Exam Winter 2025

Course name:	Computer programming
Course number:	02002 and 02003
Exam date:	21st of December 2025
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
- Task 1: Gauge Diameter
- Task 2: Gate Distance
- Task 3: Parallel Resistors
- Task 4: Graph Transpose
- Task 5: Odd Means
- Task 6: Navigation Tracking
- Task 7: Sentence Complexities
- Task 8: Bike Light
- Task 9: Suitcase
- Task 10: Shop Status

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 `2025_12_exam_English.pdf` (this document) and the same document in Danish `2025_12_exam_Danish.pdf`. Each pdf contains everything needed to solve the exam.

As additional help, the zip file contains a folder `2025_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 `2025_12_exam`. Therefore, if you are using VS Code, you should click `File` → `Open Folder...` and select the `2025_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.

- `bike_light.py`
- `gate_distance.py`
- `gauge_diameter.py`
- `graph_transpose.py`
- `navigation_tracking.py`
- `odd_means.py`
- `parallel_resistors.py`
- `sentence_complexities.py`
- `shop_status.py`
- `suitcase.py`

Task 1: Gauge Diameter

A 12-gauge ball is produced by taking one pound of lead, dividing it into twelve equal parts, and rolling each part into a ball. For another gauge, say 8-gauge, the same procedure is used, but with eight equal parts.

Therefore, the diameter d of an n-gauge ball satisfies

$$\frac{\pi}{6}d^3 = \frac{V}{n},$$

where V is the volume of one pound of lead, which is $40\,\mathrm{cm}^3$.

Write a function that, given n, returns the diameter of the ball in centimeters.

For example, for the 12-gauge ball:

$$d^3 = \frac{6V}{n\pi} = \frac{6 \cdot 40\ \mathrm{cm}^3}{12\pi} = 6.3662\ \mathrm{cm}^3,$$

$$d = \sqrt[3]{6.3662\ \mathrm{cm}^3} = (6.3662\ \mathrm{cm}^3)^{\frac{1}{3}} = 1.8534\ \mathrm{cm}.$$

So, the diameter of the 12-gauge ball is 1.8534 cm, and the function should behave as shown below.

```
>>> gauge_diameter(12)
1.8533610896304256
```

The filename and requirements are:

`gauge_diameter.py`

`gauge_diameter(n)`

Computes the diameter of an n-gauge ball.

Parameters:

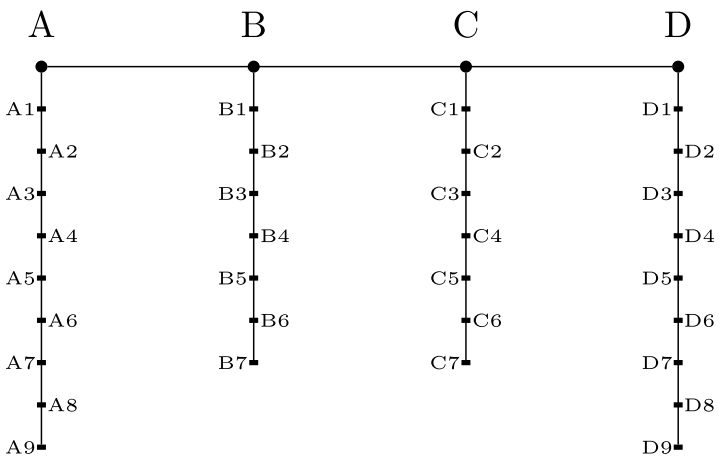
- `n` `int` The gauge.

Returns:

- `float` The diameter in cm.

Task 2: Gate Distance

An airport has four terminals. Each terminal has several gates, as shown below.



In the illustration, the start of each terminal is marked with a circle. It takes 4 minutes to walk from the start of one terminal to the start of the next terminal. Along a terminal, it takes 0.8 minutes to walk from the start to the first gate, and 0.8 minutes between each pair of neighboring gates. We need to calculate the fastest walking time between two gates, and create a message where the fastest walking time is rounded up to the nearest whole minute.

For example, consider the gates A8 and B5. Walking from A8 to the start of Terminal A takes 8 times 0.8 minutes. Walking from the start of Terminal A to the start of Terminal B takes 4 minutes. Finally, walking from the start of Terminal B to B5 takes 5 times 0.8 minutes. The fastest walking time is therefore $4 + (8 + 5)\,0.8 = 14.4$ minutes, which rounds up to 15 minutes.

