Exam S

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 upload preliminary versions during the exam (just don't "finalize" until you are ready to hand in).

Is the CodeGrade 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 "Exam" - 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. Some 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?

No. Apart from Digital Exam (eksamen.ku.dk), Absalon (absalon.ku.dk) and CodeGrade (app.codegra.de), you are not allowed to access any internet service during the exam. In particular, use of generative AI services like ChatGPT and Copilot are prohibited (even when used offline).

What material am I allowed to access?

Other than access to the internet, this is an "all aids allowed" exam, so you are allowed to use any other material, such as your own handins, the slides, physical books, ebooks, etc. Prior to the exam, we have provided PDF reference manuals for the Python library, Numpy and Pandas, which should contain basic information about the modules/functions/methods required for this exam.

How should I structure my time?

Note that the Q1, Q2, Q3, and Q4 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.

Q1 (loops)

a. Inside `exam.py`, write a function called `create_number_string` that can create strings like this `"1 2 3 4 5 6 7"`.

   The function should take two arguments: `min` and `max`, specifying the lowest and highest number (in the case above `min=1` and `max=7`), and it should return the produced string.

   Inside `exam_test.py`, test your function by calling it as `create_number_string(3, 13)`

b. Inside `exam.py`, write a function called `create_hash_string` which takes a single argument: `length`. The function should return strings like this **"# # # # # # #"**, where `length` sets the length of the string (in this case `length=13`; if it had been 12 it would have ended in a space).

   Inside `exam_test.py`, test your function by calling it as `create_hash_string(11)`.

c. Inside `exam.py`, write a function called `generate_concentric_squares`. The function should take a single argument: `size`. It should check that this size is an odd number and raise a `ValueError` exception if this is not the case. The function should return a square numpy array of this size (i.e. `size` specifies both the number of rows and columns), and initialize it with `' '` and `'#'` such that the `'#'` characters form squares centered around the middle. In the following, we've illustrated the expected result for the first 5 odd values of `size` (but note that the function should work for any `size` larger than 0):

```
size=1   size=3            size=5                   size=7                           size=9
[['#']]  [['#' '#' '#']    [['#' '#' '#' '#' '#']   [['#' '#' '#' '#' '#' '#' '#']   [['#' '#' '#' '#' '#' '#' '#' '#' '#']
          ['#' ' ' '#']     ['#' ' ' ' ' ' ' '#']    ['#' ' ' ' ' ' ' ' ' ' ' '#']    ['#' ' ' ' ' ' ' ' ' ' ' ' ' ' ' '#']
          ['#' '#' '#']]    ['#' ' ' '#' ' ' '#']    ['#' ' ' '#' '#' '#' ' ' '#']    ['#' ' ' '#' '#' '#' '#' '#' ' ' '#']
                            ['#' ' ' ' ' ' ' '#']    ['#' ' ' '#' ' ' '#' ' ' '#']    ['#' ' ' '#' ' ' ' ' ' ' '#' ' ' '#']
                            ['#' '#' '#' '#' '#']]   ['#' ' ' '#' '#' '#' ' ' '#']    ['#' ' ' '#' ' ' '#' ' ' '#' ' ' '#']
                                                     ['#' ' ' ' ' ' ' ' ' ' ' '#']    ['#' ' ' '#' ' ' ' ' ' ' '#' ' ' '#']
                                                     ['#' '#' '#' '#' '#' '#' '#']]   ['#' ' ' '#' '#' '#' '#' '#' ' ' '#']
                                                                                      ['#' ' ' ' ' ' ' ' ' ' ' ' ' ' ' '#']
                                                                                      ['#' '#' '#' '#' '#' '#' '#' '#' '#']]
```

   Hint: note that along the diagonal, we have '#' at every 2nd position. Iterate over all the places where you want a top-left '#' corner, and for each of them create a square using 4 slice operations (top-left to top-right, top-left to bottom-left, top-right to bottom-right, and bottom-left to bottom-right).

   Inside `exam_test.py`, test your function by calling it as `generate_concentric_squares(13)`.

