Assignment 1

Due: 05. November 2023, 23:59 CET

General information: This assignment should be completed in groups of **three to four people**. Submit your assignment via the corresponding Moodle activity. You can only upload one .zip file, therefore make sure to place all your files required to run/evaluate your solution into **one archive** and just upload that archive.

Group information: You can determine your own groups. The Moodle contains an activity to select your group which you should do to find your groups before trying to submit your first assignment. The first tutorial sessions can also be used to find group members.

Python exercise information: You are only allowed to use the external libraries that we provide or explicitly mention on the assignment sheet (or that are already within the provided skeleton files). Your solutions' functionality will all be tested against automated tests before being reviewed further. **You need to ensure** that your solution can directly be **imported as a Python3 module** for automatic tests and at least contains the name of the skeleton file. The easiest way to achieve this, is to directly develop your solution in the assignment1.py file and submit that. The Moodle contains information about the test environment. You will also find an example test-file alongside the skeleton file that you can use to ensure that your solution works in the evaluation environment.

Exercise 1: (7 Points)

In order to familiarize yourself with Python and the important package NumPy, implement the following functions.

Task 1: Write a function *fibonacci(n)* that returns the corresponding n's Fibonacci number. fibonacci(1) should return 1, while fibonacci(6) should return 8 etc. (see `https://en.wikipedia.org/wiki/Fibonacci_number`).

Task 2: Write a function *random_array(size, min_val=0, max_val=1)* which returns a NumPy array of the given size containing random numbers in the interval given by the default arguments min_val und max_val.

Task 3: Write a function *analyze(array)* which **prints** out the minimum, maximum, mean and median values of the provided array **and returns a dictionary** containing these values under appropriate keys.

Task 4: Write a function *histogram(array, bins=10)*, which plots the given array into a histogram with the specified number of bins. Have a look at matplotlibs *hist* method for this.

Task 5: Write a function *list_ends(original_list)*, which creates a new list that only contains the first and last item of the original list.

Task 6: (1 Point) Write a function *combine_dictionaries*, which combines two given dictionaries into one. Values from shared keys should be added.

Task 7: (1 Point) Write a function *matrix_mul(matrix_a, matrix_b)* that checks the dimensions of the matrices. If they align, i.e. matrix_a has dimensionality m x n and matrix_b n x o, the function should return the resulting matrix_a * matrix_b, otherwise it should raise an *AttributeError* exception with an appropriate error message.

Exercise 2: (13 Points)

Since the lecture focuses on reasoning and decision making with graphical probabilistic models, we will have to deal with different types of graphs, graph datastructures, and graph algorithms in subsequent exercises. Here, we will have a first look at *directed graphs*.

Complete the provided skeleton class for a simple datastructure to represent directed graphs. The graph class should at least provide the following functions (the *self* parameter is omitted for better readability):

Task 1: (1 Point) *add_node(node)*: Add a node to the graph. It is up to you what type the parameter should take.

Task 2: (1 Point) *remove_node(node)*: Remove the specified node from the graph.

Task 3: (1 Point) *add_edge(node_a, node_b)*: Add an edge pointing from node_a to node_b.

Task 4: (1 Point) *remove_edge(node_a, node_b)*: Remove the specified edge.

Task 5: (1 Point) *get_number_of_nodes()*: Returns the total number of nodes in the graph. You may also implement this function as a **property**.

Task 6: (1 Point) *get_parents(node)*: Returns a list of all parent nodes of the given node.

Task 7: (1 Point) *get_children(node)*: Returns a list of all children nodes of the given node.

Task 8: (2 Points) *is_ancestor(node_a, node_b)*: Returns True if node_a is an ancestor of node_b.

Task 9: (2 Points) *is_descendant(node_a, node_b)*: Returns True if node_a is a descendant of node_b.

Task 10: (2 Points) *is_acyclic()*: Returns True if the graph is acyclic.

You are free to implement additional helping functions or other classes (e.g. for nodes or edges), however you are not required to do so.