advent_of_code_day 10

December 15, 2021

1 Advent of Code

[ ]:
```
# set up the environment
import numpy as np
```

1.1 Day 10: Syntax Scoring

You ask the submarine to determine the best route out of the deep-sea cave, but it only replies:

Syntax error in navigation subsystem on line: all of them

All of them?! The damage is worse than you thought. You bring up a copy of the navigation subsystem (your puzzle input).

The navigation subsystem syntax is made of several lines containing **chunks.** There are one or more chunks on each line, and chunks contain zero or more other chunks. Adjacent chunks are not separated by any delimiter; if one chunk stops, the next chunk (if any) can immediately start. Every chunk must **open** and **close** with one of four legal pairs of matching characters:

```
If a chunk opens with (, it must close with ).
If a chunk opens with [, it must close with ].
If a chunk opens with {, it must close with }.
If a chunk opens with <, it must close with >.
```

So, () is a legal chunk that contains no other chunks, as is []. More complex but valid chunks include ([]), {()()()}, <([{}])>, [<>({}){}[([])<>]], and even ((((((((((())))))))))).

Some lines are **incomplete,** but others are **corrupted.** Find and discard the corrupted lines first.

A corrupted line is one where a chunk closes with the wrong character - that is, where the characters it opens and closes with do not form one of the four legal pairs listed above.

Examples of corrupted chunks include `(]`, `{()()()>`, `(((()))}`, and `<([]){()}[{}])`. Such a chunk can appear anywhere within a line, and its presence causes the whole line to be considered corrupted.

For example, consider the following navigation subsystem:

```
[({(<(())[]>[[{[]{<()<>>
[()[<>])]({[<{<<[]>>(
{([(<{}[<>[]}>{[]{[(<()>
((((({<>}<{<{<>}{[]{[]{}
[[<[([]))<([[{}[[()]]]
```

```
[{[{({}]{}}([{[{{{}}([]
{<[[]]>}<{[{[{[]{()[[[]
[<(<(<(<{}))><([]([]()
<{([([[(<>()){}]>(<<{{
<{([{{}}[<[[[<>{}]]]>[]]
```

Some of the lines aren't corrupted, just incomplete; you can ignore these lines for now. The remaining five lines are corrupted:

```
{([(<{}[<>[]}>{[]{[(<()> - Expected ], but found } instead.
[[<[([]))<([[{}[[()]]] - Expected ], but found ) instead.
[{[{({}]{}}([{[{{{}}([] - Expected ), but found ] instead.
[<(<(<(<{}))><([]([]() - Expected >, but found ) instead.
<{([([[(<>()){}]>(<<{{ - Expected ], but found > instead.
```

Stop at the first incorrect closing character on each corrupted line.

Did you know that syntax checkers actually have contests to see who can get the high score for syntax errors in a file? It's true! To calculate the syntax error score for a line, take the **first illegal character** on the line and look it up in the following table:

```
): 3 points.
]: 57 points.
}: 1197 points.
>: 25137 points.
```

In the above example, an illegal) was found twice (2*3 = **6** points), an illegal] was found once (**57** points), an illegal } was found once (**1197** points), and an illegal > was found once (**25137** points). So, the total syntax error score for this file is 6+57+1197+25137 = **26397** points!

Find the first illegal character in each corrupted line of the navigation subsystem. **What is the total syntax error score for those errors?**

1.1.1 Strategy

Should be solvable with a simple stack!

[ ]:
```
problem_data = np.genfromtxt('data/syntax.dat', dtype=np.str0, delimiter='\n')
test_data = np.array(['[({(<(())[]>[[{[]{<()<>>', \
                      '[(()[<>])]({[<{<<[]>>(', \
                      '{([(<{}[<>[]}>{[]{[(<()>', \
                      '(((({<>}<{<{<>}{[]{[]{}' , \
                      '[[<[([]))<([[{}[[()]]]', \
                      '[{[{({}]{}}([{[{{{}}([]', \
                      '{<[[]]>}<{[{[{[]{()[[[]', \
                      '[<(<(<(<{}))><([]([]()', \
                      '<{([([[(<>()){}]>(<<{{', \
                      '<{([{{}}[<[[[<>{}]]]>[]]'])

open_delimiters = ['(', '[', '{', '<']
closing_delimiters = [')',']', '}', '>']
```

