Loops:

Loops are one of the basic building blocks of coding. They allow us to perform an action repeatedly without the need to copy and paste the same line of code over and over. There are two types of loops in Python: while loops and for loops. They are each covered in more detail below.

While Loops:

While loops are used when you can't assign a specific number to how many times you need to loop. For example, suppose we wanted to write a program that continuously stores characters the user is typing until the user presses Enter. Although we can say that we want to stop looping "when the user presses Enter", this is a vague quantification and we can't actually assign a specific value to when that will happen. I.e. we can't say we will stop looping after 10 or 20 or 50 characters because the user could easily enter more than that.

The basic structure of a while loop is

```
while (some condition is True):
    do some action
```

As with conditionals, everything that is indented inside the while loop is part of the loop body. The loop will continue being executed while the specified condition, which we call the loop guard, remains True. When it becomes False, the loop exits and we continue on with the rest of the code.

A common pitfall when using while loops is the **infinite loop**. This occurs when the loop guard never becomes False, so we will keep iterating forever. If you have an infinite loop, it may look like your console has frozen when you run the code. It is good practice to always do a sanity check when writing while loops: identify what variable(s) the loop guard depends on and ensure that it is being changed in the loop body.

Example: suppose we want to write a program that asks the user to enter a positive number. Since users can be unpredictable, it is possible that they will enter a negative number (we'll pretend that they won't enter a string). We need our program to keep asking for a positive integer until we actually get a positive integer. Similar to the example at the top of this section, we cannot quantify how many tries it will take to get a positive integer input so we use a while loop.

```
def get_positive_int():
    user_input = 0
    while (user_input <= 0):
        user_input = int(input("Enter a positive integer: "))
    return user_input
```

As long as the user enters a negative integer or zero, the loop guard will be True so we keep asking for a positive integer. Once we get that, **user_input** becomes > 0, the loop guard evaluates to False, and we exit the loop and return **user_input**.

Sometimes, the loop guard condition uses a variable that is a boolean. In that case, the while loop may look like

```
while variable:
    do some action
```

This may look confusing, but it's equivalent to saying

```
while variable == True:
    do some action
```

To give a more concrete example, consider the function **get_positive_int** written a different way:

```
def get_positive_int():
    user_input_is_negative = True
    while (user_input_is_negative):
        user_input = int(input("Enter a positive integer: "))
        if user_input > 0:
            user_input_is_negative = False
    return user_input
```

In this version, the loop guard we are using is a boolean that is toggled within the loop body once the user enters a positive integer. As mentioned above, you can also write it as

```
while (user_input_is_negative == True)
```

if that makes more sense to you!

There are a couple common bugs that occur when using while loops. The first are infinite loops as discussed previously. The second is what we unofficially call the **off-by-one** error. This is a type of logical error so it's a bit harder to catch since it won't crash the code. This error occurs when there's a mistake in the loop guard that causes the loop to execute either one too many or one too few times. If you suspect that your loop is not performing correctly, you should manually check the first and last iteration of the loop to ensure that all the computations in the loop body are being performed correctly.

Let's look at an example. Suppose we have written this function called **sum_to_n** that takes in an integer n and returns the sum of all the integers from 1 to n.

```
def sum_to_n(n):
    counter = 0
    total = 0
    while (counter <= n):
        counter += 1
        total += counter
    return total
```

If we run this function with a small input, say **sum_to_n(4)**, we should expect to see 10 as the output (1 + 2 + 3 + 4). However, we actually get 14, which is incorrect. To debug, let's try to see if it's due to an off-by-one error in our loop.

First, we'll check the first iteration of the loop. At this point, we have **counter** = **0** and **total** = **0**. The loop guard is True, so we will enter the loop body and update **counter** = **1** and **total** = **1**. That seems correct.

