day03

December 3, 2023

# Day 3

0.1 Gear Ratios

You and the Elf eventually reach a gondola lift station; he says the gondola lift will take you up to the **water source**, but this is as far as he can bring you. You go inside.

It doesn't take long to find the gondolas, but there seems to be a problem: they're not moving.

"Aaah!"

You turn around to see a slightly-greasy Elf with a wrench and a look of surprise. "Sorry, I wasn't expecting anyone! The gondola lift isn't working right now; it'll still be a while before I can fix it." You offer to help.

The engineer explains that an engine part seems to be missing from the engine, but nobody can figure out which one. If you can **add up all the part numbers** in the engine schematic, it should be easy to work out which part is missing.

The engine schematic (your puzzle input) consists of a visual representation of the engine. There are lots of numbers and symbols you don't really understand, but apparently **any number adjacent to a symbol**, even diagonally, is a "part number" and should be included in your sum. (Periods (`.`) do not count as a symbol.)

Here is an example engine schematic:

```
467..114..
...*......
..35..633.
......#...
617*......
.....+.58.
..592.....
......755.
...$.*....
.664.598..
```

In this schematic, two numbers are not part numbers because they are **not** adjacent to a symbol: `114` (top right) and `58` (middle right). Every other number is adjacent to a symbol and so is a part number; their sum is **4361**.

Of course, the actual engine schematic is much larger. **What is the sum of all of the part numbers in the engine schematic**?

0.2 Part One

Most of my time was spent remembering how to use `numpy` to load string data. That might not have been the best way to do things.

My approach was to identify the locations of all the symbols (`get_symbol_locs`) then iterate through all the elements in the `numpy` array of the data. There's a buffer that stores the number currently being built (`curnum`) and digits are added to this as we find them, iterating through the array. We also check to see if the current position is adjacent to a symbol and, if so, a flag (`symadj`) to indicate symbol-adjacency is tripped. The number is stopped and added to a stack/list as an int - along with the symbol-adjacency flag - when we hit a new row or a symbol.

After this iteration, numbers in the list are filtered to exclude those adjacent to symbols, and the result returned.

[1]:
```
# Python Imports

import re
import string

from collections import defaultdict
from math import prod
from pathlib import Path

import numpy as np
```

0.3 Part One

[2]:
```
def load_data(path):
    with path.open() as ifh:
        # Convert all symbols to "X" for simplicity of location, later
        data = [list(re.sub("[^0-9.]", "X", line.strip())) for line in ifh.
 ↪readlines()]
    return np.array(data, dtype=str)

def get_symbol_locs(arr):
    locarr = np.where(arr == "X")
    return list(zip(*locarr))  # return array as list of location tuples

def cmp_symlocs(pos, symlocs):
    """Return True if current position adjacent to a symbol"""
    for symloc in symlocs:
        if abs(pos[0] - symloc[0]) < 2 and abs(pos[1] - symloc[1]) < 2:
            return True
    return False

def get_numbers(arr, symlocs):
    numbers = []
    itr = np.nditer(arr, flags=['multi_index'])  # array iterator that also␣
↪reports element locations
    curnum = ""  # build the number in this buffer
    symadj = False  # flag for whether current number is adjacent to a symbol
    lastidx = (0, -1)  # previous element location, to check row change
    for val in itr:
        if itr.multi_index[0] != lastidx[0] and len(curnum):  # changed rows
            numbers.append((int(curnum), symadj))  # update number list
            curnum = ""  # reset buffer
            symadj = False  # reset flag
        if str(val) in string.digits:  # is a digit
            curnum += str(val)  # append to buffer
            if symadj == False and cmp_symlocs(itr.multi_index, symlocs):  #␣
↪check adjacency
                symadj = True  # set flag to true
        elif len(curnum):  # not a digit, but buffer populated
            numbers.append((int(curnum), symadj))  # update number list
            curnum = ""  # reset buffer
            symadj = False  # reset flag
        lastidx = itr.multi_index

    numbers = [_[0] for _ in numbers if _[1]]  # filter out False numbers
    return numbers
```

[3]:
```
arr = load_data(Path("data/day03_test.txt"))
symlocs = get_symbol_locs(arr)
numbers = get_numbers(arr, symlocs)
sum(numbers)
```

[3]:
```
4361
```

[4]:
```
arr = load_data(Path("data/day03_data.txt"))
symlocs = get_symbol_locs(arr)
numbers = get_numbers(arr, symlocs)
sum(numbers)
```

[4]:
```
533775
```

0.4 Part Two

The engineer finds the missing part and installs it in the engine! As the engine springs to life, you jump in the closest gondola, finally ready to ascend to the water source.

