

Exam Summer 2025

Course name:	Computer programming
Course number:	02002 and 02003
Exam date:	28th of May 2025
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
- Task 1: Polygon Area
- Task 2: Modified Blackjack
- Task 3: Running Speed
- Task 4: Photo Details
- Task 5: Experiment Numbers
- Task 6: Bouncing Ball
- Task 7: Number Statistics
- Task 8: Two Means
- Task 9: Bingo Card
- Task 10: Informative Bingo Card

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 `2025_05_exam_English.pdf` (this document) and the same document in Danish `2025_05_exam_Danish.pdf`. The zip file also contains a folder `2025_05_exam` with the following content:

- An empty Python file for each task, `<task_name>.py`, where `<task_name>` is the name of the task. These are the files where you should write your solutions and submit them at the end of the exam.
- A Python 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. 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 the tests 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 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 choosing the `2025_05_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. For the testing scripts to work, your solutions must be in the same folder as the testing scripts.

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:

- `bingo_card.py`
- `bouncing_ball.py`
- `experiment_numbers.py`
- `modified_blackjack.py`
- `number_statistics.py`
- `photo_details.py`
- `polygon_area.py`
- `running_speed.py`
- `two_means.py`

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

Task 1: Polygon Area

The area of a regular polygon with n sides, each of length s , is given by

$$A = \frac{ns^2}{4 \tan\left(\frac{\pi}{n}\right)}.$$

For example, the area of a regular polygon with five sides (a pentagon) and side length 4.31 is

$$A = \frac{5 \cdot 4.31^2}{4 \tan\left(\frac{\pi}{5}\right)} = \frac{92.8805}{2.9062} = 31.9598.$$

Write a function that takes the number of sides and the side length as input and returns the area of the regular polygon.

The desired behavior for a regular polygon with five sides and a side length of 4.13 is shown below.

```
>>> polygon_area(5, 4.31)
31.9597602410807
```

The filename and requirements are:

polygon_area.py

`polygon_area(n, s)`

Calculates the area of a regular polygon.

Parameters:

- `n` `int` The number of sides.
- `s` `float` The length of each side.

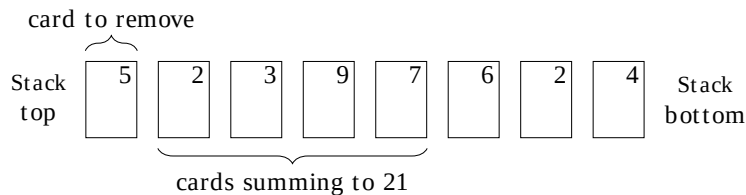
Returns:

- `float` The area of the polygon.

Task 2: Modified Blackjack

You have a stack of shuffled playing cards, each with a value between 1 and 11. You want to determine whether it's possible to remove n cards from the top of the stack so that the k cards that are now on top of the stack have values that sum exactly to 21.

For example, consider the sequence 5, 2, 3, 9, 7, 6, 2, 4, where the first value is the top card, as shown below. If we do not remove any cards, the top four cards have a sum of $5 + 2 + 3 + 9 = 19$, which is below 21, and adding the fifth card brings the sum to $5 + 2 + 3 + 9 + 7 = 26$ which is above 21 so $n = 0$ is not a solution. If we remove one card, the top four cards sum to $2 + 3 + 9 + 7 = 21$. So the combination $n = 1$ and $k = 4$ is a solution.



Write a function that takes, as input, a list of integers representing the values of the cards in the stack. The function should return two numbers, n and k . Here, n is the number of cards to remove, and k is the number of cards that sum to 21. If no solution exists, return -1 and -1 . If multiple solutions exist, return the one with the smallest n .

The desired behavior for the example case is shown below.

```
>>> modified_blackjack([5, 2, 3, 9, 7, 6, 2, 4])
(1, 4)
```

The filename and requirements are:

modified_blackjack.py

`modified_blackjack(cards)`

Finds the solution given a sequence of values.

