Advent of Code 2024

December 3, 2024

Welcome to my solution repository for the Advent of Code edition 2024! This repository contains (hopefully) a solution to all exercises. The solutions should be simple enough that anyone with a minor Python knowledge, but a solid internet connection and a curious mind should be able to comprehend all of them. I did my best to implement them in the most readable, yet elegant and *pythonic* way I could imagine. Below, we will quickly revise each of the solutions, building upon what we learned in previous quests. *'T is the season!*

1 Day 1: *Historian Hysteria*

The first two quests are simple, but require knowledge on how to read files in Python. I implemented the solution in the files *aoc_01.py* and *aoc_02.py*, however, it's best practice to keep the interaction with your file system as short as possible. That is why I always declared and implemented the input handling functions in *aoc_utils.py*.

For this first day, we're looking at the function `read_lists_01` (part of the utilities file), which reads the contents of an input file into two lists *left_list* and *right_list*.

```python
def read_lists_01(input_file="data/24-aoc-01.in"):
    left_list = []
    right_list = []

    with open(input_file, "r") as file:
        for line in file:
            values = line.split(' ')

            left_list.append(int(values[0]))
            right_list.append(int(values[-1]))

    left_list.sort()
    right_list.sort()

    return left_list, right_list
```

This function has some features that might look unfamiliar to those new to Python.

- The function's declaration shows that it takes one argument, called `input_file`. We also see that this argument is set to *"data/24-aoc-01.in"* if it is omitted. This means that if we call the function without specifying an input file, this argument will be set to that standard value.
- We are opening a file in *read-only* mode. This is done with the `with open(filename, mode) as` statement. This statement opens a file in the required mode (*write*, *read*, *append*) and gives you a handle to the file. With this handle, you can use the file as an iterator which returns the lines of the file in-order. There's some other functionality for these file handles, but that would take us too far for this day.

This input handling function is used for both the first and second quest, so naturally I put it in the utility file. I can import this function in another file. This is because the directory is a python package, but I won't go into the details of those right now. It's sufficient to know that you can import these functions as well when you create new files in the same directory as the utility file.

Let's now check how we fix the first problem. This boils down to sorting both lists - which is being handled in the input handling function - and then taking the absolute differences between their minimal values, adding these all together to produce the final result.

```
from aoc_util import read_lists_01


def find_total_distance(input_file="data/24-aoc-01.in"):
    """
    >>> find_total_distance("test/24-aoc-01.test")
    11
    """
    left_list, right_list = read_lists_01(input_file)
    total_distance = sum(abs(left - right) for left, right in zip(left_list, right_list))
    return total_distance
```

Nothing special happens here, once you understand *list comprehensions* like the one on the second line in `find_total_distance`'s implementation. I won't go into the intricate details of these types of statements, but it should be apparent that they are equivalent to writing a for loop, yet in this case more readable! For my personal input, you can check the result below. Watch out when re-running the original notebook, as the file used might not exist on your system, making the execution crash!

[1]:
```
from aoc_01 import find_total_distance
find_total_distance()
```

[1]:
```
2166959
```

Once you get your result correct, you are treated with the second quest of day one. As is typical, this is a small change made to the assignment of the first task. In this case, we are tasked with calculating some *similarity score*. The explanation boils down to multiplying each element in the first list by the amount of appearances it has in the second list. Using the same input handling function, this is very simple:

```python
from aoc_util import read_lists_01

def find_total_similarity_score(input_file="data/24-aoc-01.in"):
    """
    >>> find_total_similarity_score("test/24-aoc-01.test")
    31
    """
    left_list, right_list = read_lists_01(input_file)
    total_similarity = sum(left * right_list.count(left) for left in left_list)
    return total_similarity
```

Once again, nothing out of the ordinary to mention here but a nice and simple list comprehension. My input yields the following result:

```
[2]: from aoc_02 import find_total_similarity_score
     find_total_similarity_score()
```

```
[2]: 23741109
```

Awesome! Let's quickly check what new challenges the second day brings us!

2 Day 2: *Red-Nosed Reports*

The second day brings us two new quests. This time, we are asked to inspect some sensor data, which is presented in a list of integer sequences. Depending on the values and the way they are sorted, these sequences are considered safe or unsafe. Our task is to efficiently figure out which reports indicate safe or unsafe levels. The solution to these assignments can be found under *aoc_03.py* and *aoc_04.py*.

Once again, we start by reading the input from a file. This code is again declared and implemented in *aoc_util.py*. It is a tad bit simpler, as we only have to generate one list of sequences instead of splitting a line over two lists.

```python
def read_reports_03(input_file="data/24-aoc-03.in"):
    reports = []

    with open(input_file, "r") as file:
        for line in file:
            report = line.split(' ')
            reports.append([int(level) for level in report])

    return reports
```

Once we have our reports, we can start figuring out which of the reports indicate a safe series of levels or not. For a report to be safe, it has to conform two constraints in the beginning:

- The levels have to be in either de- or ascending order.
- Two subsequent levels have to differ at least one and at most three in absolute value.

To check this, I made use of the `all` key-word in my first implementation. Combining this with a list comprehension where I pair each value of the list with it's subsequent value makes checking the constraints very efficient.

```
from aoc_util import read_reports_03

def is_safe(report):
    all_increasing = all(
        level <= next_level
        for level, next_level
        in zip(report, report[1:])
    )

    all_decreasing = all(
        level >= next_level
        for level, next_level
        in zip(report, report[1:])
    )

    within_bounds = all(
        1 <= abs(level - next_level) <= 3
        for level, next_level
        in zip(report, report[1:])
    )

    return (all_increasing or all_decreasing) and within_bounds

def find_safe_reports(input_file="data/24-aoc-03.in"):
    """
    >>> find_safe_reports("test/24-aoc-03.test")
    2
    """
    reports = read_reports_03(input_file)
    safe_reports = sum(is_safe(report) for report in reports)
    return safe_reports
```

Once again, there is nothing out of the ordinary. We broke up the assignment in intuitive, yet simple and atomically solvable parts. By doing this, checking the constraints became almost trivial, as each of those `all`-statements almost read in human language!

The second assignment adds, as expected, an extra layer to the first. This time, we should allow one faulty level, meaning there can be, at most, one level that breaks the constraints. The key to the solution is realizing that when we omit this level, the constraints should be satisfied. We can reuse our `is_safe` function! The implementation on the next page defines an extra function.

```python
from aoc_util import read_reports_03
from aoc_03 import is_safe

def is_almost_safe(report):
    safe = False

    idx = 0
    while idx < len(report) and not safe:
        truncated_report = report[0:idx] + report[idx + 1:]
        safe = is_safe(truncated_report)
        idx += 1

    return safe


def find_safe_and_almost_safe_reports(input_file="data/24-aoc-03.in"):
    """
    >>> find_safe_and_almost_safe_reports("test/24-aoc-03.test")
    4
    """
    reports = read_reports_03(input_file)
    safe_reports = sum(is_almost_safe(report) for report in reports)
    return safe_reports
```

This implementation makes use of list slicing to efficiently omit the i-th value in iteration i. By having a *while* with two conditions, we can stop once we find a way in which our report is safe. Using these solutions, we get the following values:

```
[3]: from aoc_03 import find_safe_reports
     find_safe_reports()
```

```
[3]: 534
```

```
[4]: from aoc_04 import find_safe_and_almost_safe_reports
     find_safe_and_almost_safe_reports()
```

```
[4]: 577
```

As can be expected, there are more almost safe *and* safe reports than only safe reports.

Awesome! We just solved the assignments for the second day. One step closer to saving Christmas this year, all while learning some great skills!