A CHALLENGE: FIZZBUZZ

# Day 6: For Loops


Welcome to day 6 of the 30 Days of Python series! In this post we're going to be talking about `for` loops, which are going to allow us to repeat some action once for each item in a collection.

If you missed day 5, I'd recommend checking it out if you're not familiar with any of the following topics, or if you need a refresher:

- Boolean values
- Truth values and which values are falsy in Python
- Comparison operators
- `if` statements
- `elif` and `else` clauses
- Using truth values as conditions

Also we've created a short recap of Days 4 and 5 that you can view

here to refresh your memory.

## Why do we need loops?

Let's use an example from one of our previous posts where we defined a list of tuples representing films in a film library.

```
movies = [
        (
                "Eternal Sunshine of the Spotless Min
d",
                "Michel Gondry",
                2004
        ),
        (
                "Memento",
                "Christopher Nolan",
                2000
        ),
        (
                "Requiem for a Dream",
                "Darren Aronofsky",
                2000
        )
]
```

Here we have a list called `movies`, which contains a number of tuples. These tuples contain data in a set order: the first element in each tuple is the film title; the second element is the film's director; and the final element is the year of release.

Imagine we want to print out the contents of this movie list. Using our current knowledge, we might do something like this:

```
movie = movies[0]
print(f"{movie[0]} ({movie[2]}), by {movie[1]}")

movie = movies[1]
print(f"{movie[0]} ({movie[2]}), by {movie[1]}")
```

```
movie = movies[2]
print(f"{movie[0]} ({movie[2]}), by {movie[1]}")
```

Which will give us some output like this:

```
Eternal Sunshine of the Spotless Mind (2004), by Miche
l Gondry
Memento (2000), by Christopher Nolan
Requiem for a Dream (2000), by Darren Aronofsky
```

If we wanted to be a little more succinct, we could forego the `movie` variables, and chain together several subscription expressions:

```
print(f"{movies[0][0]} ({movies[0][2]}), by {movies[0]
[1]}")
print(f"{movies[1][0]} ({movies[1][2]}), by {movies[1]
[1]}")
print(f"{movies[2][0]} ({movies[2][2]}), by {movies[2]
[1]}")
```

If any of this is new to you, I'd recommend taking a look at the post for day 4.

There are a few problems with the kind of code we've written above. First of all, we have a lot of duplicate code, which makes this program hard to maintain. Imagine if we wanted to change the output from this:

```
Eternal Sunshine of the Spotless Mind (2004), by Miche
l Gondry
```

To this:

```
Eternal Sunshine of the Spotless Mind (2004) - Michel
 Gondry
```

We have to make this change in 3 different places. Even with this

We have to make this change in 3 different places. Even with this small number of movies, it's quite easy to miss one, or to make a small mistake in the formatting, which could lead to unintended differences in the output from movie to movie.

Now imagine we have 50 movies. Or 1,000 movies. Not only has our maintainability problem gotten so much worse, even writing this out the first time is going to be a bit of a nightmare.

Our code is also really brittle. We can't account for any changes to the `movies` list, and we need to know in advance how many movies there are, so that we can write an appropriate number of `print` calls.

Because this kind of code duplication presents such major challenges, modern programming languages give us a lot of tools to remove this kind of duplication. In this particular case, the tool we want is a `for` loop.

## What is a `for` loop?

A `for` loop in Python is a means of performing some set of operations for each item in a collection, or more generally, an iterable. In a very loose sense, we can think of an iterable as something which is capable of giving us values one at a time.

In the case of our movie library example, a `for` loop is going to allow us to get items from the `movies` list one at a time, and perform an action (or series of actions) for each movie. In this case, that action is printing the movie in some particular format.

Here is a concrete example of a loop for printing each movie in the `movies` list in the format we used previously:

```
movies = [
        (
                "Eternal Sunshine of the Spotless Min
d",
                "Michel Gondry",
                2004
```

```
                2004
        ),

        (
                "Memento",
                "Christopher Nolan",
                2000
        ),
        (
                "Requiem for a Dream",
                "Darren Aronofsky",
                2000
        )
]

for movie in movies:
        print(f"{movie[0]} ({movie[2]}), by {movie[1
]}")
```

As you can see, this kind of loop can be extremely powerful. The `movies` list could have 10,000 movies in it and we've managed to print them all with just two lines of code.

## Defining a `for` loop

The loop definition starts with the `for` keyword. This keyword is what tells Python that we want to define a `for` loop.

Directly after this we have a variable name, but this isn't a name we've defined elsewhere in our code. We'll come back to this in a moment.

Next we have the `in` keyword, followed by the iterable we want to grab values from. This whole line is capped off with a colon.

Indented underneath the block is the code we want to run for each item in our iterable. Just like with conditional statements, this indentation is important, as it indicates what code is associated with the `for` loop. This indented block is sometimes referred to as the body of the loop.