Next, we'll check the last iteration of the loop. This is when **counter** = **n** because this is the maximum value of **counter** before it becomes greater than **n** and causes us to exit the loop. When we enter the loop body, we update **counter** = **n+1** and then add **n+1** to **total**. But this seems incorrect – we want the sum from 1 to n, so at no point should we be adding **n+1** to **total**! This is where the bug lies, and we can fix it by changing the loop guard to be **counter** < **n** rather than **counter** <= **n**.

We can do another quick sanity check after this change to make sure it was the correct change. The last iteration of the loop now occurs when **counter** = **n-1**. When we enter the loop body, we will update **counter** = **n** and then add **n** to **total**. This is correct! If we run the function with the input 4 again, we'll see that we now get the correct answer of 10.

A common application (in this course at least) of while loops is finding the "nth something number". By this, we mean finding the nth number that satisfies some property. For example, the nth even number, the nth prime number, or the nth narcissistic number (yes, this is actually a thing). The algorithm for this type of problem can be formulated by considering how we would solve such a problem ourselves. Let's use a semi-concrete example.

Suppose someone asked us for the 37th prime number. I'm reasonably sure that no one knows this off the top of their head, so we'll have to employ a more methodical approach. Most people will probably settle on an approach like this:

Start with 1. This is not a prime so we move on to 2, which is prime so that's one prime number found. 3 is the next prime number, so now we've found two prime numbers. 4 is not a prime so we'll ignore it. 5 is a prime – that's three prime numbers found. 6 is not a prime, but 7 is so now we're up to four prime numbers. 8, 9, and 10 are not prime. 11 makes five primes found. Continue this process until we've found our 37th prime number.

To generalize this, we have two counters: one keeps track of how many prime numbers we've found so far and one keeps track of the number that we're checking. In the example above, the first counter is highlighted blue and the second counter is highlighted red to help visualize this process a little better.

To code this up, we can see how a while loop works so well for our purposes. Naturally, we want to loop until we've found the 37th prime number, but we have no idea when that will be. Here's the code for this problem:

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

The variables **num_found** and **guess** are highlighted blue and red accordingly to link them back to what we said they're keeping track of. We initialize **num_found** as 0 since we haven't found any prime numbers at the beginning. We initialize **guess** as 0 because we know that all prime numbers are positive, so this way we can be sure not to miss any when we start counting.

The loop guard continues while we haven't found **n** prime numbers. In each iteration, we increment our guess and check if our new guess is prime. If it is, we increment the number of primes we've found (assume that **is_prime** function is correctly written). If our guess is not prime, we do nothing and just continue to the next iteration.

The structure for these "nth something number" problems very closely resembles the **nth_prime** function above. The only real difference is the conditional check. Instead of checking if the guess is prime, you would want to write a different helper function that checks if the guess satisfies whatever condition we want – i.e. write an **is_something** function.

For Loops:

For loops are used when you can assign a specific value to how many times you want to loop. A very common use case is looping through strings – typically, you want to loop through the entire string and you know exactly how many characters are in that string. Another common case is if you want to perform an action some set number of times. For example, suppose you wanted to print out all the numbers from 0 to 100. The built-in **range** function is very useful in these cases – let's review how it works:

The **range** function can take in one, two, or three parameters.

- One parameter: **range(n)** produces a sequence of all integers from 0 to **n**, excluding n itself. For example, **range(4)** produces the integers 0, 1, 2, 3.
- Two parameters: **range(m, n)** produces a sequence of all integers from **m** to **n**, excluding n itself. For example, **range(2, 6)** produces the integers 2, 3, 4, 5.
- Three parameters: **range(m, n, s)** produces a sequence of all integers from **m** to **n**, excluding n itself, stepping by **s**. For example, **range(1, 6, 2)** produces integers 1, 3, 5.

As a quick note, **range** technically outputs a special range type. To get the entire sequence of integers from the range, it is common to use **list(range())** to convert the range type to a more familiar list type. Luckily, when using **range** in a for loop, we don't need to do this conversion as the loop can iterate through a range type just fine.

