15-110 Fall 2019
Homework Exam 02

Out: Thursday 5th December, 2019 at 15:00 AST
Due: Thursday 5th December, 2019 at 16:20 AST

Introduction

This homework exam includes questions selected from previous homework assignments.
The total number of points available from the questions is 125, where 25 points are *bonus* points (i.e., you only need 100 points to get the maximum grade).

Warning: if the code of a function doesn't execute because of syntax or logical errors of *any type* (i.e., the function doesn't even reach the end) then you'll get 0 points, the function won't be evaluated at all during the grading process! This means that you should / must try out your code, function by function, before submitting it!

In the handout, the file `hwexam02.py` is provided. It contains the functions already defined but with an empty body (or a partially filled body). You have to complete the body of each function with the code required to answer to the questions.
You need to submit to Autolab the `hwexam01.py` file with your code.

Only the provided *reference cards* (possibly with your annotations) are admitted as a support during the exam.

The code must be written and tested using Spyder on the computers in the classroom.

1 Most Common Word

Processing large amounts of text can be a very useful exercise. Imagine that you have a large amount of text and you want to figure out what the most frequently occuring word is in that text. This might be a somewhat complicated task, because normal text has punctuation, the same word may be written with different upper and lowercase letters, etc. For example, "monkey", "Monkey", "monkey,", and "Monkey." are all the same word despite varying letter case and punctuation.

Problem 1.1: (50 points)

Implement the function `most_common_word(text)`, which, given a string of text returns the word that most frequently occurs (without respect to upper and lowercase) in the form it most frequently occurs in (with respect to upper and lowercase).

For example, consider the text `"hi␣bob␣there␣Hi␣bob␣hI␣Hi␣bob"`. The most frequently occurring word in the text (without respect to case) is `"hi"`, which occurs 4 times. The form it occurs most frequently in is `"Hi"`, which occurs twice. So, that means that: `most_common_word("hi␣bob␣there␣Hi␣bob␣hI␣Hi␣bob")` should return `"Hi"`.

Further complicating things is the fact that the text may contain punctuation, numbers, special characters, and extra space characters, and those should not affect the answer. (You should ignore anything that is not a letter.) So, that means that: `most_common_word("hi,␣bob␣$there␣{Hi␣bob␣hI.␣␣␣Hi%␣bob")` should still return `"Hi"`.

You must not try to write all of your code in one function. You should alwyas write helper functions that each do small parts of the task and then use those to solve the problem. Here is the set of helper functions that you must implement:

- `filter_text(text)` (10 pts): Given a **string** of `text`, return a new string that contains only the letters and whitespace from `text`. (This filters our all of the characters we don't care about.)
- `most_frequent_ignore_case(words)` (15 pts): Given a **list** of `words`, return the word that occurs most frequently in the list, ignoring case. To keep track of the frequency of each word as the words are processed, you should use a dictionary where the key is a word and the value is its frequency.
- `most_frequent_form(words, word)` (15 pts): Given a **list** of `words` and a `word` in lowercase, return the most frequent form of `word` in `words`. You can use the same idea as before: a dictionary to keep track of word frequencies. Except this time capitalization will matter, and you only need to count the words that are the same as **word**.

Remember: Build your solution incrementally instead of trying to solve the entire problem at once. First, build a solution that can find the most common word (without regards to case) from a simple text with no special characters. Once that works, then improve it to also work in the case of strange punctuation. Once that works, improve it again to return the most common form of the word.

The following functions may be useful for the tasks above[1]

- `s.split(sep)`: Return a list of the words in the string, using `sep` as the delimiter string. If `sep` is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace.

[1] Do not feel like you are supposed to use all of them, though!

- `s.lower()`: Return a copy of the string with all the cased characters converted to lowercase.
- `s.upper()`: Return a copy of the string with all the cased characters converted to uppercase.
- `s.isalpha()`: Return true if all characters in the string are alphabetic and there is at least one character, false otherwise.
- `s.isspace()`: Return true if there are only whitespace characters in the string and there is at least one character, false otherwise.

2 DNA Representation

When DNA is stored in a computer, it can be represented as a list of letters. The letter A is used for adenine, C for cytosine, G for guanine, and T for thymine.

An important problem in working with DNA is *subsequence matching*. In short, subsequence matching determines if a short DNA sequence occurs within a longer sequence. (All the letters on the subsequence occur consecutively within the original sequence.) For example:

- The sequence `['A','C','C','C','C','T','T','T']` contains the subsequence `['C','C','T']` once.
- The sequence `['A','C','C','T','A','C','C','T']` contains the subsequence `['C','C','T']` twice.
- The sequence `['A','C','C','C','T','T','T','T']` contains the subsequence `['T','T']` three times.