Okay, so what's going on with this name `movie` in our example?

Okay, so what's going on with this name, `movie`, in our example? We're actually defining a new variable as part of the loop, and this variable is going to allow us to refer to a given value in our iterable.

When we run a `for` loop, we begin *iterating over* the iterable we provided. This just means we're getting values from the iterable one at a time.

When we get a new value from our iterable, we assign it to this name we defined as part of the loop, and then we execute the code in the loop body. Once we're done, we grab a new value from the iterable, assign this new value to the loop variable, and run the loop body again with this new value.

Once we run out of values, the loop ends, and we move onto any code which comes after the loop.

If we think about this in terms of our movie library example, we define a loop variable called `movie`, and the iterable that we're grabbing values from is the `movies` list.

During the first iteration (cycle) of the loop, we grab the first item from `movies`, which is this tuple:

```
(
        "Eternal Sunshine of the Spotless Mind",
        "Michel Gondry",
        2004
)
```

This tuple is assigned to `movie`, and then we run the code in the loop body. We only have one line of code in the body, which looks like this:

```
print(f"{movie[0]} ({movie[2]}), by {movie[1]}")
```

As you can see, we're referring to this `movie` variable here, so on this first iteration of the loop, `movie[0]` is the string, `"Eternal`

`Sunshine of the Spotless Mind"` ; `movie[1]` is the string, `"Michel Gondry"` ; and `movie[2]` is the integer, `2004` .
We're now done running the code in the loop body, so we try to get another value from `movies` . Since we still have movies in the list, Python gives us the next tuple:

```
(
        "Memento",
        "Christopher Nolan",
        2000
)
```

This new tuple is assigned to `movie` , replacing the old value, and we run the loop body once again.

This happens again and again until we run out of movies.

An important thing to keep in mind is that the name, `movie` , is not important from a functionality perspective. Python isn't magically making the connection between the names `movie` and `movies` . We could do any of the following and it would work just the same:

```
for m in movies:
        print(f"{m[0]} ({m[2]}), by {m[1]}")


for film in movies:
        print(f"{film[0]} ({film[2]}), by {film[1]}")


for movie_details in movies:
        print(f"{movie_details[0]} ({movie_details[2]}
), by {movie_details[1]}")
```

It's just a variable name like any other we might define.

## Style note

While we can name our loop variables whatever we want, it's a good

idea to make sure they accurately describe what the data we're working with is.

In the example above, each tuple represents information about a single movie, so a name like `movie` is very appropriate and descriptive. A name like `x` wouldn't be, and it can make our code harder to reason about, especially in more complex loops.

## The `break` statement

Sometimes we don't want to iterate over an entire collection, or we want to stop looping under certain circumstances. In these cases, we can use a statement called `break`.

`break` is usually used in conjunction with a conditional statement, because otherwise it's going to run during the first iteration of the loop, which renders the loop basically moot.

One common way to use a `break` statement is to prevent unnecessary operations. For example, let's say we want to check that a certain movie is in this movies library we keep referring to.

```
movies = [
        (
                "Eternal Sunshine of the Spotless Min
d",
                "Michel Gondry",
                2004
        ),
        (
                "Memento",
                "Christopher Nolan",
                2000
        ),
        (
                "Requiem for a Dream",
                "Darren Aronofsky",
                2000
        )
]
```

```
,

for movie in movies:
          # Check the title of the current movie is Me
mento
          if movie[0] == "Memento":
                    # If the title is Memento, inform
 the user that the movie exists and break the loop
                    print("Memento is in the movie lib
rary!")
                    break
```

All we care about in this case is that the movie exists, so if it ends up being the first item, there's really no need for us to keep checking the rest. If there are 10,000 movies, we just potentially saved ourselves checking 9,999 movies.

Remember that in order to include the if statement inside the for loop, it needs to be indented with 4 spaces. Therefore, the body of the if statement needs to be indented with 8 spaces to be both inside the loop and the if statement.

## The `range` function

Python has a built in function called `range` which is capable of producing a series of integers according to some set pattern. For example, we might want to get a series of numbers starting from 1 and ending at 100, moving in steps of 3. In this case, we'd have `1`, `4`, `7`, `10`, `13`, `16`, `19`, etc.

We call the `range` function just the same as we do for something like `print` or `input`, but `range` can take a variable number of values.

The simplest way to define a `range` is to provide the stop value for the `range`:

```
range(10)
```

This is going to give us a sequence of integers starting at zero, and going up to, but not including, the stop value. `range(10)` is therefore going to give us a sequence like this:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Alternatively, we can provide a start value and a stop value. In this case, the start value comes first, and this value is included in the range. `range(10)` is therefore the same thing as writing `range(0, 10)`.

