Solutions and explanations for these problems can be found at the end of this document.

Problems:

1. Write the function **length(s)** that takes in a string s and returns the number of characters in the string. You may not use the built-in **len** function.
2. Write the function **num_digits(n)** that takes in an integer n and returns the number of digits. You may not convert **n** to a string.
3. Write the function **sum_every_other(L)** that takes in a list of integers L and returns the sum of every other element in the list, starting from the first element reading left to right. For example, if we had `L = [1, 2, 3, 4, 5, 6]`, we would want to add up 1 + 3 + 5 = 9.
4. Write the function **is_fooey(n)** that takes in a positive integer n and returns True if it is “fooey” and False otherwise. We define a number as “fooey” if it is exactly one away from a multiple of 10. For example, 29, 41, 1, and 99 are all fooey numbers. (Yes, fooey is a made-up word.)
5. Write the function **nth_fooey(n)** that returns the nth fooey number. You will want to use the **is_fooey** function you wrote previously.
6. Write the function **nth_super_odd(n)** that returns the nth “super-odd” number. We define a super-odd number as one where all the digits are odd. You may not use strings in this problem.

<u>Solutions:</u>

length: For this problem, we need to loop through the string and keep a running count of the total number of characters seen. Luckily, we won't need to do any additional analysis on the characters themselves or anything. As mentioned in the notes, we have two possible ways of looping through a string. We can either do:

```
for i in range(len(s))        or          for c in s
```

The first way loops through the **indexes** of the entire string and the second way loops directly through the characters. Also as mentioned in the notes, using variables **i** and **c** is general convention (because **i** stands for index and **c** stands for character, we're very creative). You are welcome to use different variables if you think it improves clarity.

It's natural to wonder when you should use which method. As a general rule of thumb, you should loop through indexes if you care about where a character appears in the string. In other words, if somewhere in your loop, you will do different things depending on the index, then loop through the indexes. On the other hand, if you don't care about the placement of the character, then feel free to use the second way.

As a more general rule: when in doubt, use the first way. This is because looping through the indexes will always give you access to the index and character just in case you need both. But if you loop directly through the characters (in the second way), there's no way to get the index. In all honesty, it's usually not a huge hassle to switch between the two either way so don't worry too much about choosing the "wrong" way to loop.

Okay, back to the actual problem. Since we're just tallying up the number of characters in the string, we don't need to keep track of indexes, so we'll loop directly through the characters:

```
def length(s):
    num_chars = 0
    for c in s:
        num_chars += 1
    return num_chars
```

num_digits: This problem is a little trickier because it involves looping through a number. Under normal circumstances (i.e. in industry), the easiest way would be to convert the input into a string and get the length of that string. Unfortunately, for learning purposes, we are going to do this the much more tedious, but much more informative way :)

The core idea behind this is similar to the algorithm from the previous problem: we need to loop through the integer and keep a tally of how many digits we've seen. The issue is: we don't have a nice way to loop through an integer like we do with strings because an integer isn't a sequence type like strings and lists.

Let's think back to what operations we can do on integers, which are basically just all the mathematical operations. Luckily, two of these operations are quite useful: % and //. Recall that we learned last week that n%10 gives us the last digit of a number and n//10 chops off the last digit of a number. For this problem, we technically don't even need n%10 since we don't particularly care what a digit is, we just care that there is a digit. So, we just need to keep chopping off the last digit of a number until we have no more digits to chop off.

So what loop should we use for this? We actually can't use a for loop here because we don't know what to loop through. Unlike with strings, we can't just loop through the digits of an integer. So, a while loop is our only choice. What should the loop guard be? Well, let's look at a small example: suppose we had the number 12. After the first loop, we cut off the 2 and are left with 1. After the second loop, we cut off the 1 and are left with 0. (Any one digit number // 10 = 0). Once we get to 0, integer dividing by 10 will just keep giving us 0. So, our loop can be **while n > 0**.

Our entire function will look like this:

```
def num_digits(n):
    num = 0
    while n > 0:
        num += 1
        n //= 10
    return num
```

This is almost completely correct - there are just two edge cases we have to account for. The first is if **n** is negative. Then our while loop will never trigger and we'll always return 0 as the answer, which is incorrect. To fix this, we can just take the absolute value of **n** at the very beginning.

