structures found in nature.

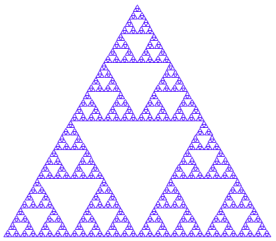
Task 1 – Preparation I (Part 2)



On the course home page you find an example using a Koch curve to render a snowflake (`koch.py`), with the expected results shown in `koch.png`. Use this example and experiment with the depth and the length parameters. Consult the documentation on the `turtle` module to understand the features used here.

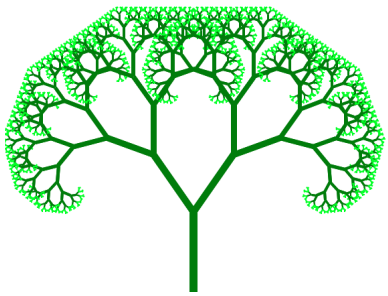
Task 2 – Sierpinski Triangle (Part 2)

A Sierpinski triangle is a triangle divided into three triangles, that are demselves divided into three triangles etc. On the course home page, you find four examples (`sierpinski0.png`, `sierpinski1.png`, `sierpinski2.png`, and `sierpinski5.png`). On the right-hand side you see a Sierpinski triangle of depth 5, i.e., a triangle that has been subdivided five times.



You task is to write a Python program that draws a Sierpinski triangle. To this end, you can follow the approach of the Koch Snowflake, i.e., at depth 0 you draw a triangle. At any other depth n, you draw Sierpinski triangle of depth $n-1$, move to the next corner, draw another Sierpinski triangle of depth $n-1$, move to the last corner, draw the third Sierpinski triangle of depth $n-1$, and, finally, move to the original corner.

Task 3 – Binary Tree (Part 2)

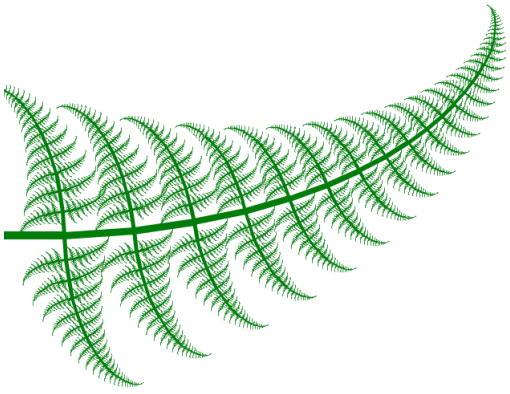


A binary tree is a tree, which is obtained by first drawing a stem and then subdividing into two branches. A tree consisting of just a stem is called a tree of depth 1. The picture to the left shows a tree of depth 12. On the course home page, you find examples (`tree3.png`, `tree6.png`, `tree9.png`, and `tree12.png`). Here, as an addition, the last two layers of branches have been colored `darkgreen` instead of `green`.

You task is to write a Python program that draws a binary tree. To this end, you have to modify the approach from the previous tasks slightly. The main idea is to draw a line, then turn and draw the left branch, then turn and draw the right branch, and, finally, go back where you came from.

Task 4 – Fern Time (Part 2, optional)

There are three differences between a fern and a binary tree. First, the fern has three branches from each stem and each subbranch. Second, the fern use some small constant angle to turn the middle branch. Third, the middle branch is scaled down less than the left and the right branch. On the course home page, you find examples (`fern10.png`, `fern3.png`, `fern1.png`, and `fern03.png`).



Your challenge task is to write a Python program that draws a fern. To this end, you have to modify the approach from the previous task. The main idea is that due to the third difference, a constant recursion depth is not useful in implementing a fern. Instead, define a limit for the size of the subbranches drawn and use this as the base case of the recursion.

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

Fractal Description Language

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 5 – Preparation II (Part 3)
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 the rules generate their respective fractals. While most of the fractals are known from earlier tasks, `dragon.fdl` represents a dragon curve and `sierpinski.fdl` represents a Sierpinski curve.

Task 6 – Applying One Rule (Part 3)
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 function `apply(left,right,state)` that applies the rule to each matching element of the list. For a function call `apply("F",["F","L","F","R","F","L","F"],["F","R","F","R","F"])` it should generate the list `[["F","L","F","R","F","L","F"],"R",["F","L","F","R","F","L","F"],"R",["F","L","F","R","F","L","F"]]`.

Task 7 – Flattening (Part 3) After applying one or more rules, the list representing the fractal is a mixed list consisting of both letters and lists as elements. Write a function `flatten(state)` that turns it into a simple list of letter. For a function call `flatten([["F","L","F","R","F","L","F"],"R",["F","L","F","R","F","L","F"],"R",["F","L","F","R","F","L","F"]]` the expected result is `["F","L","F","R","F","L","F","R","F","L","F","R","F","L","F","R","F","L","F","R","F","L","F"]`.