Let's say we want our `range` to start at 3 for some reason, and we want it to go up to, but not including 10. We would define a `range` like this:

```
range(3, 10)
```

Which would contain these values:

```
3, 4, 5, 6, 7, 8, 9
```

If we specify both a start and a stop value, we can optionally specify a step value. By default, this value is `1`, which means that the next number in the sequence is `1` higher than the number which came before.

If we wanted to instead get every other number, we could specify a step of `2`. If we wanted every third number, we'd specify a step of `3`.

```
range(0, 10, 2)  # 0, 2, 4, 6, 8
range(0, 10, 3)  # 0, 3, 6, 9
```

We can also specify a negative step value, which means we go

backwards from the start value to the stop value.

```
range(10, 0, -1)  # 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
```

Be careful, because if we specify a `range` where we can't get from one value to another, `range` isn't going to complain: it's just going to give us an empty sequence.

An example might be this:

```
range(0, 10, -1)
```

There's no way we can get from `0` to `10` in steps of `-1`, so `range` gives us nothing, rather than an infinite collection.

When you print a `range`, you might expect something like this:

```
print(range(0, 10))
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

But actually that's not what would be printed at all.

One interesting thing about `range` is that it's a lazy type. This means that our `range` doesn't actually contain all the values in this sequence. Instead, it just figures out what the next number should be when we ask for it.

This means that if you try to print a `range`, you're not going to see all the values:

```
print(range(1000))  # range(0, 1000)
```

All we get is a definition for the `range` we've generated. This range starts at `0`, goes to `1000`, and it uses a default step value. This makes `range` really memory efficient, because it doesn't have to store a collection of potentially millions of numbers.

`range` is an iterable, which means we can use it in `for` loops. It also means we can convert it to a list or tuple if we want to.

We can do this using the `list` and `tuple` functions, just like we converted integers, floats, and strings using `int`, `float`, and `str`.

```
numbers = list(range(10))              # [0, 1, 2, 3,
 4, 5, 6, 7, 8, 9]
immutable_numbers = tuple(range(10))   # (0, 1, 2, 3,
 4, 5, 6, 7, 8, 9)
```

## Using `range` in `for` loops

One thing `range` is really useful for is running a loop a set number of times.

If we generate a `range` using something like `range(10)`, this `range` is capable of providing ten numbers. If we iterate over this collection, `range` is going to give us these numbers one at a time, and we're going to run the loop body once for each number.

```
for number in range(10):
    print(number)

# 0
# 1
# 2
# ...
# 9
```

Even if we don't use the numbers `range` generated, we still know that we're going to get ten iterations out of the loop.

For example, we can print a given string ten times like this:

```
for number in range(10):
```

```
        print("Hello!")
```

Output:

```
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
```

When we write a loop like this, where the number we're generating is not being used, it's common convention to name the loop variable, `_` . This is a clear signal to readers of our code that the loop variable doesn't feature in our loop body.

## Style note

Sometimes you're going to encounter loops with `_` as the loop variable. This is perfectly legal, because `_` is a valid variable name, and Python therefore won't complain.

While this is legal, please don't use `_` in place of good descriptive names. It serves a specific purpose, which is to indicate to readers that the loop variable is not actually being used in the loop body.

A good time to use `_` as a loop variable is in situations like this:

```
for _ in range(10):
        print("Hello!")
```

Here we're just using the `range` to ensure a certain number of iterations. We never refer to the numbers that `range` is generating. This is good practice, and very helpful to readers accustomed to this naming convention.

We should never be using `_` in a loop like the one below, however:

```
for _ in range(10):
        print(_)
```

Here we're referencing `_` inside the loop body, which is not only very ugly, it's also throwing away the opportunity to use good, descriptive names.

## Exercises

1) Below we've provided a list of tuples, where each tuple contains details about an employee of a shop: their name, the number of hours worked last week, and their hourly rate. Print how much each employee is due to be paid at the end of the week in a nice, readable format.

```
employees = [
    ("Rolf Smith", 35, 8.75),
    ("Anne Pun", 30, 12.50),
    ("Charlie Lee", 50, 15.50),
    ("Bob Smith", 20, 7.00)
]
```

2) For the employees above, print out those who are earning an hourly wage above average.

Hint: you can use a for loop and two variables to keep track of the total wage and the number of employees. Then, use the two variables to calculate the average. Finally, add another loop that goes through the employees list again and prints out only those who have an hourly wage above the calculated average.

You can find our solutions to the exercises here.

## Project

Once you're finished with these exercises, we've got another mini-project for you to try! This time we're going to be tackling a common coding interview question: Fizz Buzz.

## Additional Resources

If you ever need to remind yourself of the `range` options, if you want more detailed information about it, you can look it up in the official documentation.

You can also find more information about `for` loops here.

---