day3

December 6, 2020

0.0.1 Advent of Code Day 3

We have to take a string that represents a tobogan run, trees and then plot a route down the tobogan run.

Key things to note today: 1. The run extends horizontally infinitely 2. Each tree will repeat as the run repeats 3. We need to parse a kind of map to work out where the trees are

So, if we start with the assumption that we'll parse the array, and caclulate something that looks like: `{ 0: [2, 3] 1: [5, 9] }` as the tree data, indicating that in row 0, there are trees at x=2, and x=3, then in row 1 there are trees at x=5, and x=9.

Given the additional data of how often to repeat (the stride of the run), and you'll have a tree at x=2, x=2+repeat, x=2+repeat+repeat etc… It's helpful to realise that you don't need to count repeats, that's the same as `x % stride`, or x mod stride. Remember, mobular arithmatic is like a clock, each time you reach the mod limit, you reset to 0, so `2 % 10 == 12 % 10 == 22 % 10 == 282 % 10`

Now we don't need to calculate all of these, we just need a function that can say "For point `x,y`, is there a tree"…

So lets start there

[1]:
```
import pytest
import ipytest
ipytest.autoconfig()
```

Let's define our lookup function assuming test data like that

[2]:
```
def is_tree_at(trees, x,y):
    return (x % trees["stride"]) in trees["rows"][y]
```

[3]:
```
testdata = {
    "rows":{
        0: [2, 3],
        1: [5, 9]
    },
    "stride": 10,
    "height":3
}
```

```
assert is_tree_at(testdata, 2,0)
assert is_tree_at(testdata, 3,0)
assert not is_tree_at(testdata, 4,0)
assert is_tree_at(testdata, 12,0)
assert is_tree_at(testdata, 13,0)
assert is_tree_at(testdata, 22,0)

assert is_tree_at(testdata, 5,1)
assert not is_tree_at(testdata, 7,1)
assert is_tree_at(testdata, 15,1)
```

Great, so we can turn our data structure into a simple function, now we need to draw a path. Note that we start at top left, which I've called 0,0 (X=0, Y=0).

there's two ways we can do this, if we know that we are going to follow a simple rational slope of right 3, down 1 (which is what my puzzle input says), then we can define that as `dx=3, dy=1`, and simply iterate doing `x=x+dx, y=y+dy` until `dy > height`. That's probbaly the easiest. I have a strong suspicion that part 2 will either require us to count trees inbetween, or will need us to test a range of slopes, so lets make a function that counts the trees for a given slope in dx,dy and see what happens

[4]:
```
def count_trees(startx,starty, dx,dy, trees):
    x,y = startx,starty
    count = 0
    while y < trees["height"]:
        if is_tree_at(trees, x,y):
            count += 1
        x,y = x+dx,y+dy
    return count
```

[5]:
```
testdata = {
    "rows":{
        0: [2, 3],
        1: [3, 9],
        2: [6, 9]
    },
    "stride": 10,
    "height": 3
}

assert count_trees(0,0,3,1,testdata) == 2
```

OK, so that worked first time, so lets try with the test data that the Advent of Code gave me. This requires parsing the lines. Given how I want the data, this isn't too hard, ignore the `.`'s and just count the `#`'s

Handy tip, defaultdict is a dictionary that if you ask for a key that doesn't exist, will create a new item instead of erroring. So if your dictionary is empty calling d["foo"] will return you the new default object. If those are empty lists, you can just call append on it, and modify the list in place,

and the code is the same regardless of whether you've seen an element before. No more "if has X, new list, else append to list" gubbins

[6]:
```
from collections import defaultdict

def parse_trees(lines):
    d = {}
    d["rows"] = defaultdict(list)
    d["stride"] = len(lines[0])
    d["height"] = len(lines)
    for y,line in enumerate(lines):
        for x,c in enumerate(line):
            if c == "#":
                d["rows"][y].append(x)
    return d

testdata = [
    "..##.......",
"#...#...#..",
".#....#..#.",
"..#.#...#.#",
".#...##..#.",
"..#.##.....",
".#.#.#....#",
".#........#",
"#.##...#...",
"#...##....#",
".#..#...#.#"
]

treedata = parse_trees(testdata)
assert 7 == count_trees(0,0,3,1,treedata)
```

[7]:
```
part1_treedata = parse_trees(open("day3.txt").readlines())
count_trees(0,0,3,1,part1_treedata)
```

[7]:
```
60
```

That's not working for some reason, so lets do it the other way. I'm expecting this to fail for part 2, but lets go try it from the original text instead. Lots more hard coding, but we'll compare it

[8]:
```
def is_a_tree(line, x):
    return line[x%len(line)]=='#'

def count(lines):
    x = 0
    count = 0
    for row in lines:
```

```
        if is_a_tree(row, x):
            count += 1
        x += 3
    return count

print(count(testdata))
lines = [line.strip() for line in open("day3.txt").readlines()]
print(count(lines))
```

```
7
184
```

So, turns out the issue is because I am an idiot. open().readlines() returns a set of lines, including the trailing newline.

Let's try again with the correct input

[9]:
```
part1_treedata = parse_trees([line.strip() for line in open("day3.txt").
↪readlines()])
print(count_trees(0,0,3,1,part1_treedata))
```

```
184
```

Outstanding, my code works, I just had the wrong input!

0.0.2 Part 2

Now, as expected, we have to check multiple slopes and then multiply them together. Easy with the code we wrote from the beginning, so we can solve that really quickly!

[10]:
```
r1d1 = count_trees(0,0,1,1,part1_treedata)
r3d1 = count_trees(0,0,3,1,part1_treedata)
r5d1 = count_trees(0,0,5,1,part1_treedata)
r7d1 = count_trees(0,0,7,1,part1_treedata)
r1d2 = count_trees(0,0,1,2,part1_treedata)

print(r1d1,r3d1,r5d1,r7d1,r1d2)
print(r1d1*r3d1*r5d1*r7d1*r1d2)
```

```
62 184 80 74 36
2431272960
```