A CHALLENGE: FIZZBUZZ

Day 5: Conditionals and Booleans

Welcome to day 5 of the 30 Days of Python series! In this post we're going to be talking about conditionals. This is a big step forward in our programming journey, because we're going to be able to make decisions in our programs for the first time.

If you missed day 4, I'd recommend checking it out if you're not comfortable with any of the following:

- Defining lists and tuples
- Accessing items in sequences by index
- Nested collections
- Modifying lists (adding and deleting items)

Booleans

Booleans

Before we look at conditionals, first we need to learn about a new type called Booleans. Booleans are named after the mathematician, George Boole, who defined an algebraic system for determining the truth value of logical expressions.

Every modern programming language features some representation of Boolean values, because we very often need to be able to declare that something is or is not the case.

We only ever have two Boolean values, and in Python those values are the words `True` and `False`. The capitalisation is important here: `True` is a Boolean value, while `TRUE` and `true` are not.

Importantly, `True` is also not the same thing as the string `"True"`.

We can use these Boolean values just like any other values we've used in Python so far. We can assign them to variables, and we can put Booleans in tuples and lists.

For example, maybe we want to store information about some movies in a user's watch list. We can use a tuple to represent each movie, and we can use a Boolean to represent whether or not this movie has been seen by this user:

```
movies = [
        ("Inside Out", 2015, True),
        ("Toy Story 4", 2019, False),
        ("Up", 2009, True)
]
```

In the example above, the `movies` list contains three movies. Of these three movies, the user has seen Inside Out and Up, but not Toy Story 4.

We can also print Boolean values, and the output is just `True` or `False` as appropriate.

```
print(True)   # True
print(False)  # False
```

Truth values

Every value in Python has some associated truth value. This can be a little bit weird to think about at first. It's not particularly intuitive that `"Hello"` is somehow associated with either `True` or `False`, but this can be useful, as we'll soon see.

We can find the truth value of any value in Python by passing that value to the `bool` function. `bool` will return `True` or `False` depending on the truth value of what we passed in.

Values for which `bool` returns `True` are often called truthy values, while values for which `bool` returns `False` are often called falsy.

As an example, if we pass the string `"Hello"` to `bool`, the function call is going to evaluate to `True`. `"Hello"` is therefore a truthy value.

```
print(bool("Hello"))  # True
```

Let's look at some other examples:

```
print(bool(0))              # False
print(bool(6))              # True

print(bool("Caterpillar"))  # True
print(bool(""))             # False

print(bool([]))             # False
print(bool([0, 1, 2, 3]))   # True

print(bool(True))           # True
print(bool(False))          # False
```

As we can see, passing `True` or `False` to `bool` gives us `True` and `False` respectively, but there doesn't seem to be much rhyme or reason when it comes to the other values.

While it may appear basically random, there is in fact a pattern here. Only a small number of values in Python are falsy, which are as follows:

- Any numeric representation of zero. This includes the integer `0`, the float `0.0`, and representations of zero in other numeric types.
- The values `False` and `None`. We haven't looked at `None` yet, but `None` represents the intentional absence of a value.
- Empty sequences and other collections. This includes empty strings, empty tuples, empty lists, and several types we haven't covered at this stage.

It's also possible for us to define our own types, and we can make them falsy under certain circumstances if we wish. This is an advanced topic, however, and not something we're going to be covering in this series.

Apart from these values, everything else in Python is truthy.

Comparison operators

In addition to the `bool` function, we have a number of comparison operators that yield a Boolean value. We have eight of these in total, five of which you've probably seen before in a maths class.

First up, we have the `<` (less than) and `>` (greater than) operators. In the case of `<`, the operator yields the value `True` if the first operand is *less than* the second operand; for `>`, the operator yields `True` if the first operand is *greater than* the second operand.

Generally speaking, this operator is going to be used to compare two numeric types, but we can actually compare other values as well. For

example, we can compare strings, in which case the ASCII codes of the characters are used to determine which value is greater than the other.

```
print(5 < 10)      # True
print(5 > 10)      # False
print(10 > 10)     # False
print("A" < "a")   # True

# The ASCII code for A is 65, while a is 97
```

Very similar to these operators we have `<=` (less than or equal to) and `>=` (greater than or equal to). These work the same way as `<` and `>`, but if the operands are of equivalent value, we get `True` back instead of `False`.

```
print(10 > 10)   # False
print(10 >= 10)  # True
```

Quite often we want to explicitly check that two values are equal. In that case, we can use the `==` operator. `==` yields `True` if the two values are the same, and `False` in all other cases.

