Solution to the exam
LT2111: Introduction to programming

Date: October 19, 2015, 9.00 – 12.00

Question 1 of 6: Printing the short words (3 points)

Not much to say here: just go through the list and print the relevant words.

```
def print_short_words(lst, n):
    for w in lst:
        if len(w) < n:
            print(w)
```

Question 2 of 6: Problems, problems (5 points)

(a, 2p) The problem is that when we read text from a file, it will end up as a string. So if we have the code

```
            course_name, credits = line.split()
            total_credits += credits
```

then the variable `credits` is a string, and if we try to add a string to an integer (`total_credits`) then we get the exception. The easiest solution is to convert `credits` into an integer:

```
            total_credits += int(credits)
```

(b, 3p) It seems stupid and convoluted to build a frequency table using a list when Python provides a number of data structures that are more suitable for this purpose, such as the dictionary, `defaultdict`, and `Counter`. Strictly speaking the code works, but it will not work well if the corpus is large, since the lookup function `get_word_freq_pair` is slow: it requires us to go through the words we have seen, so it is a *linear-time* operation. This means that the whole program has a *quadratic* time complexity. In contrast, lookup is done in *constant time* when using a dictionary.

So the solution would be to replace the convoluted list-based solution with one of the data structures mentioned above. To get 3 points, it is not mandatory to go into the details of time complexity: it is enough if you mention that the proposed solution is inefficient because we need to go through the list. And you should mention that it is better to use some sort of dictionary.

Question 3 of 6: Evaluation of a dependency parser (5 points)

Our task is to compare two lists of integers of the same length and see how many "matches" there are. One solution is that we go through all positions from 0 up until the length of the sentence. Then we compare the dependency links (integers) at each positions, and count how many of the links match:

```
def evaluate_parser(tree_a, tree_b):
    count_correct = 0
    for i in range(len(tree_a)):
        if tree_a[i] == tree_b[i]:
            count_correct += 1
    return count_correct / len(tree_a)
```

The code can be made a bit more neat if we use the standard function `zip`, which takes two lists and returns a new list (well, formally a *generator*) of pairs of integers. Then we can look at each pair `(a, b)` and see how often `a` is equal to `b`:

```
def evaluate_parser(tree_a, tree_b):
    count_correct = 0
    for a, b in zip(tree_a, tree_b):
        if a == b:
            count_correct += 1
    return count_correct / len(tree_a)
```

For sport, let's see if we can make the solution even more compact. This can be done with a list comprehension. We also use the fact that if we apply `sum` to a list of boolean values, it will count the number of `True` in that list:

```
def evaluate_parser(tree_a, tree_b):
    return sum(a == b for a, b in zip(tree_a, tree_b)) / len(tree_a)
```

Question 4 of 6: Lexicon-based part-of-speech tagging (5 points)

(a, 2p) The straightforward option is to use a dictionary for the tag lexicon. Here is a typical solution. We just read the lines, split each line into word and tag, and store the tag for each word in the dictionary.

```
def read_tag_lexicon(file_name):
    lexicon = {}
    with open(file_name) as f:
        for line in f:
            word, tag = line.split()
            lexicon[word] = tag
    return lexicon
```

It would also be OK if you read the whole lexicon file, and then split it into lines.

(b, 3p) The function `lexicon_tagger` just goes through the words in the corpus, and for each word we print the tag listed in the lexicon. If the word doesn't exist in the lexicon, we print `unknown`. The solution here will depend on what structure you used in (a); this code assumes that we used a dictionary as in the solution above.

```
def lexicon_tagger(lexicon, file_name):
    with open(file_name) as f:
        for line in f:
            word = line.strip()
            if word in lexicon:
                 print(word, lexicon[word])
            else:
                 print(word, 'unknown')
```

Question 5 of 6: Information retrieval (6 points)

We first define a helper function `count_hits` that determines how many occurrences there are in the document for each of the words in the list `query_terms`.

```
def count_hits(doc, query_terms):
    count = 0
    for q in query_items:
        count += doc.count(q)
    return count
```

Alternatively, a more compact implementation using a list comprehension.

```
def count_hits(doc, query_terms):
    return sum(doc.count(q) for q in query_terms)
```

It would also be OK to do it the other way around: go through the words in the document, and see how many of them are present among the query terms.

```
def count_hits(doc, query_terms):
    count = 0
    for word in doc:
        if word in query_terms:
            count += 1
    return count
```

We then apply the helper function to all the documents in the collection. We build a list of pairs, where the first item in the pair is the number of hits, and the second is the actual document. (A list comprehension is used here, but a straightforward solution where we `append` to a list is also acceptable.) We then sort that list so that the documents with the most hits come out on top, and cut the list so that we return the `n` documents where the query terms are most frequent.

```
def search(collection, query_terms, n):

    # A list of count/document pairs.
    # For instance (3, ['a', 'cat', 'is', 'not', 'a', 'dog'])
    scored = [(count_hits(d, query_terms), d) for d in collection]

    # We sort the list using reverse=True so that the documents with
    # the most hits are placed near the beginning of the list.
    # Recall that we can sort a list of tuples.
    scored.sort(reverse=True)

    # We finally return the first n elements of the list.
    return scored[:n]
```

Question 6 of 6: Deriving a polarity lexicon from a corpus (6 points)

(a, 2p) We go through all the words in the document. Each time we see a word that is listed in the polarity lexicon, we add the score for that word to the total score. We ignore the words that aren't in the lexicon – they are implicitly assumed to have a score of 0. We return `positive` if the final score is greater than (or equal to) zero, and `negative` otherwise.

```
def guess_polarity(lexicon, document):
    score = 0
    for word in document:
        score += lexicon.get(word, 0)
    if score >= 0:
        return 'positive'
    else:
        return 'negative'
```

This solution used `get` instead of plain dictionary lookup (`lexicon[word]`) to make the code a little bit more compact.

It is up to you to decide how you deal with the case when the final score is exactly zero, since the problem statement did not say exactly what to do in that case. It is OK if you return `positive`, `negative`, or even something else such as `unknown`. The algorithm in (b) should work regardless of which solution you choose, but its result will of course be slightly different.

(b, 4p) This is just a matter of converting the informal description in the problem statement into working Python code. Here is a solution. (Again, we used `get` to save a few lines, but we could also have used `defaultdict` or `Counter` for the same reason.)

```
def build_polarity_lexicon(labeled_docs, n):

    # start with an empty lexicon
    # (that is, all polarity scores are implicitly zero)
    lexicon = {}

    # go through the corpus n times
    for i in range(n):

        # for each document, with its human-annotated polarity label
        for true_polarity, doc in labeled_docs:

            # use the current version of the lexicon to guess the
            # polarity of the document
            guess_polarity = classify_document(lexicon, doc)

            # if the guess was incorrect
            if true_polarity != guess_polarity:

                # if we misclassified a positive document
                if true_polarity == 'positive':
                    # then increase the polarity scores for all the words in
                    # that document by 1
                    for w in doc:
                        lexicon[w] = lexicon.get(w, 0) + 1
                # conversely, if we misclassified a negative document
                else:
                    # then instead we subtract 1 from all the polarity scores
                    for word in doc:
                        lexicon[w] = lexicon.get(w, 0) - 1

    return lexicon
```