Reexam

Formal requirements and practicalities:

How do I hand-in?

The final version of your exam must be submitted to the Digital Exam system (https://eksamen.ku.dk). The submission to digital exam should consist of two files: `exam.py` and `exam_test.py`. While the exam is ongoing, you can hand in as many times as you like, so feel free to submit preliminary versions during the exam (but be careful not to submit as a final version).

Is the auto-correction server available during the exam?

Unlike the hand-ins throughout the course, you will (of course) **not** be able to get detailed feedback on your solution by submitting it to CodeGrade. However, we have set up the system to allow you to test whether the basic structure of your files is correct. It works exactly like the weekly handins: in Absalon, you will find an assignment called "Reexam" - under this assignment, you can submit your code to CodeGrade and get feedback as usual. Please note that even if all tests pass, it says nothing about the correctness of your code, so you cannot use it to validate your solution, but if a test fails, it means that this particular exercise does not run, and will therefore not be assessed for the exam. Using this service is optional, but we highly recommend using it to rule out any silly mistakes such as spelling errors in function names. **Please note that submitting to CodeGrade is not considered "handing in" - you must submit to digital exam in order for your exam to be registered as submitted**.

Format:

You should create the following two python files for the exam:

1. A Python file called `exam.py`, containing the function and class definitions.
2. A Python file called `exam_test.py`, containing test code for the individual questions.

The details about which code should go where will be provided in the questions below.

Content:

For the Python part, remember to use meaningful variable names, include docstrings for each function/method/class, and add comments when code is not self-explanatory. Many of the questions will instruct you to solve the exercise using specific tools (e.g. using only for-loops) - please pay attention to these instructions. Also, **please use external modules only when they are explicitly mentioned in the exercise**. Failure to do any of these will force us to deduct points even for code that works.

Can we lookup things on the internet?

Yes. You are free to search for online documentation and use websites like stackoverflow during the exam, but please try to avoid copying large blocks of code verbatim from online examples. If you for some reason find it necessary to copy code directly (without rewriting it), then please add a comment that specifies the source of this code (so that we can rule out plagiarism if two of you copied the same block). You are **not** allowed to communicate with anyone during the exam, also not using online chat services, mail etc.

Can we use ChatGPT, copilot or similar services?

No. Use of these services is prohibited. If you are caught using these services, it will be regarded as exam cheating, and you will be reported to the exam office.

How should I structure my time?

Note that the Q1, Q2, Q3, Q4, and Q5 are independent from another - in the sense that you can solve any of them without solving the others. Some exercises will also be easier than others. This means that if you are stuck on something, we strongly encourage you move on to one of the exercises following it, so that you are sure you don't miss any exercise that you could have solved.

What do I do if I find an error in the exam?

If you find an error or ambiguity in the exam, please write an email to **wb@di.ku.dk (mailto:wb@di.ku.dk)**. If we agree that something should be clarified, we will post an Absalon announcement with a clarification.

Q1

In this first exercise, we will work on a files called `lines.txt`. **You should solve this exercise in pure Python (without importing any modules).**

Download the file from: **https://wouterboomsma.github.io/ppds2022/data/lines.txt (https://wouterboomsma.github.io/ppds2022/data/lines.txt)** to your current directory. Start by opening the file in some text viewer or editor to see its content.

a. In the `exam.py` file, write a function called `print_lines` that takes a single argument called `filename` (a string). the function should open the file and print out the lines, such that they appear as in the original file (i.e. no double new lines). **You must use a python for-loop for this task (list-comprehension not allowed)**.

   In `exam_test.py`, call the `print_lines` function on `lines.txt`.

b. In the `exam.py` file, write a function called `order_numbers` that takes a single argument: `filename` (a string). The function should open the file and extract the numeric value in each line. It should then sort these in numerical order, and print them such that each of the numbers appear on a new line. **You must use a python for-loop for this task (list-comprehension not allowed)**. You are allowed to use Python built-in functions/methods to do the sorting.

   In `exam_test.py`, call the `order_numbers` function on `lines.txt`.

c. In the `exam.py` file, write a function called `order_lines` that takes a single argument: `filename` (a string). The function should open the file, and iterate over the lines. But this time, it should print out the full lines sorted in their numerically natural order. **You must use a python for-loop for this task (list-comprehension not allowed)**. You are allowed to use Python built-in functions/methods to do the sorting.

