Fundamentals of iteration

Many programming tasks require you to repeat a process over and over again, until something happens. For instance, it could be counting until you reach some number. It could be repeatedly refining your estimate of a quantity until it's accurate enough. It could even be walking until you reach the end of the street. This is called iteration.

Syntax for iteration

To approach iterative problems in Python, you'll need a `while` loop. You have to specify two things:

- Whether to keep going. This comes right after the `while` keyword.
- What to do, if we decide to keep going. This can get really complicated, so it would be unwise to try and cram it all onto the same line that has the `while` keyword and the condition. Instead we go to a new line and indent a little bit. There we write all the code that should happen if we decide to keep going.

Let's see an example:

```
>>> i = 0
>>> while i < 3:
...     print(i)
...     i += 1
...
0
1
2
```

This `while` loop's condition is `i < 3`. Here's how it works:

1. When we start out, `i` equals 0. Since 0 is less than 3, the condition is `True`. We decide to enter the `while` loop.
2. We `print(i)`, so the value 0 gets displayed.
3. We increment `i`, so its new value is 1. This ends the first loop.
4. Since 1 is less than 3, the condition is still `True`. We decide to enter the `while` loop again.
5. We `print(i)`, so the value 1 gets displayed.

loop again.
8. We `print(i)`, so the value 2 gets displayed.
9. We increment `i`, so its new value is 3. This ends the third loop.
10. Since 3 is not less than 3, the condition is now `False`. We have finished the `while` loop.

Notice that in order for this `while` loop to start, the condition had to be `True`. Similarly, in order for it to stop, the condition had to be `False`. **To make it eventually change from `True` to `False`, we had to update a variable that the condition relied on, when we were inside the `while` loop.** If you don't do that, then the condition will never become `False` and you could get stuck in the `while` loop forever.

Practice: the factorial of a number

In math contexts, $n!$ denotes the factorial of a number n. It is the product of n with every positive number below it. That is, $n! = n \times (n-1) \times (n-2) \times \cdots \times 1$. For instance $4! = 4 \times 3 \times 2 \times 1 = 24$. Define the function `factorial(n)`, which returns the factorial of `n`.

```
def factorial(n):
    ...
```

`while` loops let us repeat something many times. We should think, what do we need to repeat in order to calculate the factorial of a number? Referring to the definition of $n!$ above, it seems like we need to repeatedly perform multiplication. How many multiplications? n multiplications. So one way to solve this problem is to make the `while` loop happen `n` times, and do one multiplication each time we go through the loop.

In order to make the loop happen `n` times, we'll need a condition that's `True` for the first `n` iterations and then `False` afterwards. So how about we use `n` like a countdown, decrementing by 1 each time through the loop? Then when it becomes 0, we'll stop. Our condition should be something like `n > 0`, since that's `True` for the first `n` iterations and then becomes `False` at the end of our countdown.

```
def factorial(n):
    while n > 0:
        ...
        n -= 1
```

value that was provided as the argument. The second time through the loop, it's one less than that. And on the last time through the loop, `n` is 1.) It's a pretty common technique to use one of your variables like a countdown when you're doing iterative programming, so make sure you understand how we got to the countdown in the code above.

Now it's time to consider what we want to do in the `while` loop. Since we go through it `n` times, and we want to perform `n` multiplications in total, it seems like a good idea to do one multiplication each time we go through the loop. We'll need to somehow keep track of the numbers we're multiplying together, probably using a variable. But we're already using `n` for our countdown, so we have to pick a different variable. How about a new one called `total`? It'll act like a running total of the result so far, starting at 1. Then we'll multiply it by `n` in the first iteration, `n - 1` in the second iteration, `n - 2` in the third iteration, until eventually 1 in the last iteration. At the very end `total` will equal the factorial of `n` so we can just use it as our output.

```
def factorial(n):
    total = 1
    while n > 0:
        ...
        n -= 1
    return total
```

Notice how we defined `total` as 1, before the start of the `while` loop. If we put it inside the loop instead, then we'd be repeatedly resetting it to 1 every time through the loop. That would be a problem.

