Exam Fall 2024

Course name:	Computer programming
Course number:	02002 and 02003
Exam date:	20th of December 2024
Aids allowed:	All aids, no internet
Exam duration:	4 hours
Weighting:	All tasks have equal weight
Number of tasks:	10
Number of pages:	12

Contents

Exam Instructions

Exam Material

The exam material is a single zip file, which you should unzip to a folder on your computer. The zip file contains the exam text as a pdf document in English `2024_12_exam_English.pdf` (this document) and the same document in Danish `2024_12_exam_Danish.pdf`. Each pdf contains everything needed to solve the exam.

As additional help, the zip file contains a folder `2024_12_exam` with the following content:

- Empty Python files `<task_name>.py`, where `<task_name>` is the name of the task. There may be fewer files than tasks.
- A Python test file for each task, `test_task_<n>_<task_name>.py`, where `<n>` is the task number, and `<task_name>` is the name of the task.
- A Python file `test_tasks_all.py`.
- A folder `files` containing data files, if there are any.

Solving Exam Tasks

To be able to solve the exam tasks, you need to have a computer with Python installed. All tasks can be solved using any editor of your choice, such as IDLE or VS Code.

We recommend that you use the provided files: Write your solutions in the empty files and test them by running the provided test scripts. These scripts check whether your solution behaves correctly for the input in the task description. Use `test_tasks_all.py` to run all tests at once. The provided test scripts assume that the *current working directory* is `2024_12_exam`. Therefore, if you are using VS Code, you should click `File` → `Open Folder...` and select the `2024_12_exam` folder (which is *inside* the folder you unzipped).

A solution that fails the provided test is incorrect. If your solution passes the provided test, it is not a guarantee that your solution is correct.

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

If you believe there is a mistake or ambiguity in the text, use the most reasonable interpretation of the text and 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 during the assessment.

Evaluation of the Exam

We will run additional tests for each task to check whether your solutions behave as specified in the task. The fraction of passed tests is the score for each task. The overall score is the average of the task scores.

Handing in

To hand in, upload your Python files with solutions to the Digital Exam system. In the Digital Exam system, files can be submitted either as the *main document* or as *attachments*. You may upload *any* one of your solution files as the main document, and the remaining files as attachments.

Please submit only the following files with these exact names. All other files will be ignored.

- `chamber_pressure.py`
- `fruit_weights.py`
- `new_price.py`
- `orbital_period.py`
- `pattern_count.py`
- `reservoir_levels.py`
- `simple_tracker.py`
- `traffic_operation.py`
- `weighted_average.py`

Task 1: Traffic Operation

Traffic lights operate in one of the following three modes depending on the time of day:

- *rush hour* from 7:00-8:59 and 15:00-16:59,
- *night* from 22:00 until 5:59 the following morning,
- *normal* all other times.

Since operation mode always changes at the beginning of an hour, it is solely a function of the current hour (a number from 0 to 23).

Write a function that takes the current hour as input. The function should return the operation mode as a string, either `'normal'`, `'rush hour'`, or `'night'`.

For example, if the input hour is 8, it falls within the rush hour period between 7:00 and 8:59. Therefore, the function should return `'rush hour'`, as show below.

```
>>> traffic_operation(8)
'rush hour'
```

The filename and requirements are:

`traffic_operation.py`

`traffic_operation(hour)`

Determines the traffic light operation mode depending on the hour of the day.

Parameters:

- `hour` `int` The hour of the day.

Returns:

- `str` The traffic operation mode for the hour of the day.

Task 2: Orbital Period

Kepler's third law of planetary motion says

$$T^2 = \frac{4\pi^2}{GM}a^3$$

where T is the orbital period of the planet (in seconds), a is the length of the semi-major axis of the orbit (in meters), M is the total mass of the system (in kilograms), and $G = 6.6743 \cdot 10^{-11}\,\mathrm{m}^3\,\mathrm{kg}^{-1}\,\mathrm{s}^{-2}$ is the gravitational constant.

Write a function that takes as input two numbers: the length of the semi-major axis (in meters) and the total mass of the system (in kilograms). The function should return the orbital period (in seconds).

As an example, consider the system consisting of the Sun and the Earth. The semi-major axis of Earth's orbit is $1.5 \cdot 10^{11}$ meters, and the total mass of the system is $2 \cdot 10^{30}$ kilograms. We have (not writing units, and displaying only a few decimals)

$$T^2 = \frac{4\pi^2}{6.6743 \cdot 10^{-11} \cdot 2 \cdot 10^{30}}\left(1.5 \cdot 10^{11}\right)^3 = 9.981 \cdot 10^{14}$$

so $T = \sqrt{9.981 \cdot 10^{14}} = 3.159 \cdot 10^7$ seconds (which is about 365.7 days). This example is shown below.

```
>>> orbital_period(1.5 * 10**11, 2.0 * 10**30)
31593584.1373
```

The filename and requirements are:

`orbital_period.py`

`orbital_period(a, M)`

Calculate the orbital period of a planet using Kepler's third law.