   In `exam_test.py`, call the `order_lines` function on `lines.txt`.

d. In the `exam.py` file, write a function called `order_lines_compact` that takes a single argument: `filename` (a string). The function should do the same as `order_lines`, but the function definition should now consist of only two lines of code: one that opens the file using the `with` statement, and a second line that does all the processing in a single line and prints the result. You are allowed to use Python built-in functions/methods to do the sorting. Hint: you can use list comprehension and lambda functions to solve this.

   In `exam_test.py`, call the `order_lines_compact` function on `lines.txt`.

Q2

There is a open source project that maintains postal codes (aka zip codes) in many different countries: **https://github.com/zauberware/postal-codes-json-xml-csv (https://github.com/zauberware/postal-codes-json-xml-csv)** . We will be looking at the Danish postal codes in this exercise. Start by downloading **https://wouterboomsma.github.io/ppds2022/data/zipcodes.dk.csv (https://wouterboomsma.github.io/ppds2022/data/zipcodes.dk.csv)** to your current directory. **You should solve this exercise in pure Python (without importing any modules).**

a. In the `exam.py` file, write a function called `read_zipcode_file` that takes a single argument called `filename` (a string). The function should return a dictionary, where the zipcodes are the keys, and each value is a dictionary with key,value pairs corresponding to all the column names in the file (e.g. `{'country_code': 'DK, 'zipcode': '8789', ..., 'longitide':'10.271'}`). Some zip-codes will appear on multiple lines in the file; for these, you should use the first occurrence.

   In `exam_test.py`, test your function by calling it on `zipcodes.dk.csv`. Save the result in a variable called `zipcode_dict`.

b. In the `exam.py` file, write a function called `count_zipcodes_per_state` that takes two arguments: `zipcode_dict` (a dictionary like the ones returned by the `read_zipcode_file` function), and `state` (a string). The function should count how many different zipcodes are associated with a given state and return this value as an integer.

   In `exam_test.py`, test your function by calling it on `zipcode_dict` and `"Region Hovedstaden"`. Save the result in a variable called `region_hovedstaden_zip_count`.

c. In the exam.py file, write a function called `max_distance` that takes a single argument: `zipcode_dict` (a dictionary like the ones returned by the `read_zipcode_file` function). The function should find the pair of zip-codes in Denmark which are separated by the maximum distance. It should return a tuple containing three values: the distance (float) and the two associated zip-codes as integers (the two zip-codes can be in any order) . You should use the following function to calculate the distance between two longitude,latitude points:

```
def lon_lat_distance(lon1, lat1, lon2, lat2, radius=6373):
    '''Calculate distance (in km) between two (longitude, latitude) pairs'''
    import math
    theta1 = lon1*math.pi/180.   # Convert to radians
    theta2 = lon2*math.pi/180.
    phi1 = (90.-lat1)*math.pi/180.  # Subtract from 90 to get spherical coordinates
    phi2 = (90.-lat2)*math.pi/180.
    arc_len = math.acos(min(1.,math.sin(phi1)*math.sin(phi2)*math.cos(theta2-theta1)+math.cos(phi1)*math.cos(phi
2)))
    return arc_len * radius
```

   In `exam_test.py`, test your `max_distance` function by calling it on `zipcode_dict`. Save the result in a variable called `max_distance_zipcode_pair`.

Q3

In this exercise we will be using regular expressions. You are expected to use the `re` module.

Download the following file: **https://wouterboomsma.github.io/ppds2022/data/happy_sad.txt** (https://wouterboomsma.github.io/ppds2022/data/happy_sad.txt) to your current directory. Open it in a text editor/viewer to see its content.

a. In the `exam.py` file, create a variable called `happy_sad_re` which contains a regular expression that matches the emoticons listed in the file - **and only these emoticons**.

   In `exam_test.py`, test your `happy_sad_re` by trying to match the entries in the `happy_sad.txt` file. Also test it on some cases which you don't expect it to match.

b. In the `exam.py` file, create a new variable `happy_sad_re2`. It should do the same as the regular expression in the previous exercise, but it should introduce two groups: one that allows you to extract the eyes and nose, and one that allows you to extract the mouth. Make sure that the regular expression contains exactly two groups (such that the eyes and nose can be extracted as by group index `1`, and the mouth can be extracted by group index `2`).

   In `exam_test.py`, test your `happy_sad_re2` by trying to match the entries in the `happy_sad.txt` file, and extracting the groups.

c. In the `exam.py` file, write a function called `find_happy` that takes a single argument called `input_text` (a string). The function should iterate over the input lines, and for each line print out all happy emoticons (faces with a `')'` mouth) appearing in that line - such that each emoticon in the output appears on a separate line.

