# Exam 2023 Fall

| | |
|---|---|
| **Course name:** | Computer programming |
| **Course number:** | 02002 and 02003 |
| **Exam date:** | 6th of December 2023 |
| **Aids allowed:** | All aids, no internet |
| **Exam duration:** | 4 hours |
| **Weighting:** | All tasks have equal weight |
| **Number of tasks:** | 10 |
| **Number of pages:** | 13 |

## Contents

# Exam Instructions

## Prerequisites

To be able to solve the exam tasks, you need to have a computer with Python installed. All exam problems can be solved in either IDLE or VS Code.

## Exam Material

The exam material consists of a single zip file. You should unzip this file to a folder on your computer. The zip file contains the exam text as a PDF document in English `2023_12_exam_English.pdf` (this document) and the same document in Danish `2023_12_exam_Danish.pdf`. The zip file also contains a folder `2023_12_exam` with the following files and folders:

- An empty Python file for each task, `<y>.py`, where `<y>` is the name of the task. These are the files where you should write your solutions and submitted them at the end of the exam.

- A Python file for each task, `test_task_<x>_<y>.py`, where `<x>` is the task number, and `<y>` is the name of the task. These contain code that checks if your solution has the correct behavior for the example in the exam text. To be sure that you use test as intended, do not edit these files.

- A Python file `test_tasks_all.py` that runs all test files.

- A folder `files/` containing data files needed to run and test tasks involving files, if any.

## Solving Exam Tasks

If you are using VS Code, you should start by going to `File` → `Open Folder..` and choose the `2023_12_exam` folder inside the folder you unzipped to above.

When solving the exam tasks, follow the instructions in the exam text. You can test your solutions by running the provided testing scripts. Your solutions must be in the same folder as the testing scripts, for the testing scripts to work.

If you believe there is a mistake or ambiguity in the text, you should use the most reasonable interpretation of the text to solve the task to the best of your ability. If we, after the exam, find inconsistencies in one or more tasks, this will be taken into account in the assessment.

Your solutions should only use the tools that have been taught in the course. Solutions that import modules other than `math`, `numpy`, `os`, or `matplotlib` will not be graded. The test scripts provided do not check for this, so it is your responsibility to ensure that your solutions only use the allowed modules.

## Evaluation of the Exam

We will run a number additional tests on each of your solutions that checks if it behaves as specified in the task. The fraction of correct tests is the score for each task. The overall score is the average of the scores.

A solution where the provided test fails is incorrect. This can be because the file or function are named incorrectly. However, if a provided test passes, it does not guarantee that the solution is correct for our additional tests.

## Handing in

To hand in your solutions, upload your Python files with solutions to the [digital exam system](). In the digital exam system, files can be submitted as either *main document* or *attachments*. You can upload any of your solutions as the main document, and the rest as attachments.

You should hand in exactly the following files:

- `arrival_times.py`
- `bank_account.py`
- `checkerboard_sum.py`
- `collatz_conjecture.py`
- `event_probability.py`
- `nitrate_levels.py`
- `phonebook_merge.py`
- `punctuation_ratio.py`
- `special_occurrence.py`

Any file handed in that is not in the list above will not be taken into account in your assessment.

# Task 1: Event Probability

When describing extreme events, such as major earthquakes, landslides, and floods, we utilize the concept of a *return period* $T$, given in years. For example, a flood with a return period of 100 years, referred to as a *100-year flood*, is a flood that has a probability of $\frac{1}{100}$ of occurring in any given year. The probability that an event with a return period $T$ will occur within a time period of $n$ years can be expressed as

$$P = 1 - \left(1 - \frac{1}{T}\right)^n.$$

You should write a function that takes as input two numbers: the return period (in years) and the time period (also in years). The function should return the probability of an event with a return period occurring during the given time period.

As an example, consider a return period of 100 years and the time period of 25 years. The probability that the 100-year event will occur in a period of 25 years is (displayed with 7 decimal places)

$$P = 1 - \left(1 - \frac{1}{100}\right)^{25} = 0.2221786$$

which is what the function should return, as shown in the code cell below.

```
>>> event_probability(100, 25)
0.22217864060085335
```

The filename and requirements are in the box below:

event_probability.py

`event_probability(T, n)`

Return the event probability.

*Parameters*:

- `T`  `int`  A positive integer, the return period.

- `n`  `int`  A positive integer, the time period.