Write a function that, takes two gates as input. Gates are given as strings containing a letter and an integer. The function should return the string `'Walking time <n> min'` where `<n>` is the fastest walking time between the two gates, rounded up to the nearest whole minute. The function should also work correctly if the airport is expanded with more gates. However, you may assume that there will only be the four terminals A, B, C, and D.

The behavior of the function is shown below.

```
>>> gate_distance('A8', 'B5')
'Walking time 15 min'
```

The filename and requirements are:

`gate_distance.py`

`gate_distance(gate1, gate2)`

Compute gate distance for two airport gates.

Parameters:

- `gate1` `str` Gate identifier containing a letter (terminal) and gate number.
- `gate2` `str` Gate identifier containing a letter (terminal) and gate number.

Returns:

- `str` Information about the walking distance between the gates.

Task 3: Parallel Resistors

When resistors are connected in parallel, the total resistance R can be calculated using the formula:

$$\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3} + \ldots + \frac{1}{R_n}.$$

If three resistors with resistances $10.5\ \Omega$, $12.5\ \Omega$, and $50\ \Omega$ are connected in parallel, the total resistance is

$$\frac{1}{R} = \frac{1}{10.5\ \Omega} + \frac{1}{12.5\ \Omega} + \frac{1}{50\ \Omega} = 0.1952\ \Omega^{-1}$$

$$R = \frac{1}{0.1952\ \Omega^{-1}} = 5.1219\ \Omega.$$

Write a function that, given a list of resistances, returns the total resistance. You can assume that there is at least one resistor and that all resistances are positive.

The behavior of the function is shown below.

```
>>> parallel_resistors([10.5, 12.5, 50.0])
5.121951219512195
```

The filename and requirements are:

`parallel_resistors.py`

`parallel_resistors(resistances)`

Parallel resistors.

Parameters:

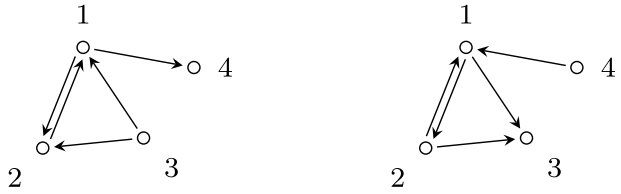
- `resistances` `list[float]` A non-empty list of positive resistances.

Returns:

- `float` Resistance.

Task 4: Graph Transpose

A graph has nodes, each identified with an integer. Nodes are connected by edges going from one node to another. The transpose of a graph is made by reversing the direction of all edges. For example, consider the graph on the left. Its transpose is shown on the right.



We represent the graph using a *dictionary*. Each *key* is a node with outgoing edges. For each *key* the associated *value* is a list of nodes that the edges go to. The nodes in the lists are in ascending order. If a node has no outgoing edges, it does not appear as a *key*. For example, the graph on the left is represented as

```
graph = {
    2: [1],
    1: [2, 4],
    3: [1, 2],
}
```

Write a function that, given a graph, returns its transpose, in the representation described above. Note that you cannot assume that the nodes are numbered consecutively — a graph may have nodes like 10, 20, 60 with gaps in the numbering.

The behavior of the function is shown below.

```
>>> graph_transpose(graph)
{1: [2, 3], 2: [1, 3], 4: [1]}
```

The filename and requirements are:

`graph_transpose.py`

`graph_transpose(graph)`

Compute the transpose of a graph.

Parameters:

- `graph` `dict` A directed graph.

Returns:

- `dict` The transposed graph.

Task 5: Odd Means

Given a 2D array of numbers, we want to compute the mean of each odd-numbered column. Odd-numbered columns are the first, third, fifth, and so on. Consider the array below.

$$\begin{bmatrix} 5.5 & 2.2 & 3.3 & 2.2 & 3.3 \\ 5.8 & 7 & 8 & 2.2 & 3.3 \\ 10 & 0.9 & 10 & 1.1 & 1.2 \end{bmatrix}$$

The array has five columns, and the odd-numbered columns are the first, third, and fifth column. The mean of the first column is $(5.5 + 5.8 + 10)/3 = 7.1$. The mean of the third column is $(3.3 + 8 + 10)/3 = 7.1$. The mean of the fifth column is $(3.3 + 3.3 + 1.2)/3 = 2.6$. So, the odd means are $\begin{bmatrix} 7.1 & 7.1 & 2.6 \end{bmatrix}$.