Parameters:

- `cards` `list` A sequence of integers representing the values of cards in a stack.

Returns:

- `tuple` n and k according to the rules.

Task 3: Running Speed

Running pace is given in minutes per kilometer. For example, a pace of 5.5 minutes per kilometer means running one kilometer in 5.5 minutes. The inverse of pace is speed.

We want to convert a given pace to the speed in kilometers per hour and the speed in meters per second.

For example, for a pace of 5.5 minutes per kilometer, the speed is

$$\frac{1}{5.5} \frac{\text{km}}{\text{min}} = \frac{1}{5.5} \cdot 60 \frac{\text{km}}{\text{h}} = 10.9 \frac{\text{km}}{\text{h}}$$

and

$$\frac{1}{5.5} \frac{\text{km}}{\text{min}} = \frac{1}{5.5} \cdot \frac{1000}{60} \frac{\text{m}}{\text{s}} = 3.03 \frac{\text{m}}{\text{s}}$$

Write a function that takes a pace in minutes per kilometer and returns the speed in kilometers per hour and the speed in meters per second. The desired behavior for the pace 5.5 minutes per kilometer is shown below.

```
>>> running_speed(5.5)
(10.90909090909091, 3.03030303030307)
```

The filename and requirements are:

running_speed.py

`running_speed(pace)`

Given pace computes speed in km/h and m/s.

Parameters:

- `pace` `float` Pace in min/km.

Returns:

- `tuple` Speed in km/h and m/s.

Task 4: Photo Details

The filename of a photo includes the date it was taken, an identifying number, and a file extension. For example, in the filename `20250305_110031.jpg`, the first part, `20250305`, is a date in the format `YYYYMMDD` followed by an `_`. The second part, `110031`, is an identifying number for the photo, and `jpg` is the file extension. We need to extract these details from the filename.

Write a function that takes a filename in the format `YYYYMMDD_ID.EXT` and returns a *dictionary* with the following *keys* and *values*:

- `'year'`: the four characters representing the year (as a string)
- `'month'`: the two characters representing the month (as a string)
- `'day'`: the two characters representing the day (as a string)
- `'number'`: the characters of the identifying number (as a string)
- `'ext'`: the file extension (as a string, without the period)

You can see the desired behavior for the string `'20250305_110031.jpg'` below.

```
>>> photo_details('20250305_110031.jpg')
{'year': '2025', 'month': '03', 'day': '05', 'number': '110031', 'ext': 'jpg'}
```

Note that while the date format is always `YYYYMMDD`, the file identifier and file extension can vary in length. For instance, `'20250305_2151.jpeg'` is also a valid filename. You can assume that the identifying number is an integer.

The filename and requirements are:

photo_details.py

```
photo_details(filename)
```

Given a filename, extract the details of the photo.

Parameters:

- `filename` `str` The filename of the photo.

Returns:

- `dict` The details of the photo.

Task 5: Experiment Numbers

A number of experiments are represented as a list, for example

```
['A-1', 'A-3', 'B-2', 'A-4', 'B-1', 'B-3', 'N-4'].
```

Each element consists of an experiment name (a single letter), a dash, and a (positive) repetition number. For example, 'A-4' means that experiment A was repeated for the fourth time. The list is unordered, not all repetitions are necessarily present, and repetition numbers may be more than one digit.

Write a function that takes, as input, the list of experiments and returns a *dictionary* with experiment names as *keys*. Each *key* should have as *value* the highest repetition number found for that experiment, represented as an integer.

You can see the desired behavior below.

```
>>> experiment_numbers(['A-1', 'A-3', 'B-2', 'A-4', 'B-1', 'B-3', 'N-4'])
{'A': 4, 'B': 3, 'N': 4}
```

The filename and requirements are:

experiment_numbers.py

`experiment_numbers(experiments)`

Computes statistics about a list of experiments.

Parameters:

- `experiments` `list` List of experiments.