*Returns*:

- `float`  The probability that an event with return period T will occur in the time period.

# Task 2: Arrival Times

Given a list of scheduled train arrivals (hours and minutes) and a delay in minutes, we need to determine the expected arrival times. The scheduled times are given as a list of strings. Each time is formatted as **hh:mm** for a 24-hour display. Here, **hh** is the number of hours between 00 and 23 written using two digits, while **mm** is the number of minutes between 00 and 59 written using two digits. Expected arrival times need to be formatted in the same way.

Write a function that takes as input a list of scheduled arrivals and a delay in minutes. The list may contain an arbitrary number of scheduled arrivals, but the delay is the same for all arrivals. The function should return a list of strings with expected arrival times formatted as **hh:mm** in 24-hour time notation with two digits for both hours and minutes. Remember to handle the case when the delay causes the arrival to be postponed until the next day.

As an example, consider the list `['12:37', '08:10']` and a delay of 25 minutes. The first scheduled arrival is **12:37**. However, with a 25-minute delay, the expected arrival is **13:02**. The second scheduled arrival is **08:10**, but with a 25-minute delay, the expected arrival is **08:35**. You can see the expected output in the code cell below.

```
>>> arrival_times(['12:37', '08:10'], 25)
['13:02', '08:35']
```

The filename and requirements are in the box below:

arrival_times.py

`arrival_times(schedule, delay)`

Return the arrival times given scheduled times and delay.

*Parameters*:

- `schedule`  `list`   A list of strings, the scheduled times.

- `delay`   `int`   A positive integer, the delay (in minutes).

*Returns*:

- `list`   The arrival times, a list of strings.

# Task 3: Special Occurrence

Given a sequence of positive integers, we want to find what we call a *special occurrence*. A special occurrence is when the number **5** is followed by two numbers where *exactly* one is **7**. Thus the occurrence **…5, 3, 7…** is a special occurrence, and so is the occurrence **…5, 7, 8…**, while **…5, 7, 7…** is *not* a special occurrence.

Write a function that takes as input a list of positive integers. The function should return the index of the number **5** in the first special occurrence. If no such occurrence exists, the function should return **-1**.

As an example, consider the sequence `[2, 8, 11, 3, 12, 5, 7, 7, 11, 3, 12, 5, 2, 7, 5, 7, 2, 6]`. The number **5** occurs three times in the sequence, at positions with index **5**, **11**, and **14**. The first occurrence of the number **5** is not a special occurrence as it is followed by *two* **7**. The second occurrence is a special occurrence as it is followed by **2** and **7**. The third occurrence is a special occurrence, but it occurs later than the second occurrence. Therefore, the function should return **11**, as shown below.

```
>>> special_occurrence([2, 8, 11, 3, 12, 5, 7, 7, 11, 3, 12, 5, 2, 7, 5, 7, 2, 6])
11
```

The filename and requirements are in the box below:

---

`special_occurrence.py`

`special_occurrence(sequence)`

Return the index of the first special occurrence.

*Parameters*:

- `sequence`  `list`   A list of positive integers with 0 or more elements.

*Returns*:

- `int`   The index of the first 5 followed by two numbers where exactly one is 7.
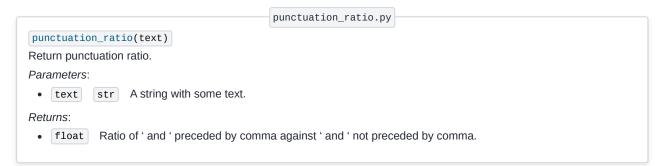
---

# Task 4: Punctuation Ratio

We would like to collect statistics about using commas in connection with the word **and**. Therefore, given a text, we first want to identify all occurrences of the lower-case word **and** between two spaces, that is the string `' and '`. Then, we want to calculate the ratio of the cases where a comma immediately precedes `' and '` against the cases without the comma (that is, any other character immediately precedes `' and '`). The ratio should be given as

$$\text{ratio} = \frac{\text{number of cases with a comma before ' and '}}{\text{number of cases without a comma before ' and '}}.$$

Write a function that takes a string with text as input. The function should return a number giving the ratio of occurrences of `' and '` preceded by a comma against the occurrences of `' and '` not preceded by a comma. You can assume that the text does not start with `' and '`. If either the numerator or the denominator is zero, the function should return **0**.

Consider the text with all seven occurrences of `' and '` highlighted.