We have to be a little careful here and remember that things like `"0"` are not the same as `0`. Python is smart enough to understand that something like `7` and `7.0` are equivalent though.

```
print(0 == "0")                 # False
print(0 == 0)                   # True
print(7 == 7.0)                 # True
print("Hello" == "Hello!")      # False
print([1, 2, 3] == [1, 2, 3])   # True
```

If we want to check that two values are *not* equal, we have operator for this as well, which looks like this `!=`. This exclamation mark, sometimes pronounced as "bang", is commonly used to mean "not". So `!=` simply means "not equal".

```
print(0 != "0")          # True
print(0 != 0)            # False
print("Hello" != "Hi")   # True
```

The last two we have to look at are a little bit more complicated, because they're not comparing the values, *per se*. These last two operators are `is` and `is not`.

Your first instinct might be to assume that `is` and `==` are the same things, and the same goes for `is not` and `!=`. They are in fact very different, and perform very different jobs.

First let's look at an example:

```
a = [1, 2, 3]
b = [1, 2, 3]

print(a == b)  # True
print(a is b)  # False
```

Here we have two lists which contain the same values. We can therefore say that the lists are the same, and the `==` operator gives us `True` when we compare the two. However, the `is` operator gives us `False`.

The reason we get this difference is because `is` actually checking that the two lists are exactly the same list. Not that they have the same values, but that they are exactly the same thing.

To make a real world analogy, we might have two identical bowls, which each contain the same arrangement of coloured balls. The `==` operator is checking that the two bowls contain the same arrangement of coloured balls, while `is` is checking is we're talking about the exact same bowl.

When we're talking about what means to be the exact same thing in

Python, what we're actually concerned with, is whether the things we're comparing are stored at the same location in memory.

We can use a function called `id` to find out where something is being stored, represented as a long series of numbers. This long series of numbers is an address that references a location in memory. We can print these memory addresses to verify that the two lists are not in fact the same:

```
a = [1, 2, 3]
b = [1, 2, 3]

print(id(a))  # 139806639351360
print(id(b))  # 139806638418944

print(a == b)  # True
print(a is b)  # False
```

You're going to get different numbers to me for each list, and different numbers every time the program runs, but what's important is that the two numbers are not the same.

We can make the two lists the same by making a small change. Instead of defining this new identical list when assigning to `b`, let's just refer to the one we assigned to `a`:

```
a = [1, 2, 3]
b = a

print(id(a))  # 139685763327296
print(id(b))  # 139685763327296

print(a == b)  # True
print(a is b)  # True
```

Now both of our memory addresses are the same, and as a result, the `is` operator yields `True` when we compare `a` and `b`.

Important

It's important to note that, in some complex cases, `is` will yield `False` when we compare two things that seemingly occupy the same memory address. In other words, there are cases were two items seem have the same `id`, but `is` nonetheless indicates that the objects are not one and the same.

This happens when a memory address has been reused.

This isn't something you need to worry about throughout this course, and likely not for a long time after, but it's something to at least be aware of.

Order of operations

One final important note is how these comparison operators interact with the arithmetic operators we learnt about previously.

Comparison operators are always lower priority than arithmetic operators. For example, if we write an expression like this:

```
5 + 4 < 3 * 2
```

The comparison is effectively:

```
(5 + 4) < (3 * 2)
```

Which is:

```
9 < 6  # False
```

Conditional statements

Now that we know how to use these comparison operators, we can use the results of these comparisons to control the flow of our application. We do this using conditional statements. The most basic conditional statement uses a single keyword: `if`.

Conditional statements allow us to run some block on code if and only if some condition is met. We can use this structure to control the flow of our application based on whether or not something is the case. For example, we might only permit a user to log into a website if they enter the correct credentials.

Let's write a simple bartender program. We're going to ask the user for their age, and then we're going to check if they're less than 18 years old. If they're under 18, we're going to print a message stating that they're underage.

```
age = int(input("How old are you? "))

if age < 18:
          print("Sorry, we can't serve you.")
```

For the conditional itself, we start with this `if` keyword, then we have some expression followed by a colon. Below this line, we have some block of code we want to run if the expression we wrote evaluates to a truthy value.