Q2 (regular expressions)

In this exercise, we will try to process the text in the following variable:

```
jazz_musicians_data = """Name: Lester Young, Active:1933-1959. Instrument: saxophone
name: Dizzy Gillespie, active: 1935-1993. Instrument:trumpet
name: Billie Holiday, Active: 1930-1959. Instrument: vocalist
Name: Dexter Gordon. Active: 1940-1986. Instrument: saxophone
name: Duke Ellington; active: 1914-1974.  instrument: piano
name Oscar Peterson.Active:1945-2007.Instrument: piano
"""
```

Start by copying the variable above into your `exam_test.py` file. We expect you to use Python's `re` module for this part of the exam.

a. First, we will create a regular expression to match the individual lines. Inside `exam.py`, write a function called `create_jazz_regexp` that takes no arguments. The function should compile a regular expression that matches each of the lines above, such that the name, active-start-year, active-end-year, and instrument can be extracted as groups 1,2, 3 and 4, respectively. It should return this regular expression object.

   Inside `exam_test.py`, test your function by calling it as `create_jazz_regexp()`. Save the return value in a variable called `jazz_regexp`.

b. In exam.py, create a function called `parse_jazz_data`. The function should take two arguments: `regexp` and `text`. The function should iterate over the lines in text (hint: you will need to split it first), and attempt to match them with `regexp`. If successful, it should extract the values corresponding to the four groups. The function should return a list of length-4 tuples, where the tuples contain the name, the active-start-year (integer), the active-end-year (integer) and the instrument.

   In `exam_test.py`, test your function by calling `parse_jazz_data` on the `jazz_regexp` returned from the previous exercise, and on the `jazz_musicians_data` variable.

c. In `exam.py`, create a function called `parse_jazz_data_sorted`. Like before, the function should take two arguments: `regexp` and `text`. The function should start by calling the `parse_jazz_data` function to get a list of 4-tuples. It should then return a new version of this list, such that the entries are sorted by how many years each musician was active (the musician who was active for most years should appear last). Hint: use the `key` argument of the `sorted` function.

In `exam_test.py`, test your function in the same way as for `parse_jazz_data`.

Q3 (reading data into data structures)

In the final two questions, we will be working on a global temperature dataset extracted from this website: https://climate.copernicus.eu/july-2023-sees-multiple-global-temperature-records-broken.

The `era5_daily_global_sfc_temp_1940-2023.csv` file is provided to you as part of the exam. Download the file, and place it in the same directory as your `exam.py` and `exam_test.py` files. The file contains montly temperature observations averaged over the whole globe. Start by looking at the file to see the format of the data.

This part of the exam should be done in pure python - no modules are allowed.

a. Inside `exam.py`, write a function called `read_data` that takes a single argument: `data_filename`. The function should open the file and read its content into a list of strings, ignoring all comment lines. Note that the header line (the first line after the comment lines) should be included in the list as the first element.

   Inside `exam_test.py`, test your function by calling it as: `read_data('era5_daily_global_sfc_temp_1940-2023.csv')`. Save the result in a variable called `temp_data1`.

b. Inside `exam.py`, write a function called `read_data2` that takes as single argument: `data_filename`. The function should open the file and iterate over the lines in the same way as the `read_data` function from the previous question (feel free to copy&paste), but it should now read the header line in separately. It should then save the data lines into a list of dictionaries, where each dictionary contains the different column fields on each line as values. For instance, the dictionary for the first data-containing line in the file should be

   ```
   {'date':(1940,1,1), 'temp':11.7181, 'status': 'FINAL'}
   ```

   Please pay attention to the exact format of the date field (tuple of ints) and the temp field (float). Also note that the keys in these dictionaries should be extracted from the header line.

   Inside `exam_test.py`, test your function by calling it as: `read_data2('era5_daily_global_sfc_temp_1940-2023.csv')`. Save the result in a variable called `temp_data2`.

c. Inside `exam.py`, write a function called `read_data3` that takes a single argument: `data_filename`. Again, the function should read in the file (feel free to copy from `read_data2`), but this time store the data into a dictionary of dictionaries of dictionaries, such that we can write `data[year][month][day]` to get the relevant temperature value as a float (we will ignore the status column). Hint: for a given year and month value, you will need to check if these are present as keys in the dictionary, and add them (with an empty dictionary as value) if they are not.

