WORKING WITH DATA

# Day 8: While Loops

Welcome to week 2 of the 30 Days of Python series!

To start this week off, we're going to be talking about another type of loop called a `while` loop. We're also going to be looking at some new ways to control the flow of our loops.

If you're new to the series, you can find out what you missed here.

Also we've created a short recap of Days 6 and 7 that you can view here to refresh your memory.

## `while` loops

In the last post we took a look at `for` loops as a means of reducing code repetition. `for` loops are great for two kinds of repeated actions:

1. When we want to do something for each item in some group.
2. When we want to do something a set number of times.

There are, however, other kinds of repetition we might want to perform. For example, what if we want to perform an action as many times as needed until some condition is met? What about if we want to perform some action over and over again forever?

For this kind of repeated action, we can use a `while` loop instead.

A `while` loop in many ways is quite a bit simpler than a `for` loop. We just need to use the `while` keyword, followed by some condition to test. If the condition evaluates to a truthy value, the loop will run one iteration, and then it will test the condition again.

For example, we can write a `while` loop which is going to run until the user enters a value of 10 or higher.

```
user_number = input("Please enter a number: ")

while int(user_number) < 10:
        print("Your number was less than 10.")
        user_number = input("Please select another num
ber: ")

print("Your number was at least 10.")
```

First we gather an initial number from outside of the loop. We need to do this, because we refer to the name `user_number` as part of the `while` loop condition, so if we don't define this name beforehand, we're going to get a `NameError` .

Even if this weren't the case, we still wouldn't have a value for `user_number` , so what would we be checking?

Once a user enters a number, we hit the line where the `while` loop begins. We check the condition, and if the condition is met, we're going to run the loop body once. In the case above, the loop body is going to run if the user enters `9` or less going to run if the user enters `9` or less.

If we end up inside the loop body, we're going to end up printing the message about the number being less than `10`, and then we're going to ask the user for another number. Note that we assign this new value to `user_number`, which means when we test the condition again, we're now testing a new value.

If we named this variable something else, we'd just keep checking the original value over and over again, and of course nothing is going to change if the value stays the same. This can land us in hot water, because the loop is going to end up iterating infinitely, and not by design.

When we have a `while` loop where we're checking a value like this, make sure that we can change the value within the loop.

## Infinite loops

Sometimes an infinite loop is what we want, and there are many ways we can achieve this.

Much like with conditional statements, we don't need to use comparisons operators for our loop condition: we can just specify a value, and the truthiness of this value is then going to determine whether or not the loop runs the next iteration.

Really we can use any expression we want, because all expressions are going to evaluate to some value, and if we get a value, we can test its truth value. We can use function calls, or arithmetic operations, or we can refer to variables. The sky's the limit.

When we want to specify an infinite loop, we want to write an expression which is always going to be evaluate to a truthy value. A good example is the Boolean value, `True`, which is always going to evaluate to `True` if we test its truth value.

I don't recommend you run this code, but we could write something like this:

```
while True:
	print("Hello there!")
```

What's going to happen here, is Python is going to *very* quickly run many iterations of this loop, and we're going to end up with a constant stream of `"Hello there!"` printed to the console.

When we write an explicitly infinite loop like this, it's usually important that we have some way of stopping it, and this is most often accomplished with some kind of conditional statement in combination with `break`.

As an example, let's create the skeleton of a simple text menu. We're going to have a few options the user can select, and one of these options is going to close the menu. That option is going to be the string `"q"`.

```
while True:
	selected_option = input("Please enter 'a',
 'b', or 'c', or enter 'q' to quit: ")

	if selected_option == "a":
		print("You selected option 'a'!")
	elif selected_option == "b":
		print("You selected option 'b'!")
	elif selected_option == "c":
		print("You selected option 'c'!")
	elif selected_option == "q":
		print("You selected option 'q'! Quitti
ng the menu!")
		break
	else:
		print("You selected an invalid optio
n.")
```

We can imagine that instead of simply printing the user's selection, each of these branches would perform some actions on behalf of the

user, and then when we're done, we prompt the user for another option.
If the user enters an invalid option, we catch all of these cases with an `else` clause, and inform the user of their mistake.

If the user enters `"q"` , we let the user know that we're quitting the menu, and then we use a `break` statement to break out of the loop.

Note that for this style of loop, we don't need to define anything outside of the loop, because we don't have a condition that relies on some variable.

