# Notebook

February 7, 2019

## 1 More Loops and Control Flow

In this lesson, we'll think about more ways to direct Python to do things repeatedly, or conditionally.

Let's start with more loops. I showed you the while loop before, remember?

```
In [1]: number = 10
        while number > 0:
            print(number)
            number = number - 1
```

```
10
9
8
7
6
5
4
3
2
1
```

There's another kind of loop, called a `for` loop, which is a little bit more complex. Here's how one looks:

```
In [2]: for letter in "cat":
            print(letter)
```

```
c
a
t
```

Here's how that works: some kinds of data types in Python are known as *iterables* (we'll look at what those are a bit later). Basically, that's anything that can be represented as a list of something else. So "cat" for example can be represented as a list of letters c, a, and t.

Let's go into the details.

- The `for` keyword starts our for loop.
- Next comes a variable name, which gets assigned to every element of the iterator in the body of the loop. Here the variable is `letter`
- Next comes the keyword `in`.
- Next comes the iterator we're iterating over, and then a colon as usual to start a block.
- Finally, the body of the loop, indented as usual.

The for loop will loop over each element of the iterator, applying the body of the loop to each element.

Importantly, loops (for loops or while loops) don't create their own scopes. That means that they can accidentally overwrite the value of some other variable in the scope that the loop is in. For example:

```
In [3]: letter = "Dear Sir or Madam:\nI am a very good lawyer. Please give me a job.\n\n-You."
        print("letter is: ")
        print(letter)

        for letter in "meow":
            print(letter.upper())

        print("now, letter is: ")
        print(letter)
```

```
letter is:
Dear Sir or Madam:
I am a very good lawyer. Please give me a job.

-You.
M
E
O
W
now, letter is:
w
```

A common use of for loops is to do something a specified number of times. In order to do so, we use the built-in `range()` function, which produces an iterator that starts at 0 and goes up to the number *before* the number you pass to it. (Actually, `range()` technically isn't a function, but we'll treat it like one here. See the Python docs if you want more.)

```
In [4]: for number in range(10):
            print(number)
```

```
0
1
```

```
2
3
4
5
6
7
8
9
```

```
In [5]: for ruby_slipper in range(3):
            print("There's no place like home!")
```

```
There's no place like home!
There's no place like home!
There's no place like home!
```

`range()` actually takes 3 arguments, `start`, `stop`, and `step`. Start is the number to start at, and has a default value of 0. Stop is the number to stop at. Step is the number to go by, and has a default value of 1. That means that you can do more complicated ranges by supplying values in place of those defaults.

```
In [6]: for n in range(5, 10):
            print(n)
```

```
5
6
7
8
9
```

```
In [7]: for n in range(0, 10, 2):
            print(n)
```

```
0
2
4
6
8
```

```
In [8]: for n in range(10, 0, -1):
            print(n)
```

```
10
9
8
```

```
7
6
5
4
3
2
1
```

## 1.1 Conditions

One thing you'll often find yourself wanting to do is to test whether something is, or is not, the case, and then have your program do different actions depending on the result. For example, suppose you wanted to know whether a number was even or odd? Here's how you'd find out.

```
In [9]: def odd_or_even(number):
            if number == 0:
                return "Zero isn't even or odd."
            elif number % 2 == 0:
                return "Even!"
            else:
                return "Odd!"

In [10]: print(odd_or_even(5))

Odd!


In [11]: print(odd_or_even(100))

Even!


In [12]: print(odd_or_even(0))

Zero isn't even or odd.


In [13]: print(odd_or_even(-5))

Odd!
```

So let's go over that. We know what defining a function looks like already, that was the last lesson. So let's look at the body.

The first line of the body can pretty much be read straight off: if the number is equal to 0, then do the thing in the indented block to follow; in this case, return a scolding about trying to pretend that 0 is even or odd.

The second line in the body starts with a weird keyword, `elif` . That's just a quirky Python shorthand for "else if". It gives you a second alternative. If that alternative is true, then run the

code in the indented block to follow. You can have as many `elif` statements as you want in a conditional.

Incidentally, the `%` symbol is known as "modulo." It basically means the same as the remainder in division, so what this block is asking is whether the remainder of the number divided by 2 is equal to 0 or not; if it is, then that means it's divisible by 2, i.e., even.

