Exam

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.

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 "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. 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.

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`. If we agree that something should be clarified, we will post an Absalon announcement with a clarification.

Q1

a. A prime number is a number that is only divisible by 1 and the number itself. Inside `exam.py`, write a function called `is_prime`, that takes a single argument, called `query_number`, and returns a boolean value indicating whether `query_number` is a prime (True) or not (False). Hint: for any number `x`, you can determine if `x` is a prime by iterating over all numbers between 1 and `x`, and checking whether x is divisible by this number (i.e. whether you can divide x by this number with zero remainder). Note that negative numbers, zero and one are not primes. **You should use a Python loop to solve this task (no list comprehension).**

In the `exam_test.py` file, test your function by calling it on the value `10` (expected result: `False`), and `11` (expected result: `True`). Save the results in variables `check_prime_1` and `check_prime_2`, respectively.

b. Now, in the `exam.py` file, write a function called `prime_range`, which takes two arguments: `lower`, and `upper`. This function should iterate over the numbers from `lower` (included) to `upper` (excluded), and return a list containing only those number which are prime numbers. This function should use the `is_prime` function from the previous question. If your `is_prime` function is not working, you can still solve this question by showing how you would call the function if it was working. **You should use a Python loop to solve this task (no list comprehension).**

In the `exam_test.py` file, test your function by printing all primes between 0 (included) and 42 (excluded). Save the result in a variable called `primes1`.

c. Finally, write a function called `prime_range_one_liner`, which takes two arguments, `lower`, and `upper`. The function should do the same as in the previous question, but now using list-comprehension. The function should (apart from a doc-string) contain just one-line of code, including the return statement.

   In the `exam_test.py` file, test your `prime_range_one_liner` function by printing all primes between 0 (included) and 42 (excluded). Save the result in a variable called `primes2`.

Q2

In these questions, you are expected to use functionality from the regular expression module: `re`.

a. Danish CPR numbers consist of a six digit date (DDMMYY), followed by a 4 digit identifier (IIII). Inside the `exam.py` file create a variable called `cpr_regexp`. The value of this variable should be a Pattern object from the `re` module (i.e. a regular expression object), which matches any CPR number, either in the DDMMYYIIII format, or in the DDMMYY-IIII format. The regular expression should contain four groups, such that the DD, MM, YY, and IIII parts can be extracted after matching.

   In your exam_test.py file, test the code by writing:
   `print(exam.cpr_regexp.match("123456-7890"))`

b. The 4-digit identifier - together with the last two digits of the date (i.e. the year), encodes in which century a person is born, using the following system

IIII	YY	Born in century
0001-3999	00-99	1900
4000-4999	00-36	2000
4000-4999	37-99	1900
5000-8999	00-57	2000
5000-8999	58-99	1800
9000-9999	00-36	2000
9000-9999	37-99	1900

   Note that in this table, both numbers in a range are included (e.g. 00-99 includes both 00 and 99).

   Inside your `exam.py` file, write a function called `cpr_century` that takes a single argument

called `cpr_number_str` as input (a string). Using the `cpr_regexp` regular expression that you defined in the previous question, the function should first assert that `cpr_number_str` is a legal CPR number (according to the regular expression). It should then determine the relevant century based on the information in the table above, and return either `1800`, `1900` or `2000` (as an integer).

In the `exam_test.py` file, test that your code works by calling the `cpr_century` on the CPR number "111111-1118". Save the result in a variable called `century`. If you implemented your function correctly, it is expected to raise an AssertionError if it is called on a string that is not a CPR number (e.g. "hello").

Q3

a. The following piece of code will generate a list of lists of numbers (floats):

```
import random
data = [ [random.random() for j in range(2)] for i in range(20) ]
```

Put this code in your `exam_test.py` file. If you inspect the `data` variable, you will notice that the outer list contains 20 values, while the inner lists contain 2 elements each. In other words, you can interpret this data as having 20 rows, and two columns. If I wish to extract the element in the 8th row, in the 2nd column, I would write `data[7][1]`.
Inside the `exam.py` file, create a function called `calc_column_averages`, which takes a single argument called `data_list_of_lists`, which we assume will be a list of lists like the `data` variable above. You can assume that all the inner lists are of size 2. The function should calculate the averages of the two columns - and return this as a list of two numbers. **This exercise must be solved using standard for-loops (no list comprehension), and without use of external modules**. Hint: you can calculate the average by first summing up all the values for each of the columns, and then dividing by the number of elements in each column.

In the `exam_test.py` file, test the `calc_column_averages` function by calling it on the `data` variable, and save the result in a variable called `column_averages`.

b. We now wish to create a new list of lists, but where the rows and columns have been interchanged. Inside the `exam.py` file, create a function called `transpose_list_of_lists`, which takes a single argument called `data_list_of_lists`, which we assume will be a list of lists like the `data` variable above (again, you are allowed to assume that the inner lists have length 2). The goal of the function is to *transpose* the list, which means that we interchange the outer list and the inner list. For the specific example above, this means that the outer list should now contains 2 elements, and the inner lists contain 20 elements each. The corresponding interpretation is that we will have a list for each column, each containing 20 row values. If I wanted to extract the same element as before in this new list, I would now access it using [1][7] instead of [7][1]. Again, **this exercise must be solved using standard for-loops (no list comprehension), and without use of external modules.**

In the `exam_test.py` file, test your function by calling it on the `data` variable and save the result in a variable called `data_transposed`.

