Week 12 | Loops and Methods

Exercise 1: Quizzes

Complete the quiz for this week on Moodle. Remember that you can ask questions in the workshop.

Exercise 2: Prime Testing

> Requisites
>
> Download `is_prime.py` from Moodle.

Implement the function `def is_prime(n)`, which returns whether the integer `n` and is a prime or not. For example, for `3`, `5`, and `2971215073`, it should return `True`; for `4`, `10`, or `10203501230` it should return `False`.

Extend your program to read an integer from the user and instead print all primes that are smaller than the given number.

Exercise 3: Decoding

While entering data into an Excel table, an intern unfortunately sometimes entered `l` (lower case L) instead of `1` (one), and `O` (upper case o) instead of `0` (zero). Write a program that reads a sequence of "numbers" in one line, e.g. `10 87120220 O112231`, fixes the mistakes, and then prints out the fixed sequence.

Hint: Use the methods `split` and `replace`.

Extend your program to also print the average value of the numbers. For example, given the input `10 11 l2`, your program should print `10 11 12` and `Average: 11.0`.

PS: We will see how to interact with tables exported from Excel in the future. (For the experienced ones: look up the modules `csv` and `xlsxwriter` / `openpyxl`.)

Exercise 4: Reading a Webpage

> Requisites
>
> Download `webpage.py` from Moodle

In this exercise, we will analyse the content of webpages. Concretely, for now we simply want to know if some text appears in a webpage. We will use a *module* to handle downloading the website data – you do not need to understand how it works for now, we will learn about modules later.

Start from the provided skeleton. This already includes code for downloading and printing the *HTML code* of `https://google.com`. (If this looks weird, here is a relevant xkcd comic: `https://xkcd.com/1605/`.) Try to see if this works for you by running `python webpage.py`.

1. Now, modify the code so that the user can enter a URL and fetch this site instead. You don't need to care about handling errors etc.

2. Check that the URL starts with `http://` or `https://`. If not, add it to the string. Use the `.startswith` method. (For example, `google.com` should be turned into `https://google.com`.)
3. Move fetching the website data into a separate function, so that you can write something akin to `content = download_data(url)`.
4. Check if the website content contains the word `div`.

   Bonus: Count how often it appears.
5. Modify the code so that the user can enter the URL and the string to search for.
6. Modify the code again so that the user once enters a URL, and then can enter multiple strings, where your program every time responds whether the website contains that word (or how often).

> ★ Bonus
>
> Search the content of the webpage for URLs (using, for example, regular expressions). You can try to connect to these pages, too, again download their content, search for links, etc. With this, you effectively build a web-crawler.

Exercise 5: Tic-Tac-Toe (Bonus)

> ℹ Info
>
> Even more than other bonus tasks, this is an *advanced* exercise, only for students who are already rather familiar with Python (or those who like a challenge).

Implement a tic-tac-toe game. Concretely, your program should:

1. Print the current state of the board and whose turn it is,
2. ask for the move of the current player (e.g. by asking for a number from 1 to 9),
3. check that the turn is valid (i.e. the mark is placed in an empty cell),
4. check if this was a "winning" move and, if so, end the game, otherwise
5. go back to step 1.

Write an AI to play against. Concretely, let the AI be the second player. Whenever it is the turn of the AI, it should check if there is a way to *ensure* winning, i.e. for each possible move that you can make there is a winning response. You can check this (roughly) as follows: First, implement a function that given the state of the game is supposed to check if there is a move available to the AI with which it can win, as follows: Go over each of the available moves. For each of the moves, "pretend" to play it, then check if the AI has won. If yes, then there definitely is a winning move. If there is a draw, then this move is not winning. Otherwise, go over all possible moves of the opponent, pretend to play the opponent's move, and then ask the same question again (i.e. call your function recursively). Now, if such a move exists, then pick that move. If this is not possible, the AI should play a move where winning is at least still possible if you play sub-optimally (i.e. instead of winning against *all* moves, there at least is a one way in which the AI could still win). If neither is possible, the AI just picks any move (the game is effectively over already, anyway).

Bonus: 1) If victory is not certain, pick the move which has the highest number of winning outcomes, i.e. where the fewest numbers of mistakes on your side are needed for the AI to win. 2) Once the AI knows it cannot win, no matter what you do, let it give up and end the game early.

To make things a bit more human, add a bit of randomness to your AI. In particular, instead of always playing winning moves if possible, add a small chance for it to play a random (valid) move.