This block of code we want to run if the condition is met has four spaces of indention at the start of the line. This indentation is *really* important. If you don't do this, Python is going to raise an exception.

```
File "main.py", line 4
    print("Sorry, we can't serve you.")
    ^
IndentationError: expected an indented block
```

The reason we need an indentation like this is because we need some way of signalling to Python what code should be bundled up with this condition. Everything we want to depend on this condition being met should be indented to the same indentation level.

Let's expand on our bartender program a little bit. Once we've

verified the user is in fact over 18, I want to be able to prompt the user for their drink of choice.

We have another conditional key word available to us called `else` which allows us to do something if the condition we're checking is not met. `else` can't be used on its own, and it needs to be attached to another structure. In this case, we're combining it with an `if` statement, but there are some other interesting options that we'll talk about in future posts.

Here is our updated program:

```
age = int(input("How old are you? "))

if age < 18:
          print("Sorry, we can't serve you.")
else:
        chosen_drink = input("What can I get for you?
 ")
```

Now if a user is under 18, they'll be told we can't serve them, but anyone who enters 18 or more for the initial prompt will be asked what they'd like to drink.

We could also write this conditional block like this:

```
age = int(input("How old are you? "))

if age >= 18:
        chosen_drink = input("What can I get for you?
 ")
else:
        print("Sorry, we can't serve you.")
```

We're now checking for the condition which would allow the user to proceed, which is more common.

Sometimes we need even more granularity than this, in which case, we'd use an `elif` clause. `elif` is a lot like `if` in that we need to specify an expression to test. Much like `else` , it can't exist on its

own: we have to combine it with an `if` statement.

When we include multiple conditions like this, Python is going to go through our conditions one at a time, until one is found to be true. Once it finds a true condition, it executes the code associated with that condition, after which no further conditions are checked for this `if` statement. If none of the conditions are found to be true, Python executes the code indented under the `else` clause.

The fact that Python doesn't necessarily check all of the conditions is important to keep in mind for a couple of reasons. First, it means that only the code for one condition is going to run. Second, it means that the order of our conditions matters, since Python is only going to run the code associated with the *first* matching condition.

Let's take a look at an `elif` clause in action by switching up our example a bit. We're going to once again ask for the user's age, but this time we're going to use a conditional statement to charge them different prices for a bus ticket. Those under 16 will be given a child rate, and those aged 60 and over will be entitled to a OAP rate. Everyone else will pay the standard ticket price.

We might write something like this:

```
age = int(input("How old are you? "))

if age < 16:
          print("You are eligible for the child rate o
f 80p.")
elif age >= 60:
          print("You are eligible for the OAP rate of
 £1.")
else:
          print("You must pay the standard rate of £1.
50.")
```

If we wanted, we could include more and more elif statements, which would allow us to have even more fine control over the application flow.

Truth values and conditional statements

We're not limited to just using comparison operators with conditional statements. For example, we can make use of the truth value of any value as a condition. This is where truth values become really useful, as they allow us to do things like succinctly checking if a collection is empty.

For example, we can check if the user actually entered anything when we requested some input:

```
name = input("Please enter your name: ")

if name:  # Checks the truth value of name by calling
 bool
          print(f"You entered {name}")
else:
          print("You didn't type anything")
```

We'll be reviewing this pattern in the upcoming posts, so don't worry if this is a little confusing right now.

Exercises

1) Try to approximate the behaviour of the `is` operator using `==`. Remember we have the `id` function for finding the memory address for a given value, and we can compare memory addresses to check for identity.

2) Try to use the `is` operator or the `id` function to investigate the difference between this:

```
numbers = [1, 2, 3, 4]
new_numbers = numbers + [5]
```

And this:

```
numbers = [1, 2, 3, 4]
```

```
numbers.append(5)
```

Are `new_numbers` and `numbers` the same thing? What about `numbers` before and after we append `5` ?

3) Ask the user to enter a number. Tell the user whether the number is positive, negative, or zero.

4) Write a program to determine whether an employee is owed any overtime. You should ask the user how many hours the employee worked this week, as well as the hourly wage for this employee.

If the employee worked more than 40 hours, you should print a message which says the employee is due some additional pay, as well as the amount due. The additional amount owed is 10% of the employees hourly wage for each hour worked over the 40 hours. In effect, the employees get paid 110% of their hourly wage for any overtime.

You can find our solutions to these exercises here.