Task 8 – Increasing Depth (Part 3) Write a function `step(rules,state)` that takes a list `state`, applies all rules in `rules`, and finally flattens the resulting list.

Then write a function `compute(depth,rules,start)` that uses the `step` function `depth` times to transform the list `start` into a list representing a fractal of depth `depth`.

Task 9 – Representing Rules (Part 3, optional) Use a user-defined type (=Python class) to represent a rule. Use methods instead of functions for the functionality from Tasks 6–8, whenever this makes sense.

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

Task 10 – Preparation III (Part 4)
Try to understand the other parts of the fdl files, in particular the commands.

Task 11 – Executing Commands (Part 4)
A command consists of a command string (such as `fd`, `lt`, `rt`, `scale`, `nop`) and a list of arguments. Your task is write a function `execute(turtle,length, cmd,args)` for executing a command with its arguments and returning the current length. This function gets as the first two arguments a turtle object and a length.

The command with command string `lt` and argument list ["60"] should execute the Python statement `turtle.lt(60)` and return `length`. The command with command string `fd` should execute the Python statement `turtle.fd(length)` and return `length`. The command `scale` should multiply `length` with the given float and return it. Finally, the command `nop` simply does nothing (no operation) and returns `length`.

Task 12 – Loading a Fractal Description Language File (Part 4)
A fractal is described by a start state, a list of rules, a mapping from single letters to commands, a length, and a target depth.

Your task is to create a function that `parse(fdl)` that takes as argument the name of an fdl file and reads the different elements from the fdl file into appropriate data structures.

Task 13 – Loading a Fractal Description Language File (Part 4)
Your final task is to create a function that `run(turtle,fdl)` that takes as argument a turtle and the name of an fdl file. The function first uses `parse` to read the fdl file. Then it uses `compute` to compute the list of letters representing the fractal at the given target depth. Finally, it uses `execute` to execute all the commands represented by this list of letters.

Task 14 – Support for Pen Size and Colors (Part 4, optional)
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 the `randint` function from the `random` module and format strings:

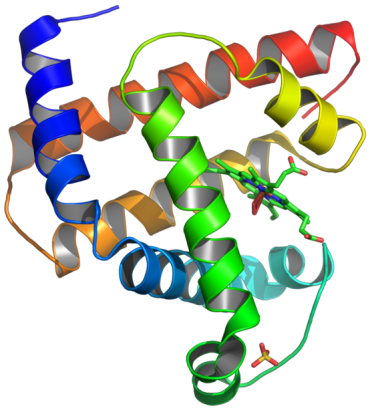
```
"#%02x%02x%02x" % (randint(0,255),randint(0,255),randint(0,255))
```

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 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.

Modern technological advances have made it possible to decipher this genetic information. You can find for example the base sequences of human chromosomes on the following web site:

`http://hgdownload.cse.ucsc.edu/downloads.html`

In this project, we will assemble base sequences from such files and analyze these sequences to identify proteins.

Task 1 – Preparation I (Part 2)

Download the file `hg38.chromFa.tar.gz` from the full data set for human and unpack it. Locate the file `chrX.fa` and view it in a text viewer or editor.

The first line is a short description of the sequence contained in the file. The rest of the lines is the base sequence. You will find lower-case and upper-case variants of the bases `A`, `C`, `G`, and `T`. In addition you will find `N`, which for the purpose of this project can be ignored.

Task 2 – Assembling the Sequence (Part 2)

For our purposes, we will ignore `N`s and we will ignore the difference between lower-case `a`, `c`, `g`, `t` and upper-case `A`, `C`, `G`, `T`.

Your task is to write a program that reads all lines and constructs one long string containing the base sequence. In this process, whitespace and `N`s have to be ignored. In addition, all base pairs should be represented by upper-case letters. Print the result and compare the beginning of the assembled sequence manually to the original file.

Task 3 – Finding Starting Points (Part 3)
The construction of a protein by a ribosome begins at a start codon. On our DNA, this is denoted by the base sequence `ATG`. This is called a *start codon*. Approximately 25 bases earlier in the sequence we often find a so-called TATA box. This is a base sequence `TATAAA`.

Your task is to write a function that will find all positions of a start codon that occurs 15-30 bases after the beginning of a TATA box. To this end, first write a function that will find all positions of the string `TATAAA` in the sequence. Then, find out how far the next ATG is located. If the distance is inside the admissible range, remember the index of the start codon.

Task 4 – Finding End Points (Part 3)
Each protein is encoded by a sequence of bases starting with ATG. Every amino acid is encoded by three bases. The end of a protein is given by a *stop codon*. This can be any of the following three sequences: `TAG`, `TAA`, or `TGA`.

Your task is to write a function that will identify the end point of a protein, i.e., the index of the first stop codon encountered after the start codon. To this end, you need to advance in steps of 3 through the base sequence until you encounter either the end of the sequence or a stop codon.