The final line is the last alternative, which starts with the keyword `else`. That is what happens if nothing in the `if` or `elifs` above is true.

Here are some things you should know about these conditionals:

- The different conditions are checked in order, and they stop after a match is found. What that means is that changing the order can change the behavior.

```
In [14]: def broken_odd_or_even(number):
             if number % 2 == 0:
                 return "Even!"
             elif number == 0:
                 return "Zero isn't even or odd."
             else:
                 return "Odd!"

         print(broken_odd_or_even(0))
```

```
Even!
```

We got the wrong answer because we checked the wrong thing first. $0/2 = 0$ with no remainder, so the first condition is true, which means that it never gets to test the second condition, the conditional just resolves.

- The stuff after an `if` or an `elif` or an `else` is an expression that evaluates to a *boolean*. A boolean is just one of the special values `True` or `False`. `10 % 2 == 0` evaluates to `True`, whereas `5 % 2 == 0` evaluates to `False`

```
In [15]: 10 % 2 == 0
```

```
Out[15]: True
```

```
In [16]: 1 + 1 == 3
```

```
Out[16]: False
```

```
In [17]: 5 > 10
```

```
Out[17]: False
```

```
In [18]: 10 > 5
```

```
Out[18]: True
```

```
In [19]: 10 > 10
```

```
Out[19]: False

In [20]: 10 >= 10

Out[20]: True
```

You should be able to figure out the basic comparison operators for those examples. It's important to know that double equals == is used for comparision, while single equals = is used to assign something to a variable. It's a common mistake to mix them up (I still do it myself sometimes).

- In Python you can use other things to stand in for booleans in conditionals. The rules are that most values act like `True` except for `None`, `0`, `False` itself (of course), strings with no characters in them, and empty compound data structures like lists and dictionaries (which we'll talk about later). Things that can stand in for `True` are colloquially known as "truthy", while things that stand in for `False` are known as "falsy

```
In [21]: if 0:
             print("truthy")
         else:
             print("falsy")

falsy


In [22]: # I'm too lazy to keep typing that, so let's stick it in a function
         def truthy_or_falsy(value):
             if value:
                 print("truthy")
             else:
                 print("falsy")

In [23]: truthy_or_falsy("1")

truthy


In [24]: truthy_or_falsy(None)

falsy


In [25]: truthy_or_falsy(False)

falsy


In [26]: truthy_or_falsy(True)

truthy
```

```
In [27]: truthy_or_falsy("cat")
truthy

In [28]: truthy_or_falsy("")
falsy

In [29]: truthy_or_falsy([1, 2, 3])
truthy

In [30]: truthy_or_falsy([])
falsy
```

- You can omit the "else" statement in a conditional, what this just means is that if it doesn't meet a condition, it just won't do anything. For example:

```
In [31]: if 0:
             print("I'm not doing anything")

In [32]: if 0:
             print("please can I do something?")
         elif None:
             print("I really want to do something, but I can't!!")
```

Functions end their execution whenever they see a `return` statement (this is known as "sort-circuiting"), so if you have a conditional with returns in a function, then it never sees the rest of the function.

A common idiom in Python is to take advantage of this last feature and just put the last alternative as a return statement outside of the conditional. For example, we could rewrite our `odd_or_even` function as follows:

```
In [33]: def odd_or_even(number):
             if number == 0:
                 return "Zero isn't even or odd."
             elif number % 2 == 0:
                 return "Even!"
             return "Odd!"
```

This function will work exactly the same.

- If it gets 0, it'll go down the first branch, return on the spot and never see the rest of the return statements.
- If it doesn't get 0, but does get a number divisible by 2, it'll go down the second branch, and ditto.
- If it doesn't get 0 or a number divisible by 2, it won't go down either branch of the conditional. But there's another line in the function! And since it didn't return in the conditional, it gets to that line, which tells it to return "Odd!"

### 1.1.1 Now you try!

I'd like you to try and write a function, `fizzbuzz()`, that takes a number, and returns:

- "Fizz" if the number is divisible by 3 and not 5,
- "Buzz" if the number is divisible by 5 and not 3,
- "FizzBuzz" if the number is divisible by 5 and 3", and
- the number itself, if it isn't divisible by 5 or 3.

Then I'd like you to print the result of that function for the numbers 1-100.