### Style note

If you start looking at other people's code, you may see some loops written like the one below, with `1` as the loop condition:

```
while 1:
        ... some actions here ...
```

This works, because `1` is a truthy value, just like all non-zero numbers. While it works, I personally would advise against using it, for the same reason I wouldn't advise you to write a loop like this:

```
while "llama":
```

Both `1` and `"llama"` are truthy values, but they're way less explicit than just writing `while True` . Being explicit, and making our code as easy to read as possible, is more important than saving on writing three characters.

## The `continue` keyword

We've use the `break` statement a number of times now, but it's not our only option for controlling the flow of our loops. Another option we have available to us is the `continue` keyword.

While `break` allows us to exit a loop, `continue` allows us to skip the remainder of the loop body for the current iteration.

For example, let's create a loop which only prints even numbers:

```
for number in range(10):
        if number % 2 != 0:
                continue
        print(number)
```

For each iteration of the loop, we use the modulo operator to determine whether or not the current number is divisible by `2`. If it isn't, we encounter this `continue` statement, which immediately moves us onto the next iteration of the loop. This means we skip the remainder of the loop body, and we don't print the number.

The output will therefore look like this:

```
0
2
4
6
8
```

While this example uses a `continue` statement in a `for` loop, they can of course be used with `while` loops as well.

In my opinion `continue` statements are not generally all that useful, but every now any again they can really help to simplify code with complex conditions.

## Using an `else` clause with loops

Back in day 5 we took a look at the `else` clause in the context of conditional statements, but I said at the time that we can use `else` with other structures as well. `for` loops and `while` loops are among those structures.

The `else` clause in the context of loops is a little bit weird, and it doesn't seem to make a lot of sense. It's certainly not as intuitive as it

is with an `if` statement. I think it's worth thinking about `else` as a "no break" clause when we use it with loops.

The reason I say it's good to think about `else` as meaning "no break", is because an `else` clause attached to a loop will only run if a `break` statement wasn't encountered during the execution of that loop.

For example, let's write a loop to determine whether or not a number is prime. A prime number is a number divisible only by itself and `1`. For example, `2`, `3`, `5`, `7`, `11`, and `13` are prime numbers.

One way we can determine whether or not something is prime is by dividing it by every number which comes before it. If none of these divisions produce an integer result, we know the number is prime.

We don't have to check every number though. If we find a division which produces an integer result, we know the number isn't prime, so we don't need to check any further. We can therefore break the loop.

If we break whenever we find a number is not prime, this means that if we complete the loop, and therefore didn't encounter a `break` statement, we have a prime number. These are also the conditions for triggering an `else` clause.

```
# Get a number to test from the user
dividend = int(input("Please enter a number: "))

# Grab numbers one at a time from the range sequence
for divisor in range(2, dividend):
        # If user's number is divisible by the curent
 divisor, break the loop
        if dividend % divisor == 0:
                print(f"{dividend} is not prime!")
                break
else:
        # This line only runs if no divisors produced
 integer results
        print(f"{dividend} is prime!")
        print(f"{dividend} is prime!")
```

We can do something similar with a `while` loop as well:

```
# Get a number to test from the user, and set the initial divisor to 2
dividend = int(input("Please enter a number: "))
divisor = 2

# Keep looping until the divisor equals the number we're testing
while divisor < dividend:
        # If user's number is divisible by the curent divisor, break the loop
        if dividend % divisor == 0:
                print(f"{dividend} is not prime!")
                break

        # Increment the divisor for the next iteration
        divisor = divisor + 1
else:
        # This line only runs if no divisors produced integer results
        print(f"{dividend} is prime!")
```

There are more sophisticated ways to find prime numbers, but these serve well enough as demonstrations. If you'd like a challenge, see if you can make them more efficient.

We also have a short blog post with more examples of using `else` if you're interested.

## Exercises

Congratulations on completing the first week!

Hopefully this wasn't too difficult, but if you're having trouble with some of the concepts, the exercises below should help solidify everything. If you get stuck, we're always around to help on our Discord server, so feel free to drop by and ask any questions.

1) Write a short guessing game program using a `while` loop. The user should be prompted to guess a number between `1` and `100`, and you should tell them whether their guess was too high or too low after each guess. The loop should keeping running until the user guesses the number correctly.

2) Use a loop and the `continue` keyword to print out every character in the string `"Python"`, except the `"o"`.

Remember that strings are collections, and they are iterable, so you can iterate over the string, which will yield one character at a time.

3) Using one of the examples from earlier—or a solution entirely of your own—create a program that prints out every prime number between 1 and 100.

You can find our solutions to the exercises here.

## Additional resources

In Python 3.8, we got a new piece of syntax called an *assignment expression*, which we can use to great effect in while loops. If you want to learn more, check out our post on assignment expressions.