Write a function that, given a 2D NumPy array, returns a NumPy array containing the means of the odd-numbered columns.

The behavior of the function is shown below.

```
>>> import numpy as np
>>> M = np.array([[5.5, 2.2, 3.3, 2.2, 3.3],
...               [5.8, 7, 8, 2.2, 3.3],
...               [10, 0.9, 10, 1.1, 1.2]])
>>> odd_means(M)
array([7.1, 7.1, 2.6])
```

The filename and requirements are:

`odd_means.py`

`odd_means(M)`

Compute the means of odd columns.

Parameters:

- `M` `numpy.ndarray` 2D array.

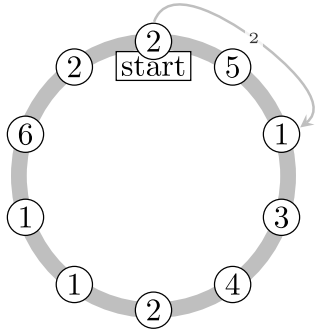
Returns:

- `numpy.ndarray` The means of odd columns.

Task 6: Navigation Tracking

Some positions for navigation are arranged in a circle, with one position designated as the starting position. You move forward the number of spaces indicated on your current position in the clockwise direction. It is not allowed to revisit a position, so you stop before that happens. You want to know the values of visited positions, in the order of they were visited.

For example, consider the illustration below.



The starting position says you should move two steps forward. After that, you land on the position with number 1, as indicated by the arrow. After moving one position forward, you land on 3. Moving three steps forward, you land on 1. Moving one forward, you land again on 1, and moving one more forward, you land on 6. After moving six positions forward, you land on 4. From there, moving four steps, you would land on 6, but this position had already been visited, and the position with 4 was the last position visited. So, the positions in the order they were visited are: 2, 1, 3, 1, 1, 6, 4.

Write a function that, as input, takes a list of integers representing the positions, where the first element is the start position. Increasing the index corresponds to moving clockwise in the circle. The function should return a list of the visited positions in the order they were visited. You can assume that there are at least two positions, and that all positions contain positive integers.

The behavior of the function for the example above is shown below.

```
>>> navigation_tracking([2, 5, 1, 3, 4, 2, 1, 1, 6, 2])
[2, 1, 3, 1, 1, 6, 4]
```

The filename and requirements are:

`navigation_tracking.py`

`navigation_tracking(instructions)`

Track navigation until repetition.

Parameters:

- `instructions` `list[int]` List of positive integers representing positions.

Returns:

- `list` The positions visited in the order they were visited.

Task 7: Sentence Complexities

You are given a text file with sentences on separate lines. Each sentence consists of words made of letters from the English alphabet separated by single spaces, and punctuation (`.,!?`) directly following some words. We define the complexity of a sentence as:

$$m = (\text{number of words}) + (\text{number of letters in the longest word})$$

and we want to compute the complexities for all sentences in the file.

For example, consider the following file.

`files/sentences_1.txt`

```
The cat sleeps.
This sentence is quite long and descriptive.
Hello world!
Programming is fun, sometimes challenging.
```

The first sentence has three words, and the longest word has six letters (*sleeps*), so its complexity is $3 + 6 = 9$. The second sentence has seven words, the longest word has eleven letters (*descriptive*), so its complexity is $7 + 11 = 18$. The third sentence has two words, each with five letters, so its complexity is $2 + 5 = 7$. The fourth sentence has complexity $5 + 11 = 16$. The complexities for all sentences in the file are therefore 9, 18, 7, and 16.

Write a function that takes a filename as input and returns a list of the complexity of each sentence in the file. If there are empty lines in the file, they should be ignored. An empty file should result in an empty list.

The function should behave as shown below.

```
>>> sentence_complexities('files/sentences_1.txt')
[9, 18, 7, 16]
```

The filename and requirements are:

`sentence_complexities.py`

`sentence_complexities(filename)`

Complexity values for all sentences in a file.

Parameters:

- `filename` `str` Path to a file with one sentence per line.

Returns:

- `list` Complexity values for all sentences in the file.

Task 8: Bike Light

A bike light has an *off* mode and three on-modes: *weak*, *strong*, and *flash*. It is controlled by a single button, which reacts differently to long and short presses.

The light starts in the *off* mode. A long press turns the light on to the *weak* mode. When the light is on, a short press cycles through the on-modes in the order: *weak*, *strong*, and *flash*. To turn the light off, from any on-mode, a long press is used. Turning the light on again starts in *weak* mode, regardless of the mode the light was in when it was turned off. When the light is off, a short press does not change its state.