   In `exam_test.py`, test your `find_happy` function on the following string `":( :( :(\n:) :( :( :( ;)\n:-( (bla bla bla) :)"`

d. In the `exam.py` file, write a function called `make_string_happy` that takes a single argument called `input_text` (a string). The function should return a new string, where all sad faces (faces with a `'('` mouth) have been replaced with the equivalent happy face. So, for example `":("` should be replaced with `":)"` while `":-("` should be replaced with `":-)"`.

In `exam_test.py`, test your `make_string_happy` function on the following string `":( :( :(\n:) :( :( :( ;)\n:-( (bla bla) :)"`

Q4

In this exercise we will use `numpy` (points might be deducted if you use for-loops when a numpy solution is possible).

Download the following two files: **https://wouterboomsma.github.io/ppds2022/data/smiley_text_image.txt (https://wouterboomsma.github.io/ppds2022/data/smiley_text_image.txt)** and **https://wouterboomsma.github.io/ppds2022/data/smiley_text_image_labels.txt (https://wouterboomsma.github.io/ppds2022/data/smiley_text_image_labels.txt)**. Start by opening the `smiley_text_image.txt` file in a text editor/viewer to see its contents.

a. In the `exam.py` file, write a function called `read_image_array` that takes single argument called `filename` (a string). The function should open the file using numpy's `loadtxt`, and return the resulting array. Hint: to get this to work, you will need to set `dtype=str`, `comments=None`, and use an appropriate delimiter.

   In `exam_test.py`, test your code by calling the `read_image_array` function on the `smiley_text_image.txt` file. Save the result in a variable called `smiley_image_array`. Print the array to screen to see what it looks like.

   In `exam_test.py`, also call `read_image_array` on `smiley_text_image_labels.txt` and save the result in a variable called `smiley_image_array_labels`. We'll get back to this labels file in exercise 4d.

b. As you can see, the array image uses `'#'` as foreground characters and `' '` as background. In `exam.py`, write a function called `change_image_array_foreground` that takes two arguments: `image_array` (an numpy array like `smiley_image_array`), and `foreground_char` (a string). The function should return a new numpy array where the `"#"` characters have been replaced by `foreground_char`.

   In `exam_test.py`, test your function by calling it on `smiley_image_array` and setting `foreground_char` to `"X"`. Save the result in a variable called `smiley_image_array_X`. Print it to screen to see what it looks like.

c. In the `exam.py` file, write a function called `frame_image_array` that takes two arguments: `image_array` (a numpy array like `smiley_image_array`), and `frame_width` (an integer). The function should return a new array, which has a frame around its content. For example, for `frame_width=1`:

```
                                              [['#' '#' '#' '#' '#' '#' '#' '#' '#' '#' '#']
[[' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']         ['#' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' '#']
 [' ' ' ' '#' '#' ' ' '#' '#' ' ' ' ']         ['#' ' ' ' ' '#' '#' ' ' '#' '#' ' ' ' ' '#']
 [' ' ' ' '#' '#' ' ' '#' '#' ' ' ' ']         ['#' ' ' ' ' '#' '#' ' ' '#' '#' ' ' ' ' '#']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']         ['#' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' '#']
 [' ' ' ' ' ' ' ' '#' ' ' ' ' ' ' ' ']    ->   ['#' ' ' ' ' ' ' ' ' '#' ' ' ' ' ' ' ' ' '#']
 [' ' '#' ' ' ' ' ' ' ' ' ' ' '#' ' ']         ['#' ' ' '#' ' ' ' ' ' ' ' ' ' ' '#' ' ' '#']
 [' ' ' ' '#' '#' '#' '#' '#' ' ' ' ']         ['#' ' ' ' ' '#' '#' '#' '#' '#' ' ' ' ' '#']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']         ['#' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' '#']
 [' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ']]        ['#' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' '#']
                                               ['#' '#' '#' '#' '#' '#' '#' '#' '#' '#' '#']]
```

   Hint: You can use `np.full((nrows, ncols), "#", dtype=str)` to create an empty array with of *nrows* rows and *ncols* cols, containing only `'#'` characters. Idea: create a larger array than you need, and then overwrite the values in the middle with `image_array`.