All that's left is the inside of the `while` loop. We want to multiply `total` by `n` in the first iteration, `n - 1` in the second iteration, `n - 2` in the third iteration, and so on. But remember, we're decrementing `n` in each iteration of the loop so by the time we reach the second iteration `n` is bound to one less than its inital value, by the time we reach the third iteration `n` is bound to two less than its initial value, and so on. See if you can complete the problem on your own before looking at the complete solution below. Once you look at the answer, make sure you understand it. Re-read this section or draw a pyagram, if that helps.

```
def factorial(n):
    total = 1
```

```
        n -= 1
    return total
```

Just a minor note, it's exactly the same if you replace the `while` condition with `while n`, rather than `while n > 0`. That's because 0 is the only falsey number, so the condition would still be truthy when `n` is positive, and falsey as soon as `n` becomes equal to 0.

You could also solve `factorial` by counting up from 0 to `n`, instead of counting down from `n` to 0. You just have to be careful not to forget `n`'s initial value. See if you can write a solution to `factorial` that counts up, instead of down.

```
def factorial(n):
    i, total = 1, 1
    while i <= n:
        total *= i
        i += 1
    return total
```

Usually after solving a problem, you'll want to check it's domain. In other words, for what values does it produce the right answer? For instance we might plug in a few numbers or try it out in the Python interpreter, just to make sure our solution to `factorial` works as expected. And indeed, both of the solutions above output 24 for the input 4, 120 for the input 5, and so on. But it's especially important to see if your solution works for less obvious inputs too, commonly called edge cases. Like, what happens if we do `factorial(0)`? We get 1, and that's good because mathematically that's how $0!$ is defined. So our function seems to work. Make sure you understand it, as well as how we got it, before reading on.

Practice: the nth Fibonacci number

Recall the definition of the Fibonacci sequence, from a few chapters back. The first two numbers are 0 and 1, and every subsequent number is the sum of the previous two. The first eight numbers in the Fibonacci sequence are 0, 1, 1, 2, 3, 5, 8, and 13. Write a function `fibonacci(n)` that iteratively calculates the `n`th number in the Fibonacci sequence.

```
def fibonacci(n):
    ...
```

So after the `n`th iteration, we'll know the `n`th Fibonacci number.

The first step is to decide on a condition for our `while` loop. It should be `True` as long as we want to keep going, and `False` as soon as we want to stop. So we should ask ourselves, when do we want to stop? Well, if we want the `while` loop to happen `n` times we could just count down from `n` to 0. How about we try `n > 0` for our condition, like we did with `factorial`?

```
def fibonacci(n):
    while n > 0:
        ...
```

But if we want that condition to eventually become `False`, we have to make sure we actually decrement `n` to 0. Let's add that bit too, otherwise `n` would never change so the condition would always be `True` and we'd be stuck in the loop forever.

```
def fibonacci(n):
    while n > 0:
        ...
        n -= 1
```

So now we have a `while` loop that happens `n` times. And in each iteration, we want to get the next number in the Fibonacci sequence. That way the `n`th time through the loop we'll get the `n`th Fibonacci number, which is our desired result. For this to work, we'll have to keep track of which Fibonacci number we're on. We can't store it with the variable `n` though, since we're using `n` for our countdown. Let's add a new variable, `current`, to keep track of the current Fibonacci number. We'll start it off at 0, since that's the first Fibonacci number. Then each time through the loop, we'll update it to be the next Fibonacci number. After the `n`th iteration it'll equal the `n`th Fibonacci number so we can use it as our output.

```
def fibonacci(n):
    current = 0    # current starts at 1st Fibonacci number.
    while n > 0:
        ...        # Make current the next Fibonacci number.
        n -= 1
    return current # current ends up at nth Fibonacci number.
```

the loop. That would be a problem.