Returns:

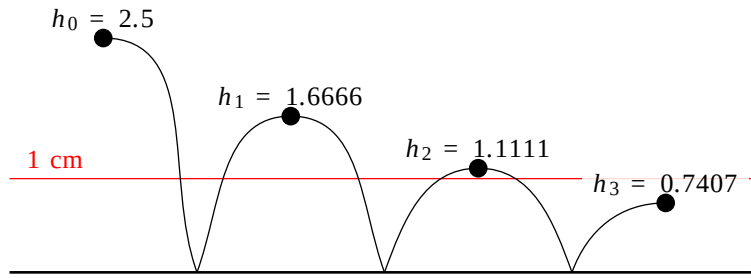
- `dict` Maximum repetitions for each experiment name.

Task 6: Bouncing Ball

A ball is dropped from a height of h_0 centimeters and allowed to bounce. After each bounce, it reaches a height that is two-thirds of the previous height. That is

$$h_{n+1} = \frac{2}{3}h_n$$

where h_n is the height after the n^{th} bounce. We want to determine how many times the ball bounces before the height it reaches is below 1 centimeter.



Consider the example from the illustration. The ball is dropped from a height of $h_0 = 2.5$ centimeters. The height after the first bounce is $h_1 = \frac{2}{3}2.5 = 1.666$ centimeters, after the second bounce $h_2 = \frac{2}{3}1.666 = 1.1111$ centimeters, and after the third bounce $h_3 = \frac{2}{3}1.1111 = 0.7407$ centimeters. Since h_3 is below 1 centimeter, the ball bounced three times before the height dropped below 1 centimeter.

Write a function that takes the initial height h_0 as input and returns the number of bounces before the height is below 1 centimeter.

The desired behavior for an initial height of 2.5 centimeters is shown below.

```
>>> bouncing_ball(2.5)
3
```

The filename and requirements are:

bouncing_ball.py

`bouncing_ball(h0)`

Calculates how many bounces a ball makes before bouncing less than 1 centimeter.

Parameters:

- `h0` `float` The initial height of the ball in centimeters.

Returns:

- `int` The number of bounces before the ball bounces less than 1 centimeter.

Task 7: Number Statistics

A specific type of text file contains lines, and each line contains integers separated by spaces. Given a number, we want to know how many times this number occurs in each line of the file.

For example, consider the file `files/numbers_1.txt` with the content:

```
7 4 8 12 11 9 18 4 25
9 16 47 2 26 74 57 33
4 7 9 8 3 6 5 8 6 4 3
```

and the number 8. The number 8 occurs once in the first line, no times in the second line, and twice in the last line. This result can be represented as the list `[1, 0, 2]`.

Write a function that, given a filename and a number, returns the list with the number of occurrences of this number in each line of the file. If there are empty lines in the file, they should be ignored. You can assume that exactly one space is between the numbers and no spaces before the first or after the last number. Note that the digit 8 in 18 should not be counted since we count integer numbers and not individual digits.

The expected output may be seen in the example.

```
>>> filename = 'files/numbers_1.txt'
>>> number_statistics(filename, 8)
[1, 0, 2]
```

The filename and requirements are:

`number_statistics.py`

`number_statistics(filename, number)`

Compute statistics.

Parameters:

- `filename` `str` The filename.
- `number` `int` The number to check for.

Returns:

- `list` The number of occurrences in each line.

Task 8: Two Means

Given some numbers and a threshold value, we create two subsets: numbers strictly below the threshold and numbers strictly above the threshold. We want to find the mean of each subset.

For example, consider the numbers

1.5, 8.5, 3.5, 4.5, 5.0, 6.5, 7.5, 2.5, 1.0

and the threshold 5.5. The numbers strictly below the threshold are 1.5, 3.5, 4.5, 5.0, 2.5, and 1.0, which have a mean of 3.0.

The numbers strictly above the threshold are 8.5, 6.5, and 7.5, which have a mean of 7.5.