   In `exam_test.py`, test your function by calling it on `smiley_image_array` and setting `frame_width` to 2. Save the result in a variable called `smiley_image_array_framed`.

d. We'll now consider the `smiley_image_array_labels` array more carefully. If you print it you will notice that it has exactly the same shape as `smiley_image_array`. This file contains the labels for the image: for each entry in the array, it specifies what this entry constitutes in the original image (e.g. `"MOUTH"`).

In the `exam.py` file, write a function called `make_image_array_happy` that takes two arguments: `image_array` (a numpy array like our `smiley_image_array`) and `label_array` (a numpy array like `smiley_image_array_labels`). The function should return a new array (i.e. a copy) of the same size as smiley, but which is happy instead of sad (it should look like the image above). Hint: use at the MOUTH_UPPER and MOUTH_LOWER labels.

In `exam_test.py`, test your `make_image_array_happy` function by calling it using `smiley_image_array` and `smiley_image_array_labels` as arguments.

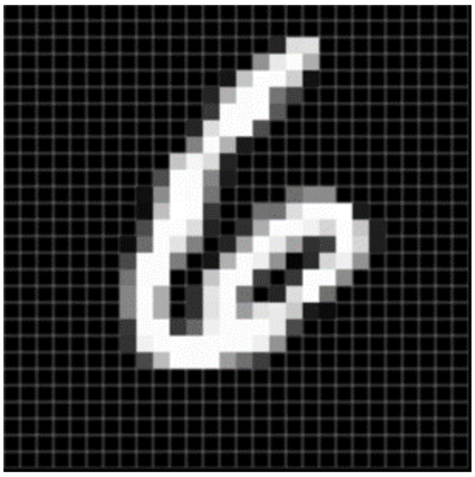
Q5

In handin5, we considered the `mnist_test_200.csv` file. We'll work on that again here, but now using Pandas. Start by downloading it from **https://wouterboomsma.github.io/ppds2022/data/mnist_test_200.csv (https://wouterboomsma.github.io/ppds2022/data/mnist_test_200.csv)**.

The `mnist_test_200.csv` file contains images 200 from MNIST. We can read it in as a pandas dataframe using the following code:

```
data = (pd.DataFrame(np.genfromtxt("mnist_test_200.csv",
                                   skip_header=1,
                                   delimiter=',')).astype(int).set_index(0)>(255/2)).astype(int)
```

Each row in the resulting dataframe will contain 784 pixel values (each being zero or one), corresponding to the 28x28 pixels in each image. As we saw in handin5, the images look something like this:



The index of the pandas dataframe contains the label of the image (which digit is shown in the image). For the example above the label is `6`.

a. In the `exam.py` file, write a class called `MnistCollection`. The constructor of the class should take a filename as argument and should initialize a attribute called data, using something similar to the code provided above.

   In `exam_test.py`, instantiate your class using `mnist_test_200.csv` as argument, and save the object in a variable called `mnist_collection`. Check that the object has a `data` attribute.

b. Add a method called `get_images_for_digit` to your class. In addition to `self`, the method should take a single argument called `digit`. The function should extract all images labeled to contain that specific digit (e.g. all images corresponding to the digit `7`). It should return the images as a numpy array with shape `(n, 28, 28)`, where `n` is the number of images for that label. **You must use the `groupby` method in pandas to solve this task.**

   In `exam_test.py`, test your `get_images_for_digit` method by calling it for `digit=7`. Save the result in a variable called `images_of_seven`. Save *the first* of the images in a variable called `images_of_seven_first`. Print this variable to screen to see what it looks like.

   To make this printing slightly nicer, add the following function to your `exam.py` file:

```
def mnist_pretty_print(mnist_image_array):
    '''Return a nicer string representation of a binary MNIST image'''
    return np.array2string(mnist_image_array, separator=' ',
                           formatter={'int_kind': lambda x: ' ' if x==0 else '#'})
```

In `exam_test.py`, call this function on the `images_of_seven_first` and save the result in a variable called `images_of_seven_first_pretty`. Print this variable to screen to see what it looks like.

c. Add a method called `get_slimmest_image_for_digit` to your class. In addition to `self`, the method should take a single argument called `digit`. This function should work similarly to `get_images_for_digit`, but instead of returning all images for a digit, it should return only the one which has the fewest pixels set to 1 (i.e for which the sum over the 784 values is the smallest). You should call the `get_images_for_digit` function inside `get_slimmest_image_for_digit`. The function should return a numpy array with shape `(28x28)`.

   In `exam_test.py`, call the `get_slimmest_image_for_digit` method using `digit=7`. Save the result in a variable called `slimmest_seven`.