Okay, so now the hard part. Inside the `while` loop, how do we compute new Fibonacci numbers? Since every Fibonacci number is defined in terms of the previous two, we'll need to know two Fibonacci numbers in order to compute a new one. That's a problem because we're only keeping track of one Fibonacci number, `current`. Luckily the solution is simple. If you want to keep track of more information, then use more variables! Let's add a variabled called `next`, to keep track of the Fibonacci number after `current`. It'll start off at 1, since that's the second Fibonacci number. Now that we have two Fibonacci numbers rather than just one, we'll be able to compute new ones.

```
def fibonacci(n):
    current, next = 0, 1
    while n > 0:
        ...
        n -= 1
    return current
```

Each time through the loop, we'll move on to the next two Fibonacci numbers. So in our first pass, we'll transition from the first and second Fibonacci numbers to the second and third ones. In our second pass, we'll transition to the third and fourth ones. And so on, until we finally reach the `n`th and `n + 1`th ones. See if you can complete the program, before looking at the finished solution below.

```
def fibonacci(n):
    current, next = 0, 1
    while n > 0:
        current, next = next, current + next
        n -= 1
    return current
```

After finishing the program, be sure to test it out with a pyagram or in the Python interpreter. It seems to behave as expected:

```
>>> fibonacci(0)
0
>>> fibonacci(1)
1
```

```
>>> fibonacci(3)
2
>>> fibonacci(4)
3
```

Note that in the solution above we could also return `next`, if we stopped one iteration earlier by changing our condition to `n > 1`. That would work for most inputs, but not all of them. In particular our program would be wrong about `fibonacci(0)`, returning 1 instead of 0. To catch problems like this, it's really important to test your code after you finish it.

Iterating infinitely

So far, we've been focusing on `while` loops where the condition starts out `True` and eventually becomes `False`. That's useful if you want to repeat something for a while but eventually stop doing it. What if you want to repeat something forever? In order for this to happen, all we need is a condition that never becomes `False`. So, how about `while True`? We'll keep repeating the `while` loop over and over again, and never stop. The code below, for example, prints all the integers from 0 to infinity.

```
i = 0
while True:
    print(i)
    i += 1
```

However, note that `while True` doesn't guarantee a program will never stop. For example, the following function only goes through one iteration before it stops:

```
>>> def example(n):
...     while True:
...         n += 1
...         return n
...
>>> example(4)
5
```

Forcibly stopping a program

you no longer wanted the program running. There are a few ways to stop it. The easiest way is to shut down your computer. You could also quit the terminal app, if you're running the program in the Python interpreter.

If you're using a Mac, you can also press `Ctrl` + `C` from within the Python interpreter. This will forcibly stop the program's execution, without having to exit the Python interpreter. Here's an example:

```
>>> i = 0
>>> while True:
...     print(i)
...     i += 1
...
0
1
2
^C
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

The same usually works for Windows users, but it depends on what version you're using. Sometimes you have to press `Ctrl` + `D` or `Ctrl` + `Z` rather than `Ctrl` + `C`.

Practice: the Collatz sequence at n

The Collatz function $C(n)$ is defined mathematically as follows:

$$C(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{cases}$$

The Collatz sequence at a number n is defined by composing this function on itself repeatedly. That is, the Collatz sequence at n is given by $n, C(n), C(C(n)), \ldots$ and so it continues forever. The Collatz sequence at 3, for example, starts with the numbers 3, 5, 16, 8, 4, 2, and 1. It continues to repeat the numbers 4, 2, and 1 forever after that. Check that you understand everything so far, before you keep going. We're going to define a function `collatz(n)` that prints the Collatz sequence for a number `n`.

```
def collatz(n):
    ...
```

```
def collatz(n):
    while True:
        ...
```

Now we should ask ourselves, what do we need to repeat each time we go through this loop? First things first, we need to print the number we're at, since that's what the problem asked us to do.

```
def collatz(n):
    while True:
        print(n)
        ...
```

Now we have a program that just prints the same number over and over again. To get it to behave how we want, we'll have to update `n` to be the next number in the sequence, each time we go through the `while` loop. But the next number in the sequence depends on whether `n` is currently even or odd, so we'll need an `if` / `else` condition.

```
def collatz(n):
    while True:
        print(n)
        if n % 2 == 0: # n is even.
            ...
        else:          # n is odd.
            ...
```

Now we can just refer to the formula from before, to see what the new value of `n` should be. Notice that in the even case, we use floor division (`//`) instead of regular division (`/`) to keep the number an integer instead of converting it into a float.