Write a class `BikeLight` that represents a bike light. The constructor should take no arguments and initialize the light in the off mode. The class should have a method `long_press` that turns the light on or off. The method should return the mode as a string. The class should also have a method `short_press` that cycles through the modes when the light is on. The method should return the mode as a string.

The behavior of the class is shown below.

```
>>> light = BikeLight()
>>> light.long_press()
'weak'
>>> light.short_press()
'strong'
>>> light.short_press()
'flash'
>>> light.long_press()
'off'
>>> light.long_press()
'weak'
```

The filename and requirements are:

`bike_light.py`

`BikeLight()`
A bike light with multiple modes.

`__init__()`
Initialize the bike light in off mode.

`long_press()`
Turn the light on or off.

Returns:

- `str` The new mode.

`short_press()`
Cycle through modes when the light is on.

Returns:

- `str` The new mode.

Task 9: Suitcase

Write a class `Suitcase` representing a suitcase, where the constructor takes a weight limit as an argument. The class should have a method, `pack_item` that takes as argument an item and its weight. It packs the item into the suitcase if the combined weight of the items already in the suitcase and the new weight does not exceed the limit. The `currently_packed` method returns a list of items in the suitcase, in the order they were packed. The `__add__` method combines two `Suitcase`s into a new `Suitcase` with a weight limit equal to the sum of the two, packed with all items from the first suitcase followed by the items from the second suitcase, each in their original packing order. Consider the example below.

```
>>> carry_on = Suitcase(7.0)
>>> check_in = Suitcase(23.0)
>>> carry_on.pack_item('Laptop', 1.5)
True
>>> check_in.pack_item('Clothes', 5.0)
True
>>> check_in.pack_item('E-Bike', 20.0)
False
>>> all_luggage = carry_on + check_in
>>> all_luggage.currently_packed()
['Laptop', 'Clothes']
```

Two suitcases are created, with a weight limit of 7 kg for the *carry-on* and 23 kg for the *check-in* suitcase. A *laptop* weighing 1.5 kg is added to the *carry-on* suitcase, and 5 kg of *clothes* are added to the *check-in* suitcase. Adding a 20 kg *E-bike* to the *check-in* suitcase fails because $5 + 20 = 25$ is greater than 23. Finally, the two suitcases are added, and the contents of the new combined suitcase are listed.

The filename and requirements are:

`suitcase.py`

`Suitcase()`
A suitcase that can hold items up to a weight limit.

`__init__(weight_limit)`
Initialize the suitcase with a maximum weight.

Parameters:

- `weight_limit` `float` The weight limit of the suitcase.

`pack_item(item, weight)`
Pack an item if the total weight of items in the suitcase after packing it does not exceed the weight limit.

Parameters:

- `item` `str` The name of the item.
- `weight` `float` The weight of the item.

Returns:

- `bool` `True` if the item was packed, `False` otherwise.

`__add__(other)`
Combine two suitcases into a new one.

Parameters:

- `other` `Suitcase` Another suitcase.

Returns:

- `Suitcase` A new suitcase with a combined limit and items.

`currently_packed()`
Return the items currently packed in the suitcase.

Returns:

- `list` List of items.

Task 10: Shop Status

A shop is open Monday to Friday from 09:00 to 17:00. On Saturdays and Sundays, the shop is closed. We would like to know whether the shop is open, and when it will open or close next time.

Write a function that takes two arguments: a day of the week and an hour (an integer from 0 to 23). The function should return whether the shop is open at that time, the day when its status will change next time and the hour when that change will occur. On working days, if the hour is exactly 9, the shop is already open; if it is exactly 17, the shop is already closed.

You can use the following line of code in your solution:

```
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

Consider the example below.

```
>>> shop_status('Monday', 9)
(True, 'Monday', 17)
```

Here, we want to know the shop status on Monday at 9 o'clock. The shop is open at this time. The status will change on Monday at 17 o'clock, when the shop closes.

The filename and requirements are:

`shop_status.py`

`shop_status(day, hour)`

Determine if the shop is open or closed.

Parameters:

- `day` `str` The day of the week.
- `hour` `int` The hour of the day (0-23).

Returns:

- `tuple[bool, str, int]` A tuple containing:
  - Whether the shop is currently open.
  - The day of the week when the status will change next.
  - The hour when the status will change next.