Boolean Problems and CodingBat

1 Turning a Word Problem Into Code

Consider the following silly problem:

> A pair of beavers can build a dam if the river is flowing slowly and they have wood. Let `fastRiver` be a `True` or `False` variable (which is called a `bool`) that is true if the river is flowing fast. Let `hasWood` represent whether the beavers have wood. Write a function `canBuildDam(fastRiver, hasWood)` that returns (aka outputs) `True` if the beavers can build their dam.

Reading this problem, we can recognize that we're using `True` or `False` to represent certain real world properties. `fastRiver` is `True` if the river is moving fast, so if the river is moving slow (aka not fast), `fastRiver` would be false. Similarly, if the beavers have wood, `hasWood` would be `True`, while if the beavers don't have wood, `hasWood` would be false.

`fastRiver`	Meaning
`True`	River is moving fast.
`False`	River is moving slow.

`hasWood`	Meaning
`True`	The beavers have wood.
`False`	The beavers have no wood.

Bools and What They Mean

Okay, so we know what the variables represent, but what would the function look like? Well, we know that we're trying to return `True` if the beavers can build their dam. And the beavers can only build their dam if the river is moving slow (so `fastRiver` must be false) and when they have wood (so `hasWood` is true). So the function would be:

```
def canBuildDam(fastRiver, hasWood):
    if (not fastRiver) and hasWood:
        return True
    else:
        return False
```

And viola! Our first boolean problem is solved. By boolean problem, what I mean is we wrote a function that takes in only `True` or `False` variables known as `bool`s, and then the function outputs `True` or `False` (aka a `bool`).

Problems involving `True` and `False` come up so much in programming as we use `bool`s for decision making in programs. Think of your own life and how you map out your day. You may make a mental list in your head like the following.

Plan for a Birthday Party

1. Check if we're out of any cake ingredients.
2. If out of milk, head to grocery store.
3. Check if any streamers from last year are in the basement.
4. If not, go to the party store.

This To-Do list is using booleans! If `outOfMilk` is `True`, then we need to go to the grocery store to buy milk. If `hasStreamers` is `False`, then we need to buy some more at the party store. `True` and `False` are a way to let decision-making be very precise, allowing us to tell computers, which are very rigid, to do exactly what we want. Mastering using booleans will serve you well as a programmer.

2 Using Tables to Solve Boolean Problems

Consider another silly problem:

> A couple is taking a stroll at a park. Write a function `isDry(raining, haveUmbrella)` that represents whether the couple will stay dry or not.

This time, this problem description didn't tell us what `raining` and `haveUmbrella` represent. Often coding problems won't give us all the details we would like. But the names of these variables make their meanings clear: `raining` is `True` if it is raining, and `haveUmbrella` is `True` if the couple has an umbrella.

`raining`	Meaning
`True`	It is raining.
`False`	It's not raining.

`haveUmbrella`	Meaning
`True`	The couple has an umbrella.
`False`	The couple does not have an umbrella.

More Variable Meanings

It's not that obvious when `isDry` should return `True`. We'll have to think logically. If it's not raining—that is, if `raining` is `False`—then we should return `True` always, as the couple won't get wet without rain. If it is raining, the couple will still not get wet if they have an umbrella. How could we represent these relationships? By a table!

		haveUmbrella	
		True	False
raining	True	Dry!	Wet.
	False	Dry!	Dry!

Converted to True and False

So it looks like the only scenario where we're wet is if it is raining and we have no umbrella. Since we return `True` when we are wet, and `False` when we are dry, we can rewrite the table as follows:

		haveUmbrella	
		True	False
raining	True	True	False
	False	True	True

How the Input Affects the Result

So if either `not raining` or `haveUmbrella` is `True`, then the couple is dry, so we return `True`. Therefore, our function would be:

```
def isDry(raining, haveUmbrella):
    if (not raining) or haveUmbrella:
        return True
    else:
        return False
```

So to summarize our method for boolean problems:

How to Write Boolean Functions:

- Determine what the input variables mean in English when they are `True` or `False`. A chart is handy here.
- Create a table that shows the result depending on the input.
- Determine when we return `True` and `False` based on the result. What does an output of `True` mean in the conext of the problem?

3 The Nitty-Gritty: What does `not`, `or`, and `and` do?

When we write "`if (not raining) or haveUmbrella`", what does `or` and `not` actually tell the computer? We understand `if` as allowing us to do code only some of the time. How about `and`, `or` and `not`? Is there a good understanding of them?