The general structure of a for loop is as follows:

```
for i in range(n):
    do some action
```

Note that **i** is a variable that is being created implicitly in the for loop and initialized as the first value in the loop. It is convention to use **i** when looping through a range, but you can technically use whatever variable name you want. Sometimes, a more descriptive variable name is helpful in terms of code clarity.

Let's look at an example. Let's go back to the function **sum_to_n** that we covered in the previous section, but we will write it with a for loop instead.

```
def sum_to_n(n):
    total = 0
    for i in range(n+1):
        total += i
    return total
```

Note that we loop through range(**n+1**) because we want the sum from 1 to n but the argument to range is exclusive. If we had looped through range(**n**), we would get the sum from 1 to n-1.

As you can see, it's possible to always convert a for loop into a while loop (but not vice versa). However, if a for loop can be used, it is usually a cleaner, more concise solution than using a while loop.

Let's look at another example: writing the **is_prime** function. To first motivate the algorithm behind this code, we will again consider how we would solve this problem ourselves if someone gave us a large number n and asked us if it were prime. Most people would probably take the following approach:

If n is less than 2, it's not prime. Check if n is divisible by 2. If it is, it's not prime. Check if n is divisible by 3. If it is, it's not prime. Continue this process until 1) we find that n is divisible by some factor in which case, we know it's not prime or 2) we get all the way up to n-1 and have found no divisors, in which case we know it is prime.

A for loop suits our purposes perfectly because we know that we need to loop up until n-1 at most. In code, this looks like:

```
def is_prime(n):
    if (num <= 1):
        return False
    for possible_factor in range(2, n):
        if (n % possible_factor == 0):
            return False
    return True
```

To recap, we are looping through all possible factors from 2 to n-1. (Notice that we are using a variable **possible_factor** instead of **i** because it helps clarify what we are using the variable for.) In each iteration, we check if n is divisible by this possible factor. If it is, we know n cannot be prime and so we can return False immediately, thus ending the function there. If we finish looping through all the possibilities, we know n has no factors (other than 1 and itself), so we can conclude that n is prime and return True.

Note that it is very important that this **return True** line is outside of the for loop. A common mistake is to put it inside the for loop like this:

```
def is_prime(n):
    if (num <= 1):
        return False
    for possible_factor in range(2, n):
        if (n % possible_factor == 0):
            return False
        else:
            return True
```

In this case, since the function has to enter either the if or else block, it will encounter a return statement in both and may end prematurely. In this scenario, if n is not divisible by the factor (the else case), we want to continue with the loop, not return True immediately because we still have more factors to check.

So far, we've mainly focused on writing for loops that loop through some range. It is also common to loop directly through the elements of a sequence as such:

```
for variable in sequence:
    do some action
```

The variable here depends on the type we are looping through, there is really no common conventional naming system. In general, the variable should just be descriptive of the value it contains.

We will cover more examples of looping through strings and lists when we cover these data types in more depth in Module 5.

Indexing:

Note: this is a very brief overview of indexing. We will discuss it in far more detail in Module 5 when you get more in depth with strings/lists.

Each character in a sequence has a corresponding location called an index. As is convention in computer science, indexing starts at 0, so the first character of a sequence (from left to right) occurs at index 0. Here's a pictorial representation of the string **s** = **"Hello"** with the corresponding indexes of each character.

Index	0	1	2	3	4
Character	H	e	l	l	o

We can access individual characters using square brackets. For example, `s[0]` gives us "H", `s[1]` gives us "e", and `s[4]` gives us "o". This is called indexing into a string. Notice that the index of the last character is `len(s)-1`, where `len(s)` is a built-in function that gives you the total number of characters in the string.

If you attempt to access an index that is out of bounds (e.g. `s[7]`), you will get an "IndexError: string index out of range" message.