Q4

In this last exercise, we will use the `Land_and_Ocean_summary.txt` file that we also worked on in the handins. Download the file from: `https://wouterboomsma.github.io/ppds2022/data/Land_and_Ocean_summary.txt` (or use an existing copy if you already have it on your computer). You are expected to use `pandas` for this exercise. Start by reading in the data by putting the following code in your `exam_test.py` file:

```
import pandas as pd
anomaly_df = pd.read_table('Land_and_Ocean_summary.txt', comment="%",
                           delimiter=r"\s+", index_col=0, usecols=range(5),
                           names=['Year', 'Annual Anomaly', 'Annual Unc.',
                                  'Five-year Anomaly', 'Five-year Unc.'])
```

a. We'll start by calculating averages of the columns in this file - using pandas. In the `exam.py` file, write a function called `calc_column_averages_pandas` that takes a single argument `anomaly_df` (a dataframe like the one above), and returns a pandas Series object, where the values are the averages for each column, and the index of the series are the 4 column labels from the input DataFrame (i.e. 'Annual Anomaly', 'Annual Unc.', 'Five-year Anomaly', and 'Five-year Unc.').

   In `exam_test.py`, call the `calc_column_averages_pandas` function on `anomaly_df`, and save the result in a variable called `column_averages_series`.

b. Now, let's say we are interested only in those years where the temperature is higher than the reference. Inside `exam.py`, write a function called `get_positive_anomalies` that takes a single argument `anomaly_df` (a dataframe like the one defined above). The function should return a new DataFrame consisting only of those rows in the original DataFrame for which `'Annual Anomaly'` is larger than zero.

   In `exam_test.py` call your `get_positive_anomalies` function on `anomaly_df` and save the result in a variable called `positive_anomalies`.

c. Let us consider the uncertainty column ('Annual Unc.'). In particular, we will try to identify the years for which the uncerainty is large. In the `exam.py` file, create a function called `remove_uncertain_anomalies`, that takes a single argument, `anomaly_df` (a dataframe like the one above). The function should return a new dataframe that only contains rows for which the `'Annual Unc.'` value is smaller than 10% of the absolute value of the `'Annual Anomaly'` column. For example, if the `'Annual Anomaly'` value is `-2.0`, we would consider anything larger than `0.2` a "large" uncertainty.

   In the `exam_test.py` file, test your function by calling it on the `anomaly_df` dataframe.

Save the result in a variable called `anomaly_df_filtered`.

d. Instead of removing the rows with high uncertainty as we did in the previous question, we could also decide to set the 'Annual Anomaly' for these values to NaN, so we can subsequently process them. In your `exam.py` file, write a function called `set_uncertain_anomalies_to_nan` that takes a single argument, `anomaly_df` (a dataframe like the one above). The function should return a new dataframe where all rows with high uncertainty (i.e. where 'Annual Unc.' is larger or equal than 10% of the value in 'Annual Anomaly'), have their 'Annual Anomaly' set to `float('nan')`.

   In the `exam_test.py` file, test your `set_uncertain_anomalies_to_nan` function by calling it on the `anomaly_df` dataframe. Save the result in a variable called `anomaly_df_w_nans`.

Q5

In this last exercise, we will create a class that simulates tossing a coin.

a. In your `exam.py` file, create a class called `Coin`. The class should have a constructor which takes a single argument called `tails_probability`, and saves it as an attribute in `self`. If the `tails_probability` argument is not given, it should default to 0.5.

   In the exam_test.py file, write the following code: `coin = exam.Coin()`. If you wrote the code correctly, this coin object should have an attribute called `tails_probability`.

b. Now, in the `exam.py` file, inside the `Coin` class, add a method called `toss`. This method should take no arguments (other than `self`). When calling the method, the function should call the `random` function from the `random` module to generate a random number between 0 and 1, compare this value to `tails_probability`, and then return either `"HEADS"` or `"TAILS"`.

   In the `exam_test.py` file, create a new `Coin` object called `biased_coin`, which has zero chance of producing `"TAILS"`. Call the `toss` method 3 times, printing the result for each one. We expect all three to be "HEADS".

c. Let's try to construct a better way to see the results of multiple tosses. In `exam.py`, inside the `Coin` class, add a method called `toss_multiple`. This method should take a single argument, `n_tosses`, which specifies how many times the coin will be tossed. The function should contain a for-loop that calls the `toss` method `n_tosses` times. As its result, the function should return a dictionary with two keys "HEADS" and "TAILS", where the values in the dictionary show how many heads and tails were produced. For instance, if `n_tosses` is 10, and 4 of those were "TAILS", the result should be {"TAILS":4, "HEADS": 6}.

In the `exam_test.py` file, call the `toss_multiple` method on your `biased_coin` object with `n_tosses` set to 1000. Save the result to a variable called `toss_statistics`.