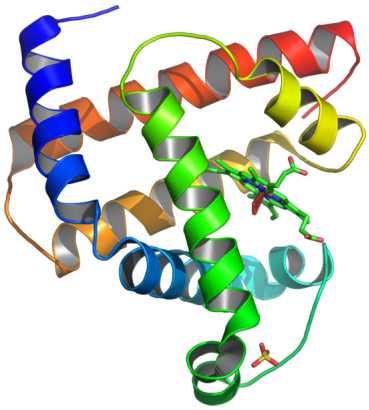
Task 5 – Proteins without TATA Boxes (Part 3, optional)
Not all proteins are prefixed by a TATA box occurring 15-30 bases earlier. In general, potential proteins can be overlapping. For example, the base sequence `ATGAATGAATAGATGA` contains a potential protein at index 0 (`ATGAATGAATAG`) and at index 4 (`ATGAATAGATGA`). We call this a genuine overlap. There is also trivial overlap, when `ATG` occurs inside a base sequence at a position divisible by three, e.g., `ATGATGTAG`.

Your challenge task is to identify all potential starts of proteins and answer the following two questions w.r.t. the file `chrX.fa`:

- How many genuine overlaps are there?
- How many potential proteins are there? Here, for genuine overlaps both proteins count while for trivial overlaps, only the longest protein counts.

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

Amino Acids



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 following tasks, we will translate the base sequences identified in the previous tasks into proteins. You find examples of output on the course home page (`chr1_gl000191_random.pro` and `chrX.pro`).

Task 6 – Preparation II (Part 4)
Start by modifying your program from the previous tasks to produce the list of base sequences that you found. This should be easy as you already compute the start and end index of theses substrings. Do not forget to skip the start codon!

Task 7 – Representing Amino Acids (Part 4)
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 8 – Setting up the Translation (Part 4)
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 6) 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 9 – Creating Proteins (Part 4)
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 10 – Representing Codons (Part 4, optional)
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 5–8. You have to write a function or method, that takes a base sequences from Task 5 and translates it into a list of instances of "Codon". You will also need to write your own `__hash__` and `__eq__` 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 project to be considered as passed.

Project "Mortgage Financing"

Most people cannot buy or build a house based solely on their savings. Instead, a part of the sum is financed by a mortgage loan, i.e., a loan that has security in the property bought or built.

Taking up a mortgage loan is a long-term decision as typical runtimes are 20-30 years. Thus, being able to model the financial implications during the runtime for different scenarios is useful when deciding which loan type to choose.

In this project, we will model mortgage loans out from a number of parameters and assumptions regarding future development of interest rates. The parameters and assumptions for each available loan type are provided in text files, which are read to in order to perform comparisons.

Task 1 – Preparation I (Part 2)
Download and view the files `fixed25.txt` and `flex1.txt` from the course home page. Understand what the parameters mean and by hand calculate how long it takes to pay back DKK 1 million with a monthly payment of DKK 10,000 assuming a stable interest rate.

Task 2 – Calculating Remaining Debt (Part 2)
Write a Python function that given the size of the debt, a fixed interest rate, and the size of the monthly payement calculates the remaining debt for each month until the whole debt is payed back. Test your function with the same setting as in Task 0.

Task 3 – Variable Interest Rate (Part 2)
Extend your function to take a function from month (identified by an integer) to interest rate instead of a fixed interest rate. Test your function with at least two different interest rate functions: one constantly returning 2.5 and one starting with 1.0 in the first month and increasing the interest rate by 0.1 each month.

Task 4 – Making it real (Part 3)
Write a function that takes a sum to be financed, a loan price, and a time frame. The function should determine the size of the monthly payment assuming a given interest rate and a debt corresponding to the desired sum

and the price of the loan. The course determines the actual amount of debt to take, also known as the "principal".

Task 5 – Data-driven Interest Rate (Part 3, optional)
Instead of assuming a fixed or a arithmetically varying interest rate, allow to use data from a file containing in the i-th line the interest rate for the i-th month.

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

Comparing Loan Types

Now that we can compute the consequences of a given loan type, we would like to present

Task 6 – Statistical Overview (Part 4)
Write a function that given loan parameters (interest rate, whether it is variable or not, time frame, current price) and a given sum to finance will use the functions developed in Tasks 2-4 to make an overview of the mortgage financing with this loan type. This overview should *at least* include

- total money paid over the full time frame
- effective yearly interest rate
- size of the principal
- remaining debt after each year

Task 6 – Using the Files (Part 4)
Ask the user for the sum to be financed and then read all available loan files in order to summarize the effects of choosing one over the other loan type.

Task 7 – Extra Payements (Part 4, optional)
In the program developed in Task 6, allow the user to specify extra payments at the end of years. Take them into account when showing the statistical overview.

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