`not` Meaning

To the computer, `not` flips a `True` to a `False` and vice versa.

`raining`	`not raining`
`True`	`False`
`False`	`True`

So if we want code to run only if something is not the case, we can use `not`!

`and` Meaning

The keyword `and` lets us combine two variables and only output `True` if both input variables are `True`.

fastRiver	hasWood	fastRiver and hasWood
False	False	False
False	True	False
True	False	False
True	True	True

This chart (and the one for `not` is called a *Truth Table*, as it shows how inputs correspond to outputs. Notice we consider all possibilities: both inputs are false, one is true and one is false, and both are true. These are four possibilities, and the reason there is four is because

$$2 \text{ possibilities for one variable} \times 2 \text{ possibilities for another} = 4 \text{ possibilities}$$

`and` only outputs true if both inputs are true. Notice also that this chart could also be represented as a table like we did before with the `isDry` problem:

		fastRiver	
		True	False
hasWood	True	True	False
	False	False	False

Result of `fastRiver and hasWood`

`or` Meaning

The keyword `or` lets us combine two variables and output `True` if at least one variable is `True`. Here is an example truth table:

raining	haveUmbrella	raining and haveUmbrella
False	False	False
False	True	True
True	False	True
True	True	True

Once again we have $2 \times 2 = 4$ possibilities for the input, so there's four rows in our chart. Once again, we could represent this chart as a table:

		`fastRiver`	
		`True`	`False`
`hasWood`	`True`	`True`	`True`
	`False`	`True`	`False`

Result of `raining or haveUmbrella`

There is something interesting here about `or` in programming. In English, we may use the word "or" in a few different ways:

- "Can you hang out today *or* tonight?"
- "Would you like the pasta *or* the burrito for your meal?"
- "Would you like sugar or cream in your coffee?"

In response to the first question, you may answer "Yes, I can hang out today or tonight." What you mean is that either time works, and potentially you could hang out for the day through the evening. The `or` here is **inclusive**: Your friend answers yes (aka `True`) to this question if today works, tonight works, or both works.

Contrastly, question 2 expects *either* answewer but not both. You'll answer, "I'd like the pasta," or you might answer, "I'd like the burrito." It's unlikely you would answer, "I'd like the pasta and the burrito," and that answer certainly isn't what waiter intended when asking the question. This is called an **exclusive** or.

The third quesiton is another inclusive or example. It's a perfectly valid response to reply, "Yes, I would like sugar and cream in my coffee." You could also say, "I want just sugar in my coffee" or "I just want cream in my coffee, no sugar." This `or` here is inclusive: you are allowed to answer both.

The inclusive `or` is what the keyword `or` means in programming. Compare and contrast the truth tables of inclusive or and exclusive or:

a	b	a or b	exclusive_or(a, b)
False	False	False	False
False	True	True	True
True	False	True	True
True	True	True	False

Inclusive Or and Exclusive Or compared in one truth table.

Notice the one difference: `exclusive_or` is `False` when `a` and `b` are both `True`, while `a or b` is `True` as we allow for both to be `True`. Remember that `or` just means "at least one of the two variables is `True`." It is inclusive and it allows for the possibility of both being true.

So we know that `or` is inclusive or in Python. How do we type exclusive or? It turns out that the not equals operator, written as `a != b`, is the same as exclusive or. In math, this would be written as $a \neq b$.

$$a \neq b \qquad \text{in code is} \qquad \texttt{a != b}$$

Why is $a \neq b$ exclusive or? Let's look at the truth table!

a	b	exclusive_or(a, b)	a != b
False	False	False	False
False	True	True	True
True	False	True	True
True	True	True	False

Exclusive Or compared to Not Equal.

If a is false and b is false, then two variables are equal (`False == False`), so not equal is `False`. If both variables are true, then the two variables are equal (`True == True`), so not equal is `False`. So we get that not equal is only True when both bools don't match, which is the same as exclusive or: we want one but not the other.

4 Simplifying the Answer

Let's revisit our `canBuildDam` function:

```
def canBuildDam(fastRiver, hasWood):
    if (not fastRiver) and hasWood:
        return True
    else:
        return False
```

I want to remark that I put the parenthese around `not fastRiver` for clarity purposes: They are not necessary in Python due to the order of operations, which we will talk about eventually. So the following code still works correctly:