> Sara **and** Emma like to travel, bike, **and** hike, **and** when they are traveling they always take their bikes, hiking shoes, **and** sleeping bags. Last year, Sarah **and** Emma traveled to Italy, France, **and** Spain. And that was fun, and, according to Sara **and** Emma, very expensive!

The string `' and '` is preceded by a comma four times (highlighted in blue), while it is not preceded by a comma three times (highlighted in orange). The ratio we should compute is therefore 4/3, and this is what your function should return, as seen in the code box below. Note that `' And '` and `' and, '` are not counted, as we only consider lower-case word **and** between two spaces.

```
>>> text = ("Sara and Emma like to travel, bike, and hike, and when they are " +
...     "traveling they always take their bikes, hiking shoes, and sleeping bags. " +
...     "Last year, Sarah and Emma traveled to Italy, France, and Spain. And that " +
...     "was fun, and, according to Sara and Emma, very expensive!")
>>> punctuation_ratio(text)
1.3333333333333333
```

The filename and requirements are in the box below:

---
punctuation_ratio.py
---

`punctuation_ratio(text)`

Return punctuation ratio.

*Parameters*:

- `text` `str`  A string with some text.

*Returns*:

- `float`  Ratio of ' and ' preceded by comma against ' and ' not preceded by comma.
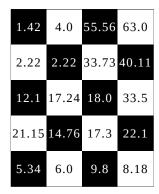
# Task 5: Checkerboard Sum

Given a 2D NumPy array, we want to compute the sum of all elements occurring in a checkerboard pattern of arbitrary size. The square in the first row and the first column is always black.

Write a function which takes as input a 2D NumPy array. The function should return the sum of all elements in the black squares of the checkerboard pattern.

Consider the 2D NumPy array below.

```
A = np.array([[ 1.42,   4.0,  55.56, 63.0],
              [ 2.22,   2.22, 33.73, 40.11],
              [12.1,   17.24, 18.0,  33.5],
              [21.15, 14.76, 17.3,  22.1],
              [ 5.34,   6.0,   9.8,   8.18]])
```

Arranged in a checkerboard pattern the array looks like this:



The sum of all elements occurring in a checkerboard pattern on the black squares is

$$1.42 + 55.56 + 2.22 + 40.11 + 12.1 + 18.0 + 14.76 + 22.1 + 5.34 + 9.8 = 181.41$$

and this is what your function should return, as seen below.

```
>>> checkerboard_sum(A)
np.float64(181.41)
```

The filename and requirements are in the box below:

checkerboard_sum.py

`checkerboard_sum(A)`

Return checkerboard sum.

*Parameters*:

- `A`  `numpy.ndarray`   A 2D NumPy array.

*Returns*:

- `float`   The sum of elements in checkerboard pattern.

# Task 6: Collatz Conjecture

The Collatz conjecture is an unsolved problems in mathematics. One step of the Collatz conjecture is defined as

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{cases}.$$

The conjecture states that for any positive integer $n$, the sequence $n, f(n), f(f(n)), f(f(f(n))), \ldots$ will reach the number 1, but whether the conjecture is true has not yet been proven or disproven.

Write a function that takes as input a positive integer. The function should return the number of steps required to reach the number 1.

Consider the number $3$. Since $3$ is an odd number, the next number in the sequence is $3 \cdot 3 + 1 = 10$. This is an even number, so the next number in the sequence is $10/2 = 5$. The full sequence is $3, 10, 5, 16, 8, 4, 2, 1$, and it took $7$ steps to reach the number $1$. The expected output is shown in the code cell below.

```
>>> collatz_conjecture(3)
7
```

The filename and requirements are in the box below:

collatz_conjecture.py

collatz_conjecture(n)

Return the number of steps to reach 1 in the Collatz conjecture.

*Parameters*:

- n   int   A positive integer, the starting number.