Problem 2.1: (15 points)

Implement the function `simple_subseq_match(sequence, subseq)` which, given a DNA sequence `sequence` and a shorter DNA sequence `subseq`, returns how many times `subseq` occurs within `sequence`. You can assume that both `sequence` and `subseq` are lists containing only `'A'`, `'C'`, `'T'` or `'G'`.

Sometimes, when doing subsequence matching, we allow partial matches instead of perfect matches. A partial match is one where we do not care about the value of certain characters. If we do not care about a character, we give it the value `N`. For example, `['A', 'N', 'C']` can match any three letter sequence that starts with `A` and ends with `C`, such as `['A', 'T', 'C']`, `['A', 'G', 'C']`, etc.

Problem 2.2: (35 points)

Implement the function `subseq_match(sequence, subseq)` which, given a DNA sequence `sequence` and a shorter DNA sequence `subseq` which may contain `N` values, returns how many times `subseq` occurs within `sequence`. You can assume that both sequences are valid. Here are some examples:

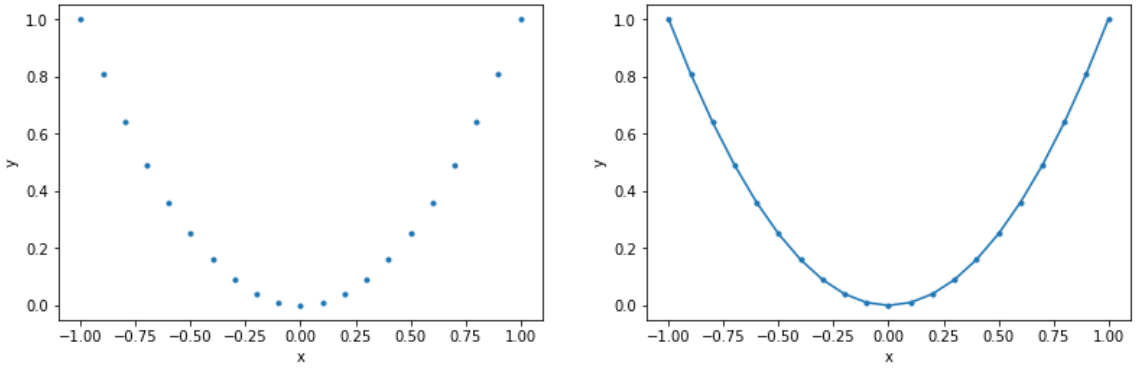
- `subseq_match(['A','C','C','T','C','T'], ['T','N','T'])` returns `1`.
- `subseq_match(['A','C','C','T','A','G','C','T'], ['N','C','T'])` returns `2`.
- `subseq_match(['A','C','C','C','T','T','T','T'], ['N','T','T'])` returns `3`.

3 Plot a function

A *mathematical* function $f : X \to Y$ maps every point $x \in X$ to a point $y \in Y$, where X and Y are subsets of the real set, $\mathbb{R}$. The notation $y = f(x)$ indicates the value y that the function associates to the value x.

To plot the mathematical function $f(x)$ using a programming language (i.e, using programming functions), we need to select n points $x_i \in X,\ i = 1, \ldots, n$, compute the corresponding values $y_i = f(x_i)$, and plot all the n pairs (x_i, y_i), where each pair will correspond to a dot/point in the $X - Y$ Cartesian plane. We can also join the dots by a line, such that an overall smooth plot of the function will appear.

The figure below shows an example for the the function $f(x) = x^2$, where the figure to the left shows the function evaluated at the points from -1 to 1 (included) with a step of 0.1 (for a total of 21 points). The figure to the right shows the same thing but with the dots now being interpolated by lines.



Problem 3.1: (15 points)

Implement the function `plot_function(f, xmin, xmax, step, lines=True)` that takes as input a function object, `f`, that implements a mathematical function, three floats, `xmin`, `xmax`, `step`, and one optional boolean parameter, `lines`. The function plots the value of the mathematical function `f` for the values of x between `xmin` and `xmax` (included) every `step` points (as in the example above). The value of the optional parameter `lines` sets whether points are interpolated by lines or not, as it is explained below.