Parameters:

- `a` `float` The length of the semi-major axis of the orbit, in meters.
- `M` `float` The total mass of the system, in kilograms.

Returns:

- `float` The orbital period of the planet, in seconds.

Task 3: Chamber Pressure

The pressure in a chamber increases every hour by $k(P_{\max} - P)$, where P is the current pressure, $P_{\max}$ is the maximum pressure, and k is a constant. We are interested in the time it takes for the pressure to reach or exceed a critical pressure P_{crit}.

Write a function that takes as input the initial pressure, the maximum pressure, the critical pressure, and the constant k. The function should return the number of hours it takes for the pressure to reach or exceed the critical pressure.

For instance, suppose the initial pressure is $P = 20$, the maximum pressure is $P_{\max} = 120$, the critical pressure is $P_{\text{crit}} = 105$, and the constant is $k = 0.1$. After one hour, the pressure increases by $0.1(120 - 20) = 10$, bringing it to 30. In the second hour, it increases by $0.1(120 - 30) = 9$, resulting in a pressure of 39. Repeating this process, the pressure reaches 104.99 after 18 hours, still below the critical value. After 19 hours, the pressure is 106.49, exceeding the critical value. This is shown in the code below.

```
>>> chamber_pressure(20.0, 120.0, 105.0, 0.1)
19
```

The filename and requirements are:

`chamber_pressure.py`

`chamber_pressure(P0, Pmax, Pcrit, k)`

Computes the number of hours it takes for the pressure to reach or exceed the critical value.

Parameters:

- `P0` `float` The initial pressure.
- `Pmax` `float` The maximum pressure.
- `Pcrit` `float` The critical pressure.
- `k` `float` The constant.

Returns:

- `int` The number of hours it takes for the pressure to reach or exceed the critical value.

Task 4: Reservoir Levels

Water levels in a reservoir are recorded every hour. If the water level drops by more than 150 cm in an hour, it must be reported.

Write a function that takes a list of water levels as input and returns a list of indices where the water level drops strictly more than 150 cm compared to the previous level.

For example, consider the water levels `[1320, 1307, 1295, 1102, 1360, 1395, 1101, 1208]`. The level at index 0 has no previous level for comparison. The next two levels have drops of 13 and 12 cm. The level at index 3 is $1295 - 1102 = 193$ cm lower than the previous level, so this is the first index to be reported. The next two levels are higher that the previous levels. The level with the index 6 drops by 294 cm compared to the previous level, so it is reported. The last level is higher than the previous one. The function should return `[3, 6]`, as shown below.

```
>>> reservoir_levels([1320, 1307, 1295, 1102, 1360, 1395, 1101, 1208])
[3, 6]
```

The filename and requirements are:

`reservoir_levels.py`

`reservoir_levels(levels)`

Detects indices for water levels that are more than 150 lower than the previous measurement.

Parameters:

- `levels` `list` The list of water level measurements.

Returns:

- `list` The indices for water levels that are more than 150 lower than the previous measurement.

Task 5: New Price

When a product goes on sale, its price is reduced by a specified percentage. The discounted price must be displayed in a standardized format.

For example, if the original price is 143.50 DKK and the discount is 40%, the new price is calculated as

$$143.5 - 143.5\frac{40}{100} = 86.1\,.$$

To standardize the display, we round the new price to exactly two decimal places, add a single space, and add the currency code DKK. The price string becomes `'86.10 DKK'`.

Write a function that takes as input the original price and the discount percentage. The function should return a tuple containing the new price and the formatted price string, as show in the code below.

```
>>> new_price(143.50, 40)
(86.1, '86.10 DKK')
```

The filename and requirements are:

`new_price.py`

`new_price(price, discount)`

Computes the new price and formats the price string.

Parameters:

- `price` `float` The original price.
- `discount` `int` The discount percentage.

Returns:

- `tuple` The new price and the price string.

Task 6: Fruit Weights

You have a table of fruit weights (in grams) for various fruits, but the fruit names are inconsistently formatted, appearing in lowercase (*apple*), uppercase (*APPLE*), or capitalized (*Apple*). You want to standardize the table to only contain lowercase names. If multiple records exist for the same fruit, the weights should be averaged and rounded down.

