1 Introduction
In this exercise, we are going to refine a tokenizer to deal with phenomena such as punctuation, abbreviations, contractions, and words containing non-alphabetic characters. We are also going to add a simple mechanism for sentence segmentation. We are going to use the same text as last time when developing the tokenizer, but in the end we are also going to evaluate it on a previously unseen (but similar) text.

2 The gold standard
The file dev1-gold.txt contains the correct tokenization of the text in dev1-raw.txt. Have a look at the tokenized text and make sure that you understand the principles by which it has been tokenized. Most of the tokens are either regular words or punctuation marks, but note the following:

- Numerical expressions like 3.8 are treated as single tokens.
- Logograms like % and $ are treated as separate tokens.
- Contractions like it’s and don’t are split into two tokens – but how?

3 A pattern-matching tokenizer
The file tokenizer1.py contains a very simple tokenizer:

```python
import re, sys
for line in sys.stdin:
    for token in re.findall("([,;:.!?""]|\w+())", line.strip()):
        print(token[0])
```

The main difference compared to the whitespace tokenizer used in Exercise 1 is that we are now trying to define what a valid token looks like, instead of just using whitespace to find token boundaries. Therefore, we use the Python method `re.findall` instead of `re.split`. This method takes as arguments a regular expression and a string, and returns a list of all the matches found in the string. The matching is done from left to right and greedily tries to find the largest possible match each time. Let us consider the regular expression used in tokenizer1.py:

```
"([,;:.!?""]|\w+())"
```

The regular expression, enclosed in double quotes ("), is a simple union expression, where the first part is a character set containing the most common punctuation marks ([,;:.!?\"]), and the second part is an expression (\w+) matching any non-empty sequence of alpha-numeric characters. Note that the double quotes must be escaped in the set of punctuation marks. Why? In addition, there are two pairs of round brackets in the regular expression, one enclosing the entire disjunction, and one enclosing an empty string at the end of the second part. These brackets are completely superfluous as far as the matching is concerned. They are only there to guarantee that the behavior of the program will not change if you later feel the need to add your own brackets for grouping. (Unless you are especially interested in the inner workings of Python, you can completely ignore this technicality for now as long as you don’t remove the brackets.)

**Note on Python versions:** In this and following exercises, we will use version 3 of Python because it assumes the Unicode text encoding as default. On most systems, this means that you should type `python3`, instead of simply `python`, when running a Python program. Make sure that you can get `tokenizer1.py` to run on your system before proceeding further and consult the teacher otherwise.

4 Refine the tokenizer
Run `tokenizer1.py` on dev1-raw.txt and compare the output to dev1-gold.txt using the linux command `diff`. This gives you a list of the problems you have to fix. Your task is to modify the regular expression used for matching in order to eliminate as many as the problems as possible. A good strategy is to...
concentrate on one type of token at a time, and add a new subexpression to the disjunction, specifically designed to handle that token type. Note that Python processes the regular expression form left to right, which means that earlier subexpressions will take priority over later subexpressions if they have overlapping matches. (The two subexpressions in the original expression never have overlapping matches. Why?) If you need help, consult the Python documentation at https://docs.python.org/3/howto/regex.html. If you want to measure the exact precision and recall on the token level, you can use `score_tokens.py`.

5 Add sentence segmentation

The final task is to add sentence segmentation to the tokenization process. In general, this is a rather hard problem, but for the sentences in `dev1-raw.txt` it should not be too difficult to come up with something reasonable. It is customary to mark sentence boundaries by a blank line, so you just need to come up with rules for inserting a blank line before a token that starts a new sentence. When you have done this, you can compare your output to `dev1-gold-sent.txt`, which is identical to `dev1-gold.txt` except that sentence boundaries have been inserted.

6 Evaluate the system

When you are confident that you have made all the improvements possible, let the teacher know and you will get a fresh text called `dev2-raw.txt` to run your tokenizer on. Compare the output of your system against `dev2-gold-sent.txt` and discuss the result.

7 Submitting the first assignment

The first assignment of the course consists of your work with Exercise 1 and 2. Make sure that you submit all of the following to joakim.nivre@lingfil.uu.se at the end of the first week:

- A manual tokenization of (at least the first 10 sentences of) `dev1-raw.txt`.
- A refined tokenizer that has at least 95% precision and recall on `dev1-raw.txt`.
- A discussion of what you learned from running your tokenizer on `dev2-raw.txt`. 