For plotting, `function_plot(f, xmin, xmax, step, lines=True)` makes use of the helper function `plot_points(x, y, lines=True)` that takes as input two lists of floats, `x` and `y`, that correspond to the (x_i, y_i) pairs to be plotted. The optional parameter `lines` sets the type of the plot: if `lines` is set to `True`, the plot shows interpolating lines, instead only dots are plotted when `line` is set to `False`.

In practice, in the function `plot_function()`, you have to generate the paired lists of x and y points and pass them to the helper function `plot_points()` that will manage their plotting. To generate the x and y lists of points you must use the `arange()` function from the `numpy` module, that allows to generate lists of real numbers.

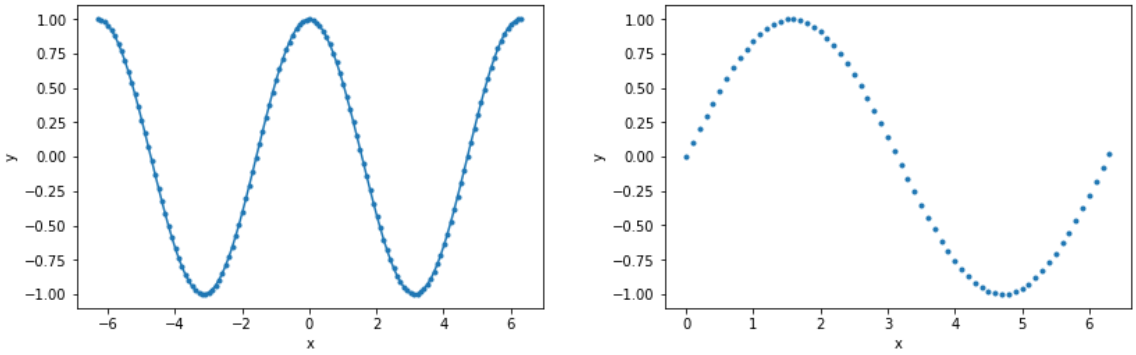
The code of the helper function is the following (and it is provided in the handout):

```
def plot_points(x, y, line=True):
  plt.figure()
  plt.xlabel('x')
  plt.ylabel('y')
  if line:
      linestyle = '-'
  else:
      linestyle = ''
  plt.plot(x, y, marker='.', linestyle=linestyle)
```

The helper function `plot_points()` makes use of four functions (`figure()`, `xlabel()`, `ylabel()`, `plot()`) from the module `matplotlib.pyplot`. In the code, the name of the module as been shortened and aliased to `plt`. You need to add, before the definition of `plot_points()`, the `import` statement that allows to correctly use the functions from the module as they are written.

Once you have written the function `function_plot()` and have added the required import statement, you can test the function and see the plotting graphs! You can test it using the functions provided by the `math` module (see lecture slides). Remember that in order to do this, you need to include the `import` statement for the module.

For instance, invoking the function as `plot_function(math.cos, -2 * math.pi, 2 * math.pi, 0.1)` and as `plot_function(math.sin, 0, 2 * math.pi, 0.1, False)` will produce the left and right plots shown in the figure below (`math.pi` is the value of π, as provided by the `math` module).

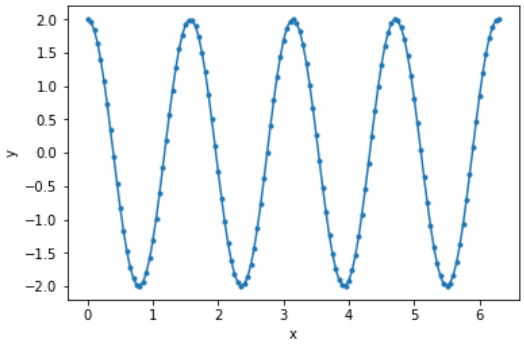


Problem 3.2: (10 points)

If you have successfully answered the previous question, now you have a practical tool to plot *any* mathematical function of interest. Remember that the first argument of `plot_function()` is a function object that implements a mathematical function. Therefore, let's build a little *library* of python functions each implementing a different mathematical function that can be displayed using `plot_function()`!

Implement the functions `parabola(x)`, `cubic(x)`, `periodic(x)` that realize, respectively, the mathematical functions $f(x) = 2x^2, f(x) = x^3 + 2, f(x) = 2\cos(4x)$. Each (pyhton) function takes as input a real value `x` and returns the value $f(x)$ that the mathematical function associates to x. For instance, `parabola(2)` returns 4, while `parabola(3.5)` returns 12.25.

For instance, invoking `plot_function(periodic, 0, 2 * math.pi, 0.05, True)` will produce the plot shown below.


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y
0
1
2
3
4
5
6
x