Write a function that takes a table as input and returns a standardized table. Both tables are *dictionaries*. For example, consider the table below.

```
>>> table = {'apple': 182,
...     'banana': 110,
...     'Orange': 160,
...     'Banana': 115,
...     'APPLE': 185,
...     'Apple': 175,
...     'lime': 67}
```

In this table, *apple* appears in three forms with the weights 182, 185, and 175 grams. In the standardized table, the name should be `'apple'` with the weight being the average (182+185+175)/3 rounded down to 180. *Banana* appears in two forms with the weights 110 and 115 grams. So, the standardized table should have `'banana'` with weight (110+115)/2 rounded down to 112. *Orange* and *lime* each appear once, so the weights remain unchanged, but the names should be converted to lowercase.

You can see the expected output in the example.

```
>>> fruit_weights(table)
{'apple': 180, 'banana': 112, 'orange': 160, 'lime': 67}
```

The filename and requirements are:

`fruit_weights.py`

`fruit_weights(table)`

Standardizes table for fruit weights.

Parameters:

- `table` `dict` A dictionary of fruit names and weights.

Returns:

- `dict` A dictionary of fruit names and standardized weights.

Task 7: Weighted Average

Given N values $x_0, x_1, \ldots, x_{N-1}$, you want to calculate their weighted average. The weights are determined from the values as

$$w_i = e^{\frac{-(x_i-\mu)^2}{2\sigma^2}}$$

where μ is the mean of the values and σ is the standard deviation. The mean and standard deviation can be calculated as

$$\mu = \frac{1}{N}\sum_{i=0}^{N-1} x_i \qquad \text{and} \qquad \sigma = \sqrt{\frac{1}{N}\sum_{i=0}^{N-1}(x_i-\mu)^2}.$$

The weighted average is then computed as

$$a = \frac{\sum_{i=0}^{N-1} w_i x_i}{\sum_{i=0}^{N-1} w_i}\,.$$

For example, consider the values $[4.8\,,\ 6.6\,,\ 12.2\,,\ 7.3\,,\ 6.5]$. The mean and standard deviation are $\mu = 7.48$ and $\sigma = 2.499$, respectively. The weight for the first element is $e^{\frac{-(4.8-7.48)^2}{2\cdot 2.499^2}} = 0.563$, and all weights are $[0.563\,,\ 0.94\,,\ 0.168\,,\ 0.997\,,\ 0.926]$. Finally, the weighted average is

$$a = \frac{0.563\cdot 4.8+\ldots}{0.563+\ldots} = \frac{24.254}{3.594} = 6.749\,.$$

In a special case where $\sigma = 0$, for example if all values x_i are the same, the weighted average is equal to the mean of the values.

Write a function that takes an NumPy array of values as input and returns the weighted average. You can see the expected output in the example.

```
>>> import numpy as np
>>> weighted_average(np.array([4.8, 6.6, 12.2, 7.3, 6.5]))
np.float64(6.748513328262833)
```

The filename and requirements are:

`weighted_average.py`

`weighted_average(x)`

Computes the weighted average of the array.

Parameters:

- `x` `numpy.ndarray` An array of numbers.

Returns:

- `float` The weighted average of the array.

Task 8: Pattern Count

A strand of DNA is represented as a text containing the letters `A`, `C`, `G`, and `T`, and is stored in a file. The goal is to count how many times a given pattern appears in the DNA strand. There might be line breaks in the file, but they carry no meaning and should be ignored. All occurrences of the pattern should be counted, even if they overlap. For example, the pattern `ACA` appears twice in the DNA strand `ACACA`.

Write a function that takes a file name and a pattern as input and returns the number of times the pattern occurs in the DNA strand.

For example, consider a file `files/dna_data_1.txt` with the content:

```
CGGTGCGGGCCTCTTCGCTATTACGCCAGCTGGCG
AAAGGGGGATGTGCTGCAAGGCGATTAAGTTGGGTAACGCCAGGG
TTTTCCCAGTCACGACGTTGTAAAACGACGGCCAGT
GAGCGCGCGTAATACGACTCACTATAGGGCGAATTGGAGCTCCA
CCGCGGTGGCGGCCGCTCTAGAACTAGTGGATCCCCCGG
GCTGCAGGAATTCGATATCAAGCTTATCGATACCGTCGACC
TCGAGGGGGGGCCCGGTACCCAGCTTTTGTTCCCTTTAGTGAG
GGTTAATTGCGCGCTTG
```

and a pattern `'ACCG'`. This pattern appears twice in the DNA strand: once with the `A` at the end of the fourth line which continues onto the fifth line, and a second time with the `A` located 10 characters before the end of the sixth line.