*Returns*:

- int   The number of steps.
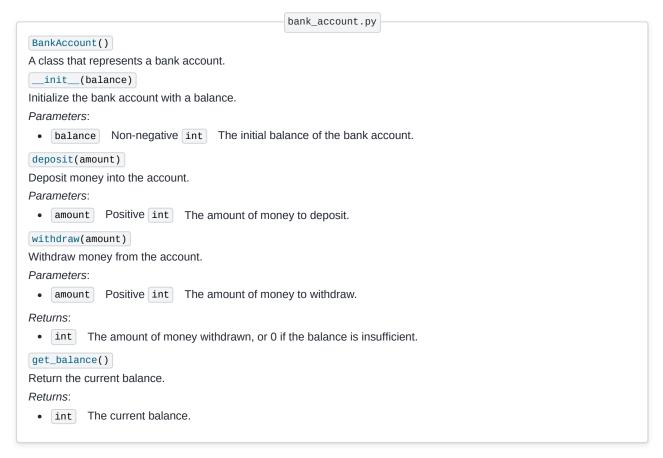
# Task 7: Bank Account

We want to create a class to represent a bank account, allowing for depositing and withdrawing money while ensuring the balance never goes negative.

Write the class definition for the class `BankAccount`. The balance must be stored in an attribute called `balance`. The `__init__` method should take the initial balance as input. The `deposit` method should take as input an amount to deposit and add it to the balance. The `withdraw` method should take as input an amount to withdraw, subtract it from the balance, and return the amount withdrawn. If the withdrawal would result in a negative balance, the method should leave the balance unchanged and return 0. The `get_balance` method should return the current account balance.

Consider the example below.

```
>>> my_account = BankAccount(1000)
>>> my_account.get_balance()
1000
>>> my_account.deposit(500)
>>> my_account.get_balance()
1500
>>> my_account.withdraw(200)
200
>>> my_account.get_balance()
1300
>>> my_account.withdraw(2000)
0
>>> my_account.get_balance()
1300
```

In this example, the balance is 1000 initially. Then, 500 is deposited. Next, 200 is withdrawn, which is allowed since the balance before withdrawing is 1500. Finally, an attempt to withdraw 2000 is made, but since the current balance is only 1300, the withdrawal is not possible. Therefore the method returns 0, and the balance remains unchanged.

The filename and requirements are in the box below:

---
bank_account.py
---

`BankAccount()`

A class that represents a bank account.

`__init__(balance)`

Initialize the bank account with a balance.

*Parameters*:

- `balance`   Non-negative `int`   The initial balance of the bank account.

`deposit(amount)`

Deposit money into the account.

*Parameters*:

- `amount`   Positive `int`   The amount of money to deposit.

`withdraw(amount)`

Withdraw money from the account.

*Parameters*:

- `amount`   Positive `int`   The amount of money to withdraw.

*Returns*:

- `int`   The amount of money withdrawn, or 0 if the balance is insufficient.

`get_balance()`

Return the current balance.

*Returns*:

- `int`   The current balance.

# Task 8: Phonebook Merge

A phonebook is represented as a dictionary where each key corresponds to a contact's name, and the corresponding value is a list of phone numbers associated with that contact. Both the names and phone numbers are strings. Given a second phonebook, we want to add its content to the first phonebook, but without creating duplicates.

Write a function that takes two dictionaries representing phonebooks as input. The function should not have a return statement, but it should *modify* the first phonebook by adding the content from the second phonebook. Specifically:

1. If a name from the second phonebook is not present in the first phonebook, it should be added to the first phonebook along with its associated phone numbers.

2. If a name from the second phonebook is already present in the first phonebook, then we look at the two lists of phone numbers for that name. Phone numbers that are only present in the second list should be appended to the first list in the order they occur in the second list.

Consider the example below.

```
phonebook = {'Liv': ['55511112', '18777890'] ,
             'Mads': ['27274445', '48533336'],
             'Steve': ['45455555', '25455525']}

second_phonebook = {'Anna': ['89577772'] ,
                    'Steve': ['25257755', '25455525'],
                    'Mads': ['48533336', '27274445']}
```

Consider now the elements of `second_phonebook`. The name **Anna** is not present in `phonebook`, so it should be added along with its associated phone numbers. The name **Steve** is already present in `phonebook`, and the phone numbers from `second_phonebook` include a new number **45455555**, which should be appended to the list of phone numbers for **Steve**. The name **Mads** is already present in `phonebook`, and `second_phonebook` does not provide any new phone numbers for **Mads**, so there is nothing to add.

The expected behavior is shown in the code cell below.

```
>>> phonebook_merge(phonebook, second_phonebook)
>>> for name in phonebook:
...     print(name, phonebook[name])
Liv ['55511112', '18777890']
Mads ['27274445', '48533336']
Steve ['45455555', '25455525', '25257755']
Anna ['89577772']
```

The filename and requirements are in the box below:

---
phonebook_merge.py
---

`phonebook_merge(phonebook, second_phonebook)`

Modify phonebook by adding new content from second_phonebook.

