07-for-loops

February 4, 2020

0.1 For loops

In several problems we solved in class, there was the need to repeat the same steps for a number of times. Instead of writing the same steps over and over again, we used the "repeat" word. One of the ways of translating this repetition to python is using `for` loops.

The syntax for `for` loops is:

```
for i in <sequence of things>:
    <steps>
```

Let's break this down.

0.1.1 Sequence of things

The `<sequence of things>` could be a sequence of integers, floats, booleans, etc. We start by looking at sequences of integers. We will do this by using the python command `range`. There are three ways to use that command:

- `range(stop)` will generate the sequence of numbers from 0 to **stop-1**. So `range(6)` will generate the sequence 0,1,2,3,4,5.
- `range(start,stop)` will generate the sequence of numbers from `start` to **`stop-1`**. So `range(4,10)` will generate the sequence 4,5,6,7,8,9.
- `range(start,stop,step)` will generate the sequence of numbers beginning at `start`, with increments of `step`, until reaching `stop-1`. So `range(4,10,2)` will generate the sequence 4,6,8.
- Note that `range(start,stop)` is the same as `range(start,stop,1)`.
- Note that steps may be negative.

The number of elements in your sequence determines how many times the loop repeats itself, or, more technically, how many times it **iterates**. Each loop repetition is called an **iteration**.

If you are not sure which sequence of numbers a range would generate, you can check it by typing the range command you want to use, wrapped around with a `list` function. We will see more about lists later in the course.

```
In [1]: list(range(-4,10))
```

```
Out[1]: [-4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [2]: list(range(10,3,-1))

Out[2]: [10, 9, 8, 7, 6, 5, 4]
```

0.1.2 Steps

The steps that will be repeated in a loop need to be indented to the right, since this is inside the scope of the loop only. If you need to write more code after the loop is done, simply indent it back to align with the `for` keyword.

Most of the times, you will want to repeat steps for different values of a variable. That is why the *loop variable* `i` is available for you to use in your `<steps>`. Each time the loop is repeated, `i` takes the next value in the sequence of things you determined.

For example, take the loop:

```
for i in range(5):
    <steps>
```

The steps will be repeated 5 times (5 iterations), since the sequence is 0,1,2,3,4. * On the first iteration, `i` has the value 0. * On the second iteration, `i` has the value 1. * On the third iteration, `i` has the value 2. * On the fourth iteration, `i` has the value 3. * On the fifth iteration, `i` has the value 4.

So, if we write something (useless) like this:

```
In [3]: x = 0
        for i in range(100):
            x = i
```

Once the loop is over, what will be the value of `x`?

0.1.3 Loop variable

Every loop has what we call a **loop variable**. In the examples above, we have used `i` as this variable's name, but it could really be anything you want. For example:

```
In [4]: for even in range(0,100,2):
            ...
```

Careful: Do not modify variables that are used in the loop command. This can only make things more complicated.

0.1.4 Exercise 1

Implement the funcion `sumOdds(m, n)` that returns the sum of all odd numbers between m and n, not inclusive. So `sumOdds(-5,6)` should return 5 (-3-1+1+3+5).

```
In [5]: def sumOdds(m,n):
            sum = 0
            for i in range(m+1,n):
                if i % 2 != 0:
                    sum = sum + i
            return sum

        sumOdds(1,3)

        # Another option
        def sumOdds(m,n):
            sum = 0

            if m % 2 == 0:
                m = m + 1
            else:
                m = m + 2

            for odd in range(m,n,2):
                sum = sum + odd
            return sum
```

Follow up to exercise 1 Draw the iteration table for the code above.

0.1.5 Exercise 2

The *alternating sum* of a sequence $(a_1, a_2, ...a_n)$ is defined as: $a_1 - a_2 + a_3 - ... \pm a_n$.

Implement the function `alternatingSum(n)` that returns the alternating sum of all numbers between 1 and n, inclusive.

```
In [6]: def alternatingSum(n):
            return 42
```

Follow up to exercise 2: Try out this function for a few different numbers. Can you come up with a closed formula (one that does not use loops)?

```
In [7]: def alternatingSum(n):
            return 42
```

0.1.6 Exercise 3

In mathematics, a perfect number is an integer for which the sum of all its own positive divisors (excluding itself) is equal to the number itself. For example the number 6 is perfect, because 1+2+3 is equal to 6.

Implement the function `isPerfect(n)` that returns `True` if `n` is a perfect number, or `False` otherwise.

```
In [8]: def isPerfect(n):
            return True

        # Perfect numbers to test with: 6, 28, 496
```

0.1.7 Exercise 4

Given some positive integer `n`, we can write `n*2` lines following a certain pattern. For example, if `n` is 5, the 10 lines are:

```
1 1 1
1 2 2
2 4 8
2 5 9
3 9 27
3 10 28
4 16 64
4 17 65
5 25 125
5 26 126
```

Find out what is the pattern for generating the lines, and implement a function `patternSum(n)` that takes a positive integer `n` and returns the sum of all numbers in the `n*2` lines generated using the pattern above.

```
In [10]: def patternSum(n):
             return 42
```