Week 5: Lists

Contents

- Exercise 5.1: Warm-up with lists
- Exercise 5.2: Average of a list
- Exercise 5.3: Conditional maximum
- Exercise 5.4: Find the tallest animal
- Exercise 5.5: Short words
- Exercise 5.6: Check if the list is rectangular
- Exercise 5.7: Vector addition
- Exercise 5.8: Count Multiples
- Exercise 5.9: Create list containing multiples
- Exercise 5.10: Water height
- Exercise 5.11: Best buy
- Exercise 5.12: Tic Tac Toe - Printing the board
- Exercise 5.13: Tic Tac Toe - Update board
- Exercise 5.14: Tic Tac Toe - Get game state!
- Exercise 5.15: Tic Tac Toe - Putting everything together!

Important

First, `download_week05.py` and run the script in `VSCode`. You may need to confirm the download in your browser.

The script should end with `Successfully completed!`. If you encounter an error, follow the instructions in the error message. If needed, you can proceed with the exercises and seek assistance from a TA later.

Note

Many of the following exercises can also be solved with NumPy arrays. This has some advantages, but also some disadvantages compared to lists. We will cover NumPy arrays in a later lecture. For this exercise, please use lists.

Exercise 5.1: Warm-up with lists

Before you start with the exercises, try out some simple list operations:

- Create a list of numbers and print it.
- Reverse the list
- Sort the list
- Add a number to the list
- Remove a number from the list (try the last number, the first number, and a number in the middle)
- Create a list of strings and print it.
- Only keep the first three elements of the list
- Assign a new value to the second element of the list
- Try out what happens when you assign the same list to two different variables and then modify one of the variables. Why is this important to keep in mind?

Tip

Skip to main content

Exercise 5.2: Average of a list

Exercise 5.2

In this first exercise, you will write a function that calculates the average of a list of numbers. You can assume that the input list contains one or more numbers.

Test your function on a few examples. For instance, the average of the list `[4, 2, 1, 3]` should be `2.5`.

cp.ex05.average.calculate_average(*nums*)

Return the average of a list of numbers.

Parameters:
nums (`list`) – list of numbers

Return type:
`float`

Returns:
average of list

You can run the provided tests by inserting the function into `cp/ex05/average.py`.

Exercise 5.3: Conditional maximum

Exercise 5.3

In this exercise, you are given a list of numbers and a threshold. You want to find the largest number that is strictly smaller than the threshold. If no number is smaller than the threshold, your function should return `0`.

Try a few examples:

```
>>> a = [1, 2, 3, 4, 5]
>>> conditioned_maximum(a, 3)
2
>>> conditioned_maximum(a, 0)
0
>>> conditioned_maximum(a, 10)
5
```

cp.ex05.conditioned_maximum.conditioned_maximum(*nums, threshold*)

Return the maximum of a list of numbers, that is strictly smaller than the given threshold.

Parameters:
- **nums** (`list`) – list of numbers
- **threshold** (`float`) – limit for maximum value

Return type:
`float`

Returns:
maximum of list thats smaller than the threshold

Insert your function into `cp/ex05/conditioned_maximum.py` to test it.

Exercise 5.4: Find the tallest animal

Exercise 5.4

Imagine you are given a list of heights of animals, and you want to find the name of the tallest animal that's still smaller than a

Skip to main content

smaller than a given height. You can assume that the list of heights and the list of names have the same length, and there is always a number smaller than the threshold.

You can use these lists to test your function:

```
>>> heights = [1.5, 1.6, 5.5, 3.0, 2.5, 1.0, 1.4]
>>> names = ['cow', 'horse', 'giraffe', 'elephant', 'dinosaur', 'dog', 'deer']
>>> conditioned_maximum_name(heights, names, 2.0)
'horse'
>>> conditioned_maximum_name(heights, names, 3.0)
'dinosaur'
```

Optional: try writing a new function that takes the name of an animal as input and returns the name of the largest animal that's smaller than the given animal.

cp.ex05.conditioned_maximum.conditioned_maximum_name(*nums*, *names*, *threshold*)

Return the name of the maximum of a list of numbers, that is smaller than the given threshold.

Parameters:

- **nums** (`list`) – list of numbers with animal heights.
- **names** (`list`) – list of animal names
- **threshold** (`float`) – limit for maximum value

Return type:

`str`

Returns:

animal name with the corresponding maximum height, which is shorter than the threshold height.

