# Exam: Introduction to programming (LT2111)

**Date: October 28, 2013, 9.00 – 12.00**

**Course responsible** Richard Johansson, Språkbanken, Department of Swedish

**Exam accessories** None

**Grade limits** Pass with distinction: 24p, Pass: 15p, Max: 30p

**Please note:**

- Write legibly (unreadable = wrong)!
- Make sure that your code has a clear indentation (if it is unclear, then the least favorable interpretation will be chosen).
- Number the pages, and start every question on a new page.
- Points will be removed for unnecessarily complicated or unstructured solutions.
- You may use built-in functions and methods, unless otherwise stated, but make sure that you know what they do. If you are unsure, define your own functions.
- If your solution to a question is only partial, please turn it in anyway! Every point counts.

## Question 1 of 6: Summing the even numbers (3 points)

Define a function `sum_even(lst)` that sums all even numbers in a list `lst`. All odd numbers are ignored. Since this is the first question, we will be nice to you and give you this utility function:

```
def is_even(nbr):
    return nbr % 2 == 0
```

Here are a few examples of the output of `sum_even`.

```
>>> sum_even([7, 8, 3, 7, 6, 1])
14
>>> sum_even(range(10))
20
>>> sum_even([])
0
```

## Question 2 of 6: Finding the longest word in a text file (4 points)

Define a function `find_longest_word(filename)` that reads a text file and returns the longest word in that file. (If there are more than one word with that length, return any of them.) The file consists of one word per line. For instance, if we process the text file

```
This
is
a
file
with
some
words
.
```

you should return the string `'words'`. You are free to ignore all encoding issues and file processing errors.

## Question 3 of 6: Lexical variety (4 points)

The *type–token ratio* (TTR) is used in many research fields to measure things like child language development or the effect on the vocabulary of a stroke or Alzheimer's disease. For a given document, TTR is defined as the number of word types $N_t$ (that is, the number of distinct words) divided by the total number of words in the document $N_w$:

$$\text{TTR} = \frac{N_t}{N_w}$$

Define a function `ttr(doc)` that computes the type–token ratio for a document `doc`, where `doc` is represented as a list of strings. For instance,

```
>>> ttr(['rose', 'is', 'a', 'rose', 'is', 'a', 'rose', 'is', 'a', 'rose'])
0.3
```

## Question 4 of 6: Readability (5 points)

In Swedish readability research, a numerical measure called **LIX** (*läsbarhetsindex*) is popular. It is a score that is computed for a given document, and it is defined

$$\text{LIX} = \frac{N_w}{N_s} + 100 \cdot \frac{N_{lw}}{N_w},$$

where $N_w$ is the number of words in the text, $N_s$ the number of sentences, and $N_{lw}$ the number of words whose length is greater than 6. A low score (near 0) means that the text is easy, and a high score (greater than 80) means it is hard.

Define a function `lix_score(text)` that computes LIX for a text. The text has already been segmented, so the input to the function is a list of sentences, and each sentence is a list of word strings.

For instance, given the input

```
[ ['The', 'hedgehog', 'lives', 'in', 'the', 'barn'], ['His', 'name', 'is', 'Oscar'] ]
```

the function should return the value 15.0, a typical LIX score for texts for children.

## Question 5 of 6: Spam filter evaluation (6 points)

Assume we have implemented a spam filter as a class `SpamFilter`. This class has a method `guess(email_text)` that takes an email string and returns `True` if it believes that the message is spam, otherwise `False`.

Now we want to evaluate how well our spam filter works. We get a collection of emails and categorize them *manually* as spam or non-spam (again, `True` or `False`). This is our *test set*, and we represent it as a list of tuples where the first item of the tuple is our manual categorization and the second item the email text. For instance, here is a small test set:

```
test_set = [(True, "Buy more prescription DRUGS CHEAP!!!! !"),
            (False, "Hi, how are you? How did the exam go?"),
            (False, "Dear student, Here is the result of your exam in Formal Linguistics."),
            (True, "Dear Madam, would you like to win $1000000000000?")]
```

We can now evaluate the filter by applying it to all the messages in the test set and compare its guesses to the manual categorizations. We define the *accuracy* and *false positive rate* as follows:

$$\text{accuracy} = \frac{n_{TT} + n_{FF}}{n} \qquad\qquad \text{FPR} = \frac{n_{FT}}{n_{FT} + n_{FF}}$$

where

- $n$ is the total number of messages in the test set,
- $n_{TT}$ the number of spam classified as spam,
- $n_{FF}$ the number of non-spam classified as non-spam,
- $n_{FT}$ the number of non-spam classified as spam.

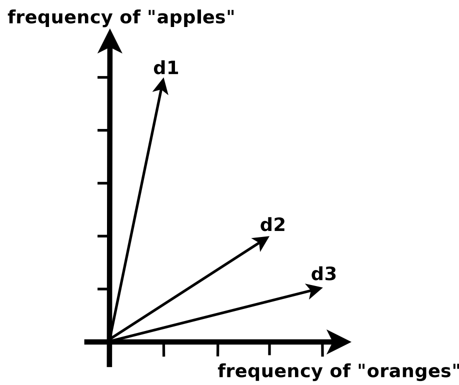
Define a method `evaluate` in the class `SpamFilter`. When calling `f.evaluate(ts)` on a spam filter `f`, the method returns the accuracy and false positive rate of `f` as evaluated on the test set `ts`. If we use the example test set above, and the third message is misclassified, we should get the result `(0.75, 0.5)`.

## Question 6 of 6: Document similarity (8 points)

In information retrieval systems, it is important to have some way to compare how similar two documents are. The most widely used document similarity measure is called *cosine similarity* and is based on a geometric view of documents. We first represent each document as a vector (an arrow) in a coordinate system. Each coordinate corresponds to the frequency of one of the words in the vocabulary. For instance, consider these three documents:

```
d1 = ['apples', 'apples', 'oranges', 'apples', 'apples', 'apples']
d2 = ['apples', 'oranges', 'apples', 'oranges', 'oranges']
d3 = ['oranges', 'apples', 'oranges', 'oranges', 'oranges']
```

This figure shows how the documents are represented. The coordinates of `d1` are (5, 1) because it has an `'apples'` frequency of 5 and an `'oranges'` frequency of 1. In our case the coordinate system has two dimensions since there are two words in the vocabulary, but in general there are more dimensions.




We compute the *dot product* of `d1` and `d2` by multiplying the coordinates of the two documents one by one, first the `'apples'` coordinate and then the `'oranges'` coordinate: $5 \cdot 2 + 1 \cdot 3 = 13$. Using the dot product, we can now define the cosine similarity:

$$\text{cos\_sim}(d_1, d_2) = \frac{\text{dot}(d_1, d_2)}{\sqrt{\text{dot}(d_1, d_1)} \cdot \sqrt{\text{dot}(d_2, d_2)}}$$

**Your tasks:**

**(a, 4p)** Implement a function `cos_sim(doc1, doc2)` that computes the cosine similarity between two documents. The documents `doc1` and `doc2` are represented as lists of words. **Hints**: This could be any two documents, not just about apples or oranges. Use `math.sqrt` to compute the square root ($\sqrt{\ }$).

**(b, 4p)** Implement a function `most_similar(d, n, docs)` that goes through a document list `docs` and returns the `n` documents most similar to `d`. Again, `d` is a list of words; the same is true for each of the documents in `docs`. **Hint**: Try to solve this task even if you are unable to solve task (a). Then just assume that `cos_sim` has been implemented.