day01

December 1, 2023

1 Day 1

1.1 Trebuchet?!

Something is wrong with global snow production, and you've been selected to take a look. The Elves have even given you a map; on it, they've used stars to mark the top fifty locations that are likely to be having problems.

You've been doing this long enough to know that to restore snow operations, you need to check all fifty stars by December 25th.

Collect stars by solving puzzles. Two puzzles will be made available on each day in the Advent calendar; the second puzzle is unlocked when you complete the first. Each puzzle grants one star. Good luck!

You try to ask why they can't just use a weather machine ("not powerful enough") and where they're even sending you ("the sky") and why your map looks mostly blank ("you sure ask a lot of questions") and hang on did you just say the sky ("of course, where do you think snow comes from") when you realize that the Elves are already loading you into a trebuchet ("please hold still, we need to strap you in").

As they're making the final adjustments, they discover that their calibration document (your puzzle input) has been **amended** by a very young Elf who was apparently just excited to show off her art skills. Consequently, the Elves are having trouble reading the values on the document.

The newly-improved calibration document consists of lines of text; each line originally contained a specific **calibration value** that the Elves now need to recover. On each line, the calibration value can be found by combining the **first digit** and the **last digit** (in that order) to form a single **two-digit number**.

For example:

```
1abc2
pqr3stu8vwx
a1b2c3d4e5f
treb7uchet
```

In this example, the calibration values of these four lines are 12, 38, 15, and 77. Adding these together produces **142**.

Consider your entire calibration document. **What is the sum of all of the calibration values?**

[1]:
```
## Python Imports

import re
import string

from pathlib import Path
```

1.2 Part One

It appears as though there's a simple initial solution: find all the instances of digits `0` to `9` and their locations in the line, then concatenate the first and last of these, and turn it into an `int`. Then sum the resulting `int`s from each line.

The approach retains only characters in the line that are in `string.digits`, and adds the `int` corresponding to the concatenated first and last digit to a list (`vals`) before returning it.

Maybe not the quickest approach, but it's relatively direct (despite the separate functions).

[2]:
```
def load_data(path):
    """Return generator of (whitespace-stripped) lines from input."""
    with path.open() as ifh:
        return (_.strip() for _ in ifh.readlines())

def parse_data(data):
    """Return list of integers made from of first and last digit in each line.
    """
    vals = []
    for line in data:
        digits = [char for char in line if char in string.digits]
        vals.append(int(digits[0] + digits[-1]))
    return vals
```

[3]:
```
data = load_data(Path("data/day01_test.txt"))
vals = parse_data(data)
sum(vals)
```

[3]:
```
142
```

[4]:
```
data = load_data(Path("data/day01_data.txt"))
vals = parse_data(data)
sum(vals)
```

[4]:
```
54304
```

And this seems to work.

1.3 Part Two

Your calculation isn't quite right. It looks like some of the digits are actually spelled out with letters: one, two, three, four, five, six, seven, eight, and nine also count as valid "digits".

Equipped with this new information, you now need to find the real first and last digit on each line. For example:

```
two1nine
eightwothree
abcone2threexyz
xtwone3four
4nineeightseven2
zoneight234
7pqrstsixteen
```

In this example, the calibration values are 29, 83, 13, 24, 42, 14, and 76. Adding these together produces **281**.

What is the sum of all of the calibration values?

This reframed problem has some catches. We need to (i) handle multiple occurrences of the same spelled-out digit, (ii) translate spelled-out numbers into digits, and (iii) handle cases where spelled-out numbers overlap, e.g. `threeight` or `zerone`. That causes problems for some of the Python builtin functions, such as `re.finditer` and `re.findall`, which are efficient ways to match strings, but only find non-overlapping matches.

The solution below uses a regular expression to match spelled-out digits, and the construction `(?=)` to avoid consuming characters in any overlapping strings. For instance `re.findall("zero|one", "zerone")` returns `['zero']`, because of the overlap. Asking `re` not to consume the last letter in `zero`, using `re.findall("zer(?=o)|one", "zerone")` returns `['zer', 'one']`, and it's then straightforward to match the truncated string to its corresponding digit in the dictionary `digdict`.

[5]:
```
digre = re.compile("0|1|2|3|4|5|6|7|8|9|zer(?=)o|on(?=e)|tw(?=o)|thre(?
 ↪=e)|four|fiv(?=e)|six|seve(?=n)|eigh(?=t)|nin(?=e)")

digdict = {"zero": "0", "on": "1", "tw": "2", "thre": "3", "four": "4", "fiv":␣
 ↪"5", "six": "6", "seve": "7", "eigh": "8", "nin": "9",
           "0": "0", "1": "1", "2": "2", "3": "3", "4": "4", "5": "5", "6": "6",␣
 ↪"7": "7", "8": "8", "9": "9"}

def parse_data(data):
    """Return list of integers made from of first and last digit in each line.
 ↪"""
    vals = []
    for line in data:
        diglist = []  # ordered list of positions of digits
        for match in re.finditer(digre, line):
            diglist.append((match.start(), digdict[match.group()]))
        vals.append(diglist[0][1] + diglist[-1][1])
    return [int(_) for _ in vals]
```

[6]:
```
data = load_data(Path("data/day01_test2.txt"))
vals = parse_data(data)
sum(vals)
```

[6]:
```
281
```

[7]:
```
data = load_data(Path("data/day01_data.txt"))
vals = parse_data(data)
sum(vals)
```

[7]:
```
54418
```

And this gives us the correct solution.

[ ]:
```

```