Two Ways To Solve a Linear Equation in Python

Abigail Haddad

3/8/23

I got nerd-sniped recently by a YouTube ad.

The description is **"Math Olympiad Question | Nice Algebra Equation | You should know this trick!!"**

It shows you how to solve for x+y when x + xy + y =54 and x and y are positive integers.

The video solves it via factoring. I solved it in Python.

Here are two ways to do it.

The first is the brute-force way, where we loop through every possible number that x and y could be, and then we return the sets where our equation is true. This is the relatively "computationally-intensive" way of solving the problem, because we're trying out every possible combination of numbers, which is a lot of computations. However, it doesn't take too much time here because there's only one equation and we can already limit the numbers we try.

We'll use the range from 1 to 54. (Below it says 1, 55 - but that's because the last number, the 55, won't be included.)

The code for that is this below, and the numbers below it are the output:

```
for x in range(1, 55):
    for y in range(1, 55):
        if x+x*y+y==54:
            print(x, y, x+y)
```

```
4 10 14
10 4 14
```

What the output is showing is that there are two solutions for x and y, but for both of them, x+y= 14.

And the 'two solutions' are really just x=4, y=10 or x=10, y=4.

You can also be a little neater if you want to write your code using what's called list comprehension, like this:

```
print([[x+y, x, y] for x in range(1,55) for y in range(1,55) if (x+x*y+y==54)])
```

```
[[14, 4, 10], [14, 10, 4]]
```

But it's doing the same thing.

A second way to do this is by using a library for symbolic mathematics called SymPy, which will solve the equation for us symbolically – that is, not through brute force, but via math.

Here is the code for that, followed by the results:

```
from sympy.solvers.diophantine.diophantine import diop_solve
from sympy import symbols

x, y = symbols("x,y", integer=True)
solutions=list(diop_solve(x + y + x* y - 54))
positive_solutions=[i for i in solutions if (i[0]>0) & (i[1]>0)]
print(positive_solutions)
```

```
[(4, 10), (10, 4)]
```

You can see I'm importing functionality from the SymPy library (which I have installed). I'm telling SymPy when I create the x and y symbols that I only want integer, or whole number, solutions. (That's also what my loop above does – we're only iterating through whole numbers.)

I then run diop_solve to solve the equations. It returns to me some solutions which include negative numbers, so I filter those out in the subsequent line which begins "positive_solutions".

That line is also an example of list comprehension—I could do this in a for loop instead if I wanted.

I then print out the solutions and they're the same as above: 4 and 10, or 10 and 4.

If we wanted to abstract this out to a format that solves all equations of the form x+xy+y= some constant (which is still not very abstracted out!), we could write functions like this:

```
def loopThroughSolvex_xy_y(constant):
    results=[[x, y] for x in range(1,constant) for y in range(1,constant) if (x+x*y+y==con
    return(results)

results=loopThroughSolvex_xy_y(47)
print(results)
```

```
[[1, 23], [2, 15], [3, 11], [5, 7], [7, 5], [11, 3], [15, 2], [23, 1]]
```

But this time, the constant that I used was 47 instead, and so I got a whole different list of numbers.

For x+xy+y=47, x+y can be 24, 17, 14, or 12 –we had this function return the underlying pairs of possible numbers.

Or alternatively, if we want to do it in SymPy, we can define that as a function instead:

```
def symbolSolvex_xy_y(constant):
    x, y = symbols("x,y", integer=True)
    solutions=list(diop_solve(x + y + x* y - constant))
    positive_solutions=[i for i in solutions if (i[0]>0) & (i[1]>0)]
    return(positive_solutions)

results=symbolSolvex_xy_y(47)
print(results)
```

```
[(23, 1), (5, 7), (1, 23), (15, 2), (3, 11), (11, 3), (2, 15), (7, 5)]
```

And then when we called symbolSolvex_xy_y(47), we get the same set of answers as we did using the brute force/list comprehension function.