You can run the provided tests by inserting your functions into `cp/ex05/conditioned_maximum.py`. (Note: If you solve the optional exercise, the tests will fail.)

Exercise 5.5: Short words

Exercise 5.5

In this exercise, you are given a string of words, and you want to find all words that are shorter than a given length. Your function should return a string of all shorter words separated by a whitespace `' '`.

Try a few examples:

```
>>> s = 'This is a sentence with some short and some longlonglong words.'
>>> short_words(s, 4)
'is a and'
>>> short_words(s, 2)
'a'
>>> short_words(s, 10)
'This is a sentence with some short and some words.'
```

Tip

cp.ex05.short_words.short_words(*words*, *max_len*)

Return a string of words that are shorter than max_len.

Parameters:

- **words** (`str`) – string of words
- **max_len** (`int`) – maximum length of words

Return type:

`str`

Returns:

string of words that are shorter than max_len

Skip to main content

You can run the provided tests by inserting your functions into `cp/ex05/short_words.py`.

Exercise 5.6: Check if the list is rectangular

Exercise 5.6

In this exercise, you want to check whether a nested list is rectangular. A nested list is rectangular if all sublists have the same length. Your function should return `True` if the nested list is rectangular and `False` otherwise.

Try a few examples:

```
>>> is_rectangular([[1, 2, 3], ['cat', 'dog', 'mouse']]  )
True
>>> is_rectangular([[1, 2, 3], [1, 2, 3, 4]])
False
>>> is_rectangular([[1, 2, 3], ['cat', 'dog', 'mouse'], [3, 4, 3]])
True
>>> is_rectangular([[1, 2, 3], ['cat', 'dog', 'mouse'], []])
False
```

You can test your function by inserting it into `cp/ex05/rectangular.py`.

cp.ex05.rectangular.is_rectangular(*nested_list*)

Check if a list is rectangular.

Parameters:

nested_list (`list`) – nested list

Return type:

`bool`

Returns:

True if list is rectangular, False otherwise

Exercise 5.7: Vector addition

Exercise 5.7

In this exercise, you will add two lists elementwise. You can assume that the two lists have the same length. Make sure that you define a new list for the result and that you're not modifying the input lists.

Note

This corresponds to adding two vectors in linear algebra. Here, we represent the vectors as lists of numbers. Here is a simple example:

$$\begin{pmatrix}1\\2\\3\end{pmatrix}+\begin{pmatrix}4\\5\\6\end{pmatrix}=\begin{pmatrix}5\\7\\9\end{pmatrix}$$

Test your function on a few examples and make sure to check if your input vectors are still the same.

Test a few examples to see if your function works as expected:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> print(a,'+',b,'=',vector_add(a, b))
[1, 2, 3] + [4, 5, 6] = [5, 7, 9]
```

Skip to main content

Optional: What happens if you have two lists of different lengths? Try including a check for length and print an error message if the lists have different lengths.

`cp.ex05.vector_add.vector_add(v, w)`

Add two vectors.

Parameters:
- **v** (`list`) – vector 1 (list of numbers)
- **w** (`list`) – vector 2 (list of numbers, same length as v)

Return type:
`list`

Returns:
sum of v and w (list of number)

You can test your function by inserting it into `cp/ex05/vector_add.py`.

Exercise 5.8: Count Multiples

Exercise 5.8

In this exercise, you will write a function that counts how many multiples of a given number "m" are in a list of numbers. A multiple of a number "m" is any number that can be divided by m with a remainder of 0. You should return the count of multiples within the list. You can assume that the list contains only numbers.

Test a few examples to see if your function works as expected:

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> count_multiples(a, 2)
4
>>> count_multiples(a, 3)
3
>>> count_multiples(a, 5)
1
>>> count_multiples(a, 10)
0
```

`cp.ex05.multiples.count_multiples(numbers, m)`

Count the number of numbers that are divisible by m.

Parameters:
- **numbers** (`list`) – list of numbers
- **m** (`int`) – number to divide by

Return type:
`int`

Returns:
number of numbers that are divisible by m

You can test your function by inserting it into `cp/ex05/multiples.py`.

Exercise 5.9: Create list containing multiples

Exercise 5.9

Now, you are not interested in the number of multiples, but in the multiples themselves. Write a function that returns a list of all multiples of a given number in a list. You can assume that the list contains one or more numbers.

