Intro

This is going to be a short chapter, but you'll learn very powerful magic. A lot of programming tasks require you to do something over and over again. It could be anything from applying a function until some limit, to walking until you reach the end of the street. To approach these problems, you need a `while` loop.

Syntax

```
while CONDITION:
    Do something.
    Change the variables in the CONDITION.
```

This is the basic structure of a `while` loop. The `CONDITION` is an expression that can evaluate to `True` or `False`, usually depending on other variables. The inside of the `while` loop happens over and over again as long as the `CONDITION` is `True`. Once the `CONDITION` becomes `False`, the `while` loop stops. Here's an example.

```
>>> i = 0
>>> while i < 10:
...     print(i)
...     i += 1
0
1
2
3
4
5
6
7
8
9
>>> i
10
```

This `while` loop's condition is `i < 10`. Here's how it works:

1. When we start out, `i` equals `0`. Since `0 < 10`, the condition is `True`. We decide to enter the `while` loop.
2. We `print(i)`, so the value `0` is displayed.
3. We increment `i`, so it's new value is 1.
4. This ends the first loop. Since `1 < 10`, the condition is still `True`. We have to enter the `while` loop again.
5. We `print(i)`, so the value 1 is displayed.
6. We increment `i`, so it's new value is 2.
7. This ends the second loop. Since `2 < 10`, the condition is still `True`. We enter the `while` loop again.
8. This continues until `i` equals `10`. `10 < 10` is `False`, so we are done with the `while` loop. `i` is now `10`, and the last value we printed was 9.

If the `CONDITION` is always `True`, then the loop goes forever. If the `CONDITION` is always `False`, then the loop never happens at all.

```
>>> word = ''
>>> while True:
...     word += 'U'
...     print(word)
U
UU
UUU
UUUU
```

```
>>> while False:
...     print('triceratops')
>>>
```

In the future you might see a `return` statement inside a `while` loop. Remember, when Python sees the word `return`, it immediately stops the function. Even inside a `while` loop.

```
>>> def f(n):
...     while True:
...         n -= 1
...         return n
>>> result = f(5)
>>> result
4
```

Practice: factorial

The factorial of a number is the product of itself with every positive number below it. For example, the factorial of 4 is 4 * 3 * 2 * 1, which equals 24. Define the function `factorial`, which takes in a number `n` and returns the factorial of `n`.

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

We're going to have to cycle through all the numbers from `n` all the way down to `1`. This is a lot like the example on the previous page, except we're counting down instead. That means we'll be decrementing `n`, until it's `0`.

```
def factorial(n):
    while n > 0:  <-------- while CONDITION:
        ...       <-------- Do something.
        n -= 1    <-------- Change the variables in the CONDITION.
```

Since we are already using `n` like a counter, we'll need a different variable to keep track of our final `return` value. Our variable will act like a result so far. This is a common thing to do in a `while` loop.

```
def factorial(n):
    result = 1
    while n > 0:
        ...       <-------- Change result.
        n -= 1
    return result
```

Now all that's left to do is modify the `result` by multiplying by the current value of `n`. At the end, we will have multiplied `result` by all the numbers counting down from `n` to `0`. Try finishing the problem on your own, before looking at the answer on the next page.

```
def factorial(n):
    result = 1
    while n > 0:
        result *= n
        n -= 1
    return result
```

You could also solve factorial by counting up, instead of counting down. The solution isn't as good, though, because it takes up more lines and you might have limited space on an exam.

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

Practice: fibonacci

The first two fibonacci numbers are `0` and `1`. Every other fibonacci number is the sum of the two that came before it. Define a function `fibonacci(n)` that calculates the `n`th fibonacci number, using a `while` loop.

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

First, we need to decide on a condition for our `while` loop. Usually we count up or down to a number. In this case, we'll choose to count down from `n`, since the problem is asking us to cycle through `n` things (the fibonacci numbers). It would also work to count up to `n`, but that takes more lines, like in the previous example.

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

Since we're already using `n` like a counter, we need another variable to keep track of the current fibonacci number we're on.

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

Now we have a problem. We need to increment `current` to the next fibonacci number, but in order to do that, we also need to know what fibonacci number came before `current`! This is because each fibonacci number is defined as the sum of the previous two, so we need to know both of them. Luckily, the solution is simple. If you want more information, use more variables!

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

Notice we use `0` and `1` as the initial values for current and next. That's because those are the first two fibonacci numbers. Starting from two fibonacci numbers, we will be able to calculate all the rest. `next` becomes the new `current` fibonacci number, and the sum of both becomes the new `next` fibonacci number.

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