The expected output may be seen in the example.

```
>>> filename = 'files/dna_data_1.txt'
>>> pattern_count(filename, 'ACCG')
2
```

The filename and requirements are:

`pattern_count.py`

`pattern_count(filename, pattern)`

Finds the number of occurrences of the pattern in the DNA strand.

Parameters:

- `filename` `str` The name of the file with the DNA strand.
- `pattern` `str` The pattern to search for.

Returns:

- `int` The number of occurrences of the pattern in the DNA strand.

Task 9: Simple Tracker

We want to create a tracker that can keep track of multiple numbers, but only those that are positive and that do not exceed a specified *limit*. The tracker should start empty and numbers are added to the tracker one-by-one if they fulfil the requirements. The tracker should be able to compute simple statistics of the added numbers. It should be possible to reset the tracker.

Write a class definition for the class `SimpleTracker`. The constructor should take a positive integer, the *limit*. The tracker should be empty initially. The `add` method should take a number as input, and add it to the added numbers if it is positive and not larger than the *limit*. The `add` method should return `True` if the number was added, and `False` if it was not. The `reset` method should clear the added numbers. The `stats` method should return a tuple with two elements: the *total* and the *span*. Here, the *total* is the sum of all added numbers, and the *span* is the difference between the largest and the smallest added number. If the tracker is empty, both the *total* and the *span* should be 0.

Consider the example below.

```
>>> tracker = SimpleTracker(500)
>>> tracker.add(510)
False
>>> tracker.add(200)
True
>>> tracker.add(352)
True
>>> tracker.stats()
(552, 152)
>>> tracker.reset()
>>> tracker.stats()
(0, 0)
```

In this example, a tracker is created with a *limit* of 500. When attempting to add the number 510, `False` is returned because the number exceeds the *limit* and cannot be added to the tracker. Next, the numbers 200 and 352 are successfully added. At this point the *total* and the *span* are 200 + 352 = 552 and 352-200 = 152. Then, the tracker is reset and both the *total* and the *span* are 0.

The filename and requirements are:

`simple_tracker.py`

`SimpleTracker()`
A class that tracks valid numbers.

`__init__(limit)`
Initialize a tracker with a limit.

Parameters:

- `limit` `int` The limit that valid numbers cannot exceed.

`add(number)`
Add a valid number to the tracked numbers.

Parameters:

- `number` `int` The number to be added.

Returns:

- `bool` `True` if the number is added, `False` otherwise.

`reset()`
Reset tracker.

`stats()`
Return the total and the span.

Returns:

- `tuple` The total and the span of the tracked numbers.

Task 10: Advanced Tracker

We want to create a subclass of the `SimpleTracker` class from Task 9 that performs an additional check before adding a number to tracker.

When attempting to add a new number, if the tracker is not empty, the new number should be added only if it differs from the previous by no more than a specific value, *max delta*. If the absolute difference exceeds *max delta*, the number should not be added. In addition, the number should still be checked for validity as in the parent class, and the `add` method should return `True` if the number was added, and `False` if it was not.

Write the class definition for the subclass `AdvancedTracker`, which inherits from `SimpleTracker`. The constructor should take both *limit* and *max delta* as input parameters. Modify the necessary methods to incorporate this additional behavior, and inherit the unchanged methods from the parent class.

You should write the class definition for `AdvancedTracker` in the same file as the class definition for `SimpleTracker`.

Refer to the example below for expected behavior.

```
>>> tracker = AdvancedTracker(500, 100)
>>> tracker.add(200)
True
>>> tracker.add(352)
False
>>> tracker.add(252)
True
>>> tracker.stats()
(452, 52)
>>> tracker.add(510)
False
```

In this example, the tracker is initialized with a *limit* of 500 and a *max delta* of 100. The number 200 is added successfully. The number 325 is not added because the absolute difference between its value and the previous 352-200=152 is larger than the *max delta*, which is 100. The number 252 is added successfully because the absolute difference between its value and the previous is 52. At this point, the *total* is 252+200=452 and the *span* is 252-200=52. When attempting to add 510, it is not added because it exceeds the *limit* of 500.

The filename and requirements are:

`simple_tracker.py`

`AdvancedTracker()`

A class that tracks of non-extreme valid numbers.

`__init__(limit, max_delta)`

Initialize a record with a limit and a maximum delta.

Parameters:

- `limit` `int` The limit that valid numbers cannot exceed.
- `max_delta` `int` The maximum delta value for determining non-extreme numbers.

`add(number)`

Add a valid and non-extreme number to the tracked numbers.

Parameters:

- `number` `int` The number to be added.

Returns:

- `bool` `True` if the number is added, `False` otherwise.