Test a few examples to see if your function works as expected:

Skip to main content

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> multiples_list(a, 2)
[2, 4, 6, 8]
>>> multiples_list(a, 3)
[3, 6, 9]
>>> multiples_list(a, 5)
[5]
>>> multiples_list(a, 10)
[]
```

cp.ex05.multiples.multiples_list(*numbers*, *m*)

Make a list with all numbers from a list that are divisible by m.

Parameters:

- **numbers** (`list`) – list of numbers
- **m** (`int`) – number to divide by

Return type:

`list`

Returns:

list of numbers that are divisible by m

You can test your function by inserting it into `cp/ex05/multiples.py`.

Note

The fo
Projec
link co
includ

Exercise 5.10: Water height

The height of the water in a pond changes daily, governed by two contributions: the constant decrease of height due to the water outflow and the variable increase of height due to the rain. Given the water height for one day and the rain value r, the water height for the next day can be computed as

$$h_{today} = h_{yesterday} + r - 2$$

If the computed value is negative, it should be replaced by 0, indicating that the pond is empty. The computation can be repeated for any number of days as long as we have rain values.

Write a function that takes a list of rain values and a starting water height and returns the final water height. The list of rain values can be of any length, including zero length.

Does your function return the correct value for the following examples?

```
>>> water_height(0, [1])
0
>>> water_height(0, [4, 2, 3])
3
>>> water_height(5, [1, 0, 2, 1])
1
```

Add your solution to `cp/ex05/water_height.py` to get it graded for the project.

cp.ex05.water_height.water_height(*h0*, *r*)

Calculate the water height after multiple days of rain.

Param:

h0: initial height of the water

Param:

r: list of rain showers

Return type:

`int`

Returns:

height of water after days of rain

Skip to main content

Exercise 5.11: Best buy

You are given a list of prices for items you want to buy and some money. How many items can you buy?

You should be able to start at any index in the list and work your way forwards. You buy the first item you can afford, then the second, and so on. If you run out of money or reach the end of the list, you stop. You should also be able to go through the list backwards in the same manner, stopping if you reach the start of the list.

Tip

Write a function that takes a list of prices and the amount of money you have and returns the number of items you can buy. Your function should be able to handle all three scenarios.

Test a few examples to see if your function works as expected:

```
>>> prices = [3, 2, 1, 3, 5]
>>> best_buy(prices, 10, 0, False)
4
>>> best_buy(prices, 3, 1, False)
2
>>> best_buy(prices, 8, 4, True)
2
>>> best_buy(prices, 10, 4, True)
3
```

Add your solution to `cp/ex05/best_buy.py` to get it graded for the project.

cp.ex05.best_buy.best_buy(*prices*, *money*, *start_index*, *reverse*)

Return the number of items that can be bought with the given amount of money. The function should be able to handle arbitrary starting points and to run the list in reverse.

Parameters:

- **prices** (`list`) – list of prices
- **money** (`int`) – amount of money
- **start_index** (`int`) – starting index in list
- **reverse** (`bool`) – boolean to indicate if list should be run in reverse

Return type:

`int`

Returns:

number of items that can be bought with the given amount of money

Exercise 5.12: Tic Tac Toe - Printing the board

Starting from this exercise and continuing through the exercises for the week, you will be tasked with writing multiple functions to create a Python implementation of the game Tic Tac Toe using lists. The rules of Tic Tac Toe are straightforward:

- There are two players, `'X'` and `'O'`.
- The game is played on a 3×3 board.
- The players take turns placing their symbol on the board.
- The first player to get three of their symbols in a row wins.
- If no player succeeds in getting three of their symbols in a row, and there are no empty spaces, the game is a draw.

For this implementation, you can assume that the game board is represented as a nested list (a list of lists) of strings. Each string in the nested list can be one of three values: `'X'`, `'O'`, or `'-'` indicating an empty position.

The moves in the game are specified using a list of two integer indices, referred to as `position`. The first integer indicates index of the row, and the second integer indicates the index of the column where the player wants to make their move on the board.

Start by implementing a function `print_board()` that takes as input a board and prints the current game board.

Skip to main content

```
>>> board = [['X', 'X', 'X'], ['-', 'O', '-'], ['-', 'O', '-']]
>>> print_board(board)
XXX
-O-
-O-
```

cp.ex05.tictactoe.print_board(*board*)

Print the current state of the game board.

Parameters:

board (`list`) – List of lists of strings representing the game board.

Add your solution to `cp/ex05/tictactoe.py`.

Exercise 5.13: Tic Tac Toe - Update board

Now, let's create a function called `update_board()` that allows a player to make a move in the game of Tic-Tac-Toe. This function should take three inputs:

- The current state of the game board.
- The player who is making the move (either `'X'` or `'O'`).
- The position where the player wants to make their move.

The primary purpose of this function is to update the game board with the player's move, but it should also check that the intended player position is empty so that the board is updated consistently, for a fair game of Tic-Tac-Toe. The function should return the updated game board with the last move played by the specified player. If an invalid move is played, the function should print `'Invalid move!'` and not update the board.