Write a function that, given an array of numbers and a threshold, returns the mean of the numbers strictly below the threshold and the mean of the numbers strictly above the threshold. If one or both subsets are empty (i.e., no numbers below or above the threshold), return the threshold value instead of a mean for that subset.

The desired behavior is shown below.

```
>>> import numpy as np
>>> two_means(np.array([1.5, 8.5, 3.5, 4.5, 5.0, 6.5, 7.5, 2.5, 1.0]), 5.5)
(np.float64(3.0), np.float64(7.5))
```

The filename and requirements are:

two_means.py

`two_means(numbers, threshold)`

Computes two means.

Parameters:

- `numbers` `numpy.ndarray` Array of numbers.
- `threshold` `float` Threshold value.

Returns:

- `tuple` Two means.

Task 9: Bingo Card

Write a class definition for the class `BingoCard` that represents a card for a game of bingo. The constructor should, as input, take a list of numbers on the card. The `match` method should take a number as input and, if the number is on the card, keep a record of this number being matched. It should return `True` if the number was matched and `False` otherwise. The `unmatched` method should return `True` if any numbers remain unmatched and `False` if all numbers have been matched. The `reset` method should clear the matched numbers so the card can be used again.

You can assume that the class is used as bingo is typically played. That is, `match` will not be called with the same number more than once before the card is reset, a number will not appear twice on a card, and the card will not be empty. Consider the example below.

```
>>> card = BingoCard([15, 27, 73])
>>> card.match(27)
True
>>> card.match(53)
False
>>> card.match(73)
True
>>> card.unmatched()
True
>>> card.match(15)
True
>>> card.unmatched()
False
>>> card.reset()
>>> card.unmatched()
True
```

In this example, a bingo card is created with the numbers 15, 27, and 73. When matching the number 27, `True` is returned because the number is on the card. Next, 53 is not found on the card, and 73 is found on the card. Checking whether any numbers are still unmatched, the method returns `True` because 15 is still not matched. Then, 15 is matched followed by `match` which returns `False` as all numbers have been matched. The card is reset, and now some numbers are unmatched.

The filename and requirements are:

bingo_card.py

`BingoCard()`

A class representing a bingo card.

`__init__(numbers)`

Initializes the card with the given numbers.

Parameters:

- `numbers` `list` The numbers on the card.

`match(number)`

Match the number with the card.

Parameters:

- `number` `int` Number to match.

Returns:

- `bool` The number was matched.

`unmatched()`

Check whether any unmatched numbers are left on the card.

Returns:

- `bool` There are unmatched numbers.

`reset()`

Resets the card.

Task 10: Informative Bingo Card

We want to create a subclass of the `BingoCard` class from Task 9 that gives more informative feedback.

The `unmatched` method should return how many numbers on the card have not yet been matched.

Write the class definition for the subclass `InformativeBingoCard`, which inherits from `BingoCard`. Modify the necessary methods to incorporate this additional behavior and inherit the unchanged methods from the parent class.

You must write the class definition for `InformativeBingoCard` in the same file as the class definition for `BingoCard`.

Refer to the example below for the expected behavior.

```
>>> card = InformativeBingoCard([15, 27, 73])
>>> card.match(27)
True
>>> card.match(53)
False
>>> card.match(73)
True
>>> card.unmatched()
1
>>> card.match(15)
True
>>> card.unmatched()
0
>>> card.reset()
>>> card.unmatched()
3
```

In this example, an informative bingo card is created with the numbers 15, 27, and 73. When matching the number 27, `True` is returned because the number is on the card. Next, 53 is not found on the card, and 73 is found on the card. Checking whether numbers are still unmatched, the method returns `1` because 15 is still not matched. Then, 15 is matched, and checking whether any numbers are unmatched, the method returns `0`. The card is reset, and now all three numbers are unmatched.

The filename and requirements are:

bingo_card.py

`InformativeBingoCard()`

A class representing a bingo card that provides more information.

`unmatched()`

Check how many numbers are unmatched.

Returns:

- `int` The number of unmatched numbers.