[ ]:
```
closing_vals = {")" : 3, "]" : 57, "}" : 1197, ">": 25137 }

line_to_check = ""
bad_closing = []

def check_line(line: str):
    delimiter_stack = []
    for i, c in enumerate(line):
        if c in open_delimiters:
            delimiter_stack.append(c)
        else:
            opening_del = delimiter_stack.pop()
            del_index = open_delimiters.index(opening_del)
            closing_del = closing_delimiters[del_index]
            if c != closing_del:
                bad_closing.append(c)
                return
    return


for line in problem_data:
    check_line(line)

points = 0
for x in bad_closing:
    points += closing_vals[x]

print(f'{bad_closing} give {points} points')
```

```
[']', ')', ']', '>', ')', ']', ']', ')', '>', '}', '>', '>', ']', ']', ')', '}',
'}', ']', ')', '>', ')', ']', ')', ')', ']', ']', '>', '>', '>', ']', '>', '}',
']', '>', '}', '>', ')', ')', '>', '>', ']', ')', '>', ')', ']'] give 358737
points
```

Part Two

Now, discard the corrupted lines. The remaining lines are **incomplete.**

Incomplete lines don't have any incorrect characters - instead, they're missing some closing characters at the end of the line. To repair the navigation subsystem, you just need to figure out the **sequence of closing characters** that complete all open chunks in the line.

You can only use closing characters (),], }, or >), and you must add them in the correct order so that only legal pairs are formed and all chunks end up closed.

In the example above, there are five incomplete lines:

```
[({(<(())[]>[[{[]{<()<>> - Complete by adding }}]])})].
[(()[<>])]({[<{<<[]>>( - Complete by adding )}>]}).
((((({<>}<{<{<>}{[]{[]{} - Complete by adding }}>}>)))).
{<[[]]>}<{[{[{[]{()[[[] - Complete by adding ]]}}]}]}>.
```

`<{([{{}}[<[[[<>{}]]]>[]] - Complete by adding ])}>.`

Did you know that autocomplete tools also have contests? It's true! The score is determined by considering the completion string character-by-character. Start with a total score of 0. Then, for each character, multiply the total score by 5 and then increase the total score by the point value given for the character in the following table:

```
): 1 point.
]: 2 points.
}: 3 points.
>: 4 points.
```

So, the last completion string above -])}> - would be scored as follows:

```
Start with a total score of 0.
Multiply the total score by 5 to get 0, then add the value of ] (2) to get a new total score o
Multiply the total score by 5 to get 10, then add the value of ) (1) to get a new total score
Multiply the total score by 5 to get 55, then add the value of } (3) to get a new total score
Multiply the total score by 5 to get 290, then add the value of > (4) to get a new total score
```

The five lines' completion strings have total scores as follows:

```
}}]])})] - 288957 total points.
)}>]}) - 5566 total points.
}}>}>)))) - 1480781 total points.
]]}}]}]}> - 995444 total points.
])}> - 294 total points.
```

Autocomplete tools are an odd bunch: the winner is found by *sorting **all of the scores and then taking the** middle **score. (There will always be an odd number of scores to consider.) In this example, the middle score is** 288957** because there are the same number of scores smaller and larger than it.

Find the completion string for each incomplete line, score the completion strings, and sort the scores. **What is the middle score?**

1.1.2 Strategy

Should be pretty much like part 1 but this time we need to save any stack that isn't empty when a good result occurs and walk back through the stack completeing the delimiters.

[ ]:
```python
import statistics

closing_vals = {")" : 1, "]" : 2, "}" : 3, ">": 4 }

line_to_check = ""
bad_closing = []

def check_line(line: str):
    delimiter_stack = []
    for i, c in enumerate(line):
        if c in open_delimiters:
```

```
            delimiter_stack.append(c)
        else:
            opening_del = delimiter_stack.pop()
            del_index = open_delimiters.index(opening_del)
            closing_del = closing_delimiters[del_index]
            if c != closing_del:
                bad_closing.append(c)
                return []
    return delimiter_stack

incomplete_closings = []
incomplete_values = []
for line in problem_data:
    remaining_openings = check_line(line)
    if len(remaining_openings) > 0:
        score = 0
        closings = []
        remaining_openings = remaining_openings[::-1]
        for x in remaining_openings:
            closing = (closing_delimiters[open_delimiters.index(x)])
            closings.append(closing)
            score = 5 * score + closing_vals[closing]
        incomplete_closings.append(closings)
        incomplete_values.append(score)

print(f'Middle score is {statistics.median(incomplete_values)}')
```

```
Middle score is 4329504793
```