```
>>> board = [['X', 'X', 'X'], ['-', 'O', '-'], ['-', 'O', '-']]
>>> update_board(board, 'O', [2, 2])
[['X', 'X', 'X'], ['-', 'O', '-'], ['-', 'O', 'O']]
>>> update_board(board, 'O', [0, 0])
Invalid move!
[['X', 'X', 'X'], ['-', 'O', '-'], ['-', 'O', 'O']]
```

cp.ex05.tictactoe.update_board(*board*, *player*, *position*)

Update the game board with the player's move.

Parameters:

- **board** (`list`) – List of lists of strings representing the game board.
- **player** (`str`) – Player making the move ('X' or 'O').
- **position** (`list`) – List containing two integer indices [row, column] indicating the position to make a move.

Return type:

`list`

Returns:

Updated game board after the move.

Add your solution to `cp/ex05/tictactoe.py`.

Exercise 5.14: Tic Tac Toe - Get game state!

Let's proceed by implementing a function named `get_game_state()` to determine the game state of the Tic-Tac-Toe game. This function should take the game board as input and assess whether the game has been won by one of the players, if it has resulted in a draw, or if it is still ongoing.

To achieve this, the function will examine the board to check if any player has achieved three symbols in a row, either horizontally, vertically, or diagonally. The function should return:

- `'X'` if player `'X'` has won,

Skip to main content

- `'Draw'` if the game is a draw (all positions are filled by players' symbols),
- `'-'` if the game is still ongoing.

```
>>> winning_board = [['X', 'X', 'X'], ['-', 'O', '-'], ['-', 'O', '-']]
>>> get_game_state(winning_board)
'X'
>>> ongoing_board = [['X', 'O', '-'], ['-', '-', '-'], ['-', '-', '-']]
>>> get_game_state(ongoing_board)
'-'
```

cp.ex05.tictactoe.get_game_state(*board*)

Check if a player has won the game, if it's a draw, or if it's ongoing.

Parameters:

board (`list`) – List of lists of strings representing the game board.

Return type:

`str`

Returns:

A string which is 'X' if player 'X' won, 'O' if player 'O' has won, 'Draw' if the game is a draw, or '-' if the game is ongoing.

Add your solution to `cp/ex05/tictactoe.py`.

Exercise 5.15: Tic Tac Toe - Putting everything together!

Finally, implement a function `tictactoe()` that combines the functions above. Your function should take the `board` (list of lists), the `player` (`'X'`, `'O'`) and the `position` (list) of the next move as input. It should return and print the updated board. In the case that there is a winner it should print `'Player X won!'` or `'Player O won!'`. You can use the function iteratively to play Tic-Tac-Toe as in the following example:

```
>>> board = [['-', '-', '-'], ['-', '-', '-'], ['-', '-', '-']]
>>> board = tictactoe(board, 'X', [0, 1])
-X-
---
---
>>> board = tictactoe(board, 'O', [0, 2])
-XO
---
---
>>> board = tictactoe(board, 'X', [1, 0])
-XO
X--
---
>>> board = tictactoe(board, 'O', [1, 1])
-XO
XO-
---
>>> board = tictactoe(board, 'X', [1, 1])
Invalid move!
-XO
XO-
---
>>> board = tictactoe(board, 'X', [1, 2])
-XO
XOX
---
>>> board = tictactoe(board, 'O', [2, 0])
-XO
XOX
O--
Player O won!
```

Add your solution to `cp/ex05/tictactoe.py` to get it graded for the project.

cp.ex05.tictactoe.tictactoe(*board*, *player*, *position*)

Play a move in the Tic-Tac-Toe game and determine the winner.

Parameters:

- **board** (`list`) – List of lists of strings representing the game board.

Skip to main content

- **position** (`list`) – List containing two integer indices [row, column] indicating the position to make a move.

Return type:

`list`

Returns:

Updated game board after the move.