   Inside `exam_test.py`, test your function by calling it as: `read_data3('era5_daily_global_sfc_temp_1940-2023.csv')`. Save the result in a variable called `temp_data3`.

Q4 (pandas)

In this final part, we'll work with the same data as in Q3, but now using pandas.

a. Inside `exam.py`, write a function called `read_data_pandas` that takes as single argument: `data_filename`. The function should use the built-in support in pandas to read in the file, and return it as a pandas dataframe (although pandas has support for dates, we will read the date column as simple strings).

   Inside `exam_test.py`, test your function by calling it as `read_data_pandas('era5_daily_global_sfc_temp_1940-2023.csv')`. Save the result in a variable called `data_df`.

b. Inside `exam.py`, write a function called `count_entries_per_year` that takes a single argument: `data_df`. The function should use the pandas `groupby` functionality to group entries in the dataframe by the year values. Hint: use the fact that the year values are always the first 4 characters in the `'date'` column. The function should return a pandas Series, which as index has the years (just keep them as strings), and as values has the counts for how many temperature observations there were for that year.

Inside `exam_test.py` test your function by calling it as `count_entries_per_year(data_df)`.

c. Inside `exam.py`, write a function called `create_pivot_table` that takes a single argument: `data_df`. Just as in the previous question, start by using pandas' `groupby` functionality to group entries in the dataframe by the year values. Now, write a for-loop that iterates over the groups. Within the loop:
   1. Remove the year from the date columns of the group so that dates are now in the format MM-DD (hint: use the `.str` accessor)
   2. Rename the 'temp' column of the group to be the value of the current year (hint: `df.rename`)
   3. Remove the 'status' column (hint: `df.drop`)
   4. Set the index in the group to be the 'date' column
   5. For all groups that have only 365 entries, add an entry for `'02-29'` containing an empty value (hint: use `pd.Series`).

Note that inside the loop, the group value will work just like a normal dataframe. Hint: For many of the steps above, you can use an `inplace=True` argument to make pandas change the current dataframe, instead of creating a new one. If you don't use this, you will have to remember to save the return value in a new variable each time.

Finally, save all the group dataframes to a list, and outside the loop, use `pd.concat` to merge them into a big dataframe, where the index is the date strings, and the columns are the temperature values for the individual years. It should look similar to this:

```
        1940    1941    1942    ...
date
01-01   11.7181 11.8165 11.5658 ...
01-02   11.6620 11.8078 11.5172 ...
...
```

Inside `exam_test.py`, test that your function is working by calling it as `create_pivot_table(data_df)`. Save the result in a variable called `pivoted_df`.

If you have done this correctly, you should be able to call the following provided function on `pivoted_df` to get a plot of the temperatures for each year.

```
def plot_temperatures(pivoted_df):
    """Plot temperature values for each year in dataframe"""

    # Importing here in function to make function self-contained
    import matplotlib
    import matplotlib.pyplot as plt

    # Ensure that dates and years are sorted correctly
    pivoted_df.sort_index(axis='index', inplace=True)
    pivoted_df.sort_index(axis='columns', inplace=True)

    # Initialize plot
    fig, ax = plt.subplots(figsize=(12, 4))

    # Setup line colors
    years = pivoted_df.columns
    norm = matplotlib.colors.Normalize(vmin=0, vmax=len(years))
    matplotlib.cm.OrRd(norm(1))
    colors = [matplotlib.cm.Greys(norm(idx)) for idx in range(len(pivoted_df.columns)-1)] + ['red']
    lw = [1] * (len(colors)-1) + [3]
    from cycler import cycler
    ax.set_prop_cycle(cycler(color=colors, lw=lw))

    # Create plot
    pivoted_df.plot(ax=ax)

    # Set tick marks to Month values
    labels = pivoted_df.reset_index()['date'][pivoted_df.reset_index()['date'].str[3:] == '01']
    ax.set_xticks(labels.index, ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])
    ax.set_ylabel("temperature")

    # create legend
    box = ax.get_position()
    ax.set_position([box.x0, box.y0, box.width * 0.8, box.height])
    # Put the legend to the right of the current axis
    ax.legend(loc='center left', bbox_to_anchor=(1, 0.5), ncol=6)

    fig.show()
```