*Parameters*:

- `phonebook`          `dict`    Dictionary with names and list of phone numbers.
- `second_phonebook`   `dict`    Dictionary with names and list of phone numbers.

# Task 9: Nitrate Levels

Once a week, samples of drinking water are tested for nitrate. The test results are stored in a file where each line contains a floating-point number representing one nitrate level measurement. Nitrate levels are categorized as:

- **Very low**: Nitrate levels less than or equal to 4.0 mg/l.

- **Low**: Nitrate levels above 4.0 but less than or equal to 9.0 mg/l.

- **Normal**: Nitrate levels above 9.0 and below 40.0 mg/l.

- **High**: Nitrate levels greater than or equal to 40.0 but below 50.0 mg/l.

- **Very high**: Nitrate levels greater than or equal to 50.0 mg/l.

Note here that when the nitrate level falls on the border between two categories, it is included in the category further from normal. For example, a nitrate level of 4.0 mg/l is **very low**, and a nitrate level of 40.0 mg/l is **high**.

Write a function that takes a string containing the file name with the nitrate levels as input. The function should return the number of weeks where the nitrate levels were very low, low, normal, high, and very high, respectively, as shown in the example below.

Consider the file `files/nitrate_data_A.txt` with the content below.

```
34.5
34.9
36.7
29.9
34.5
44.5
34.5
46.5
29.9
34.5
```

None of the values are below 9.0, so none belong to the lower two categories. Eight values are in the range from 9.0 to 40.0, classifying them as normal. Two values are between 40.0 and 50.0, placing them in the high category. There are no values that are classified as very high. The function therefore returns 0, 0, 8, 2, 0.

The expected output may be seen in the example.

```
>>> nitrate_levels('files/nitrate_data_A.txt')
(0, 0, 8, 2, 0)
```

The filename and requirements are in the box below:

---
nitrate_levels.py

`nitrate_levels(filename)`

Return the number of weekly measurements in each category.

*Parameters*:

- `filename` `str` Filename of the data file.

*Returns*:

- `tuple` Number of measurements in each of five categories for nitrate levels.
---

# Task 10: Overdraft Account

We want to create a subclass of the `BankAccount` class from Task 7. This subclass should allow the user to have a negative balance, as long as the sum of the balance and the overdraft limit is not negative.

Write the class definition for the subclass `OverdraftAccount`, which inherits from `BankAccount`. Each instance of the subclass should store the overdraft limit. The constructor of the new class should take as input the initial balance and the overdraft limit (a non-negative integer). The `withdraw` method should ensure that the overdraft limit is not exceeded. If the withdrawal is not possible, the balance should remain unchanged. As before, the `withdraw` method should return the amount withdrawn. You should modify the necessary methods of the class to achieve this behavior, and inherit the rest of the methods from the parent class.

You should write the class definition for `OverdraftAccount` in the same file as the class definition for `BankAccount`.

Refer to the example below for expected behavior.

```
>>> my_account = OverdraftAccount(0, 500)
>>> my_account.get_balance()
0
>>> my_account.deposit(1000)
>>> my_account.get_balance()
1000
>>> my_account.withdraw(1300)
1300
>>> my_account.get_balance()
-300
>>> my_account.withdraw(500)
0
>>> my_account.get_balance()
-300
```

In this example, the initial balance is 0, and the overdraft limit is 500. First, 1000 is deposited. Then, 1300 is withdrawn. This is allowed since it brings the balance to -300 which is above -500. Finally, 500 is attempted to be withdrawn, but that would bring the balance below -500. So, this withdrawal is not possible, and the balance remains unchanged.

The filename and requirements are in the box below:

bank_account.py

`OverdraftAccount()`

A class that represents a bank account allowing overdraft.

`__init__(balance, overdraft_limit)`

Initialize the overdraft account with a balance and an overdraft limit.

*Parameters*:

- `balance`          Non-negative `int`   The initial balance of the bank account.
- `overdraft_limit`  Non-negative `int`   The overdraft limit of the bank account.

`withdraw(amount)`

Withdraw money from the bank account.

*Parameters*:

- `amount`   Positive `int`   The amount of money to withdraw.

*Returns*:

- `int`   The amount of money withdrawn, or 0 if withdrawal fails.