The second edge case is if the input is 0 itself. In this case, the while loop will again not trigger and the function will tell us that we have zero digits, but the number 0 has one digit. To account for this, we can just add a conditional for this specific input at the beginning. This might feel a bit like cheating but in many situations, we have one annoying input that doesn't act like all the other ones so we do have to account for it specifically.

The new and improved version of this function looks like this:

```
def num_digits(n):
    num = 0
    n = abs(n)
    if n == 0:
        return 1
    while n > 0:
        num += 1
        n //= 10
    return num
```

Note that our **if** conditional doesn't have an else because we have a return statement inside it. So, if we enter the conditional, we'll exit the function immediately. If we don't, we'll just skip it and carry on with the rest of the function.

sum_every_other: The key idea of this algorithm is that we only add a number to our total if its index is even. Since we always start with the first element of the list, that means we want to add the elements at indexes 0, 2, 4, etc. So, when looping through the list, we need to loop through the indexes since those are pretty important.

```
def sum_every_other(L):
    total = 0
    for i in range(len(L)):
        if i % 2 == 0:
             total += L[i]
    return total
```

A couple syntactic things to point out: we are indexing into the list using L[i] – i.e. accessing the element at the ith index. Additionally, we could have gotten around using a conditional but adding a **step** argument of 2 to the range in our for loop so that we jumped through only the even indexes.

```
def sum_every_other(L):
    total = 0
    for i in range(0, len(L), 2):
        total += L[i]
    return total
```

Note that in this case, we also have to explicitly specify that we are starting at index 0 in order for Python to interpret the arguments correctly.

is_fooey: We want this function to return True if the input is one away from a multiple of 10. So, we can just check if n-1 or n+1 is a multiple of 10 – i.e. if we get a remainder of 0 when we mod it by 10.

```
def is_fooey(n):
    if ((n-1) % 10 == 0) or ((n+1) % 10 == 0):
        return True
    return False
```

Note that I combined both if statements into one using the **or** logical operator, but it's totally fine to keep them separate as well!

nth_fooey: As discussed in the notes, the function for finding the "nth something" number is pretty standard – the only real change is the function that checks the condition. So, we'll pretty much just regurgitate that template here and use the **is_fooey** function from above:

```
def nth_fooey(n):
    num_found = 0
    guess = 0
    while (num_found < n):
        guess += 1
        if (is_fooey(guess)):
            num_found += 1
    return guess
```

If you have questions about this template, feel free to reread the notes (it's under the While Loops section) or ask me/Austin!

nth_super_odd: Again, we can recycle the nth template from the previous problem – we just need to write a function **is_super_odd** to check if a number is super-odd or not. Similar to when we were counting the number of digits, we are going to loop through the digits of the number and check if each one is odd (so this time, we will need to make use of our n%10 operation). If the digit is not odd, then we can return False immediately because we know at that point that our number is not super-odd and so there's no point in continuing to loop.

```
def is_super_odd(n):
    while n > 0:
        last_digit = n % 10
        if last_digit % 2 == 0:
            return False
    return True
```

Note the placement of our **return True** statement. We need to put it after the while loop is finished because this indicates that we've looped through all the digits and never found an even digit, so we can conclude that all the digits were odd. A frequent mistake I see is this:

```
def is_super_odd(n):
    while n > 0:
        last_digit = n % 10
        if last_digit % 2 == 0:
            return False
        else:
            return True
```

This doesn't work because in our first iteration, we are going to enter either the **if** or **else** statement. Since both of them have return statements, we will exit the function after just the first loop. This incorrect version of the function is basically the same as checking if just the last digit of **n** was even or odd instead of *all* the digits.

So, with our **is_super_odd** function written, we can go ahead and finish it up with **nth_super_odd**

```
def nth_super_odd(n):
    num_found = 0
    guess = 0
    while (num_found < n):
        guess += 1
        if (is_super_odd(guess)):
            num_found += 1
    return guess
```