```
def collatz(n):
    while True:
        print(n)
        if n % 2 == 0: # n is even.
            n = n // 2
        else:          # n is odd.
            n = 3 * n + 1
```

there's no point in specifying an output for a program that is meant to never end.

Counters and boolean flags

Sometimes we're doing a `while` loop and we want to keep track of whether something happens during our iteration, or perhaps how many times something happens during our iteration. As you might guess, we can use variables to keep track of that information.

Practice: testing the primality of a number

Suppose we want to make a function `is_prime(n)` to test whether a number `n` is prime. In other words, we want to know if there's a positive number smaller than `n` which is also a factor of `n`. One way we might do that is to individually check all the numbers less than `n`, using a `while` loop, to see if any of them divide into `n` without a remainder. If we encounter a number that's a factor of `n`, we know `n` isn't prime so we can output `False`. Meanwhile, if we finish checking all the numbers smaller than `n` and we still don't find one that's a factor, then we know `n` is prime and we can output `True`. Try writing such a program, and then look at the solution below. For simplicity you can assume `n` is at least 2.

```
def is_prime(n):
    i = 2
    while i < n:
        if n % i == 0:
            return False
        i += 1
    return True
```

Make sure you understand `is_prime(n)` before moving on.

Now imagine we want to modify this function, so that it prints all the positive numbers less than `n`, and then returns whether `n` is prime. We might try just adding in a few `print` statements, to get something like this:

```
def is_prime(n):
    print(1)
    i = 2
    while i < n:
```

```
            return False
        i += 1
    return True
```

But there's a problem. This program successfully tell us whether `n` is prime, but it doesn't necessarily print out all the positive numbers less than `n`. That's because it might stop early, if it reaches the `return False` inside the loop. For instance, here's what we get when we try it out on 6. It only prints out 1 and 2, when it should've printed 3, 4, and 5 as well!

```
>>> is_prime(6)
1
2
False
```

So we need the `while` loop to finish running, even if we find that `n` has a factor. That means we're not allowed to have a `return` statement inside the `while` loop, because that would end it early and then we wouldn't get around to printing some of the numbers. We need to somehow keep track of whether we found a factor of `n`, and save that information for after the `while` loop is done. As usual when we want to keep track of more information, we can use a variable. Let's call it `has_factor`. Instead of returning `False` when we find a factor, we'll just set `has_factor` to `True` to indicate we found a factor of `n`. Then after the `while` loop is complete, we can just test whether `has_factor` is `True` or not.

```
def is_prime(n):
    print(1)
    i = 2
    while i < n:
        print(i)
        if n % i == 0:
            has_factor = True
        i += 1
    return not has_factor
```

But this will break on primes now, because `has_factor` won't be defined unless we find a factor. We should start it off `False`, before the beginning of the loop, to make sure it's defined no matter what.

```
    i, has_factor = 2, False
    while i < n:
        print(i)
        if n % i == 0:
            has_factor = True
        i += 1
    return not has_factor
```

Now our program works as desired. Check that you understand how it works.

`has_factor` is called a boolean flag, but again, terminology is pretty stupid. The important takeaway is that in order to let our `while` loop finish, we needed to save information for later. We did that with a variable.

Practice: counting primes

Now suppose we want to keep track of a numerical value, instead of a boolean value. The approach is pretty similar. Just like before, we can use a variable to save the desired information for later.

Write a function `count_primes(n)`, which prints all the numbers from 0 up to `n` and then returns the number of primes less than `n`. For the purposes of this problem, we'll also assume we have access to a version of `is_prime(n)` that works correctly for all inputs. We'll start out with a `while` loop that prints all the numbers from 0 up to `n`.

```
def count_primes(n):
    i = 0
    while i < n:
        print(i)
        ...
        i += 1
```

For each number that gets printed, we need to know whether it's prime. Let's use the `is_prime` function to find out.

```
def count_primes(n):
    i = 0
```

```
        if is_prime(i):
            ...
        i += 1
```

And whenever we come across a prime number, we need to make a note of it. We should keep track of how many we've seen, and we can do that using a new variable. Let's call it `count`, and increment it every time we see another prime number. It can start out at 0, since we have seen 0 primes before the `while` loop starts. Then at the end of our function, we just have to return `count`.

```
def count_primes(n):
    i, count = 0, 0
    while i < n:
        print(i)
        if is_prime(i):
            count += 1
        i += 1
    return count
```