You don't seem to be going very fast, though. Maybe something is still wrong? Fortunately, the gondola has a phone labeled “help”, so you pick it up and the engineer answers.

Before you can explain the situation, she suggests that you look out the window. There stands the engineer, holding a phone in one hand and waving with the other. You're going so slowly that you haven't even left the station. You exit the gondola.

The missing part wasn't the only issue - one of the gears in the engine is wrong. A **gear** is any * symbol that is adjacent to **exactly two part numbers**. Its **gear ratio** is the result of multiplying those two numbers together.

This time, you need to find the gear ratio of every gear and add them all up so that the engineer can figure out which gear needs to be replaced.

Consider the same engine schematic again:

```
467..114..
...*......
..35..633.
......#...
617*......
.....+.58.
..592.....
......755.
...$.*....
.664.598..
```

In this schematic, there are **two** gears. The first is in the top left; it has part numbers `467` and `35`, so its gear ratio is 16345. The second gear is in the lower right; its gear ratio is 451490. (The * adjacent to `617` is not a gear because it is only adjacent to one part number.) Adding up all of the gear ratios produces **467835**.

What is the sum of all of the gear ratios in your engine schematic?

My solution was adapted directly from that in part 1. The process of identifying numbers to keep is the same, except (i) we only trigger the `symadj` flag if a digit is next to the `*` symbol, and (ii) we set the flag to the gear symbol's location.

After iterating, we retain numbers adjacent to a `*` symbol, and to find pairs of numbers sharing the same gear, we make a dictionary keyed by gear location, with values a list of the adjacent numbers. We filter this dictionary for the gears with exactly two adjacent numbers, and return the pairs as a list of tuples.

[5]:
```python
def load_gears(path):
    with path.open() as ifh:
        # No need to convert symbol locations if only looking for gears
        data = [list(line.strip()) for line in ifh.readlines()]
    return np.array(data, dtype=str)

def get_gear_locs(arr):
    locarr = np.where(arr == "*")  # Only identify gear symbols
    return list(zip(*locarr))

def cmp_gearlocs(pos, symlocs):
    """Return gear location if current position adjacent to a gear"""
    for symloc in symlocs:
        if abs(pos[0] - symloc[0]) < 2 and abs(pos[1] - symloc[1]) < 2:
            return symloc
    return False

def get_numbers(arr, gearlocs):
    numbers = []
    itr = np.nditer(arr, flags=['multi_index'])
    curnum = ""
    symadj = False
    lastidx = (0, -1)
    for val in itr:
        if itr.multi_index[0] != lastidx[0] and len(curnum):  # changed rows
            numbers.append((int(curnum), symadj))
            curnum = ""
            symadj = False
        if str(val) in string.digits:
            curnum += str(val)
            if symadj is False:  # Check for gear adjacency
                gearloc = cmp_gearlocs(itr.multi_index, gearlocs)
                if gearloc:  # if adjacent, set flag to gear location
                    symadj = gearloc
        elif len(curnum):  # not a digit, but buffer populated
            numbers.append((int(curnum), symadj))
            curnum = ""
            symadj = False
        lastidx = itr.multi_index

    numdict = defaultdict(list)  # key by gear location, values are adjacent␣
↪numbers
    [numdict[_[1]].append(_[0]) for _ in numbers if _[1]]  # filter out␣
↪non-adjacent numbers

    # Return only gear numbers where gear is adjacent to two values
    return [tuple(nums) for gear, nums in numdict.items() if len(nums) == 2]
```

[6]:
```
arr = load_gears(Path("data/day03_test.txt"))
gearlocs = get_gear_locs(arr)
numbers = get_numbers(arr, gearlocs)
sum([prod(_) for _ in numbers])
```

[6]:
```
467835
```

[7]:
```
arr = load_gears(Path("data/day03_data.txt"))
gearlocs = get_gear_locs(arr)
numbers = get_numbers(arr, gearlocs)
sum([prod(_) for _ in numbers])
```

[7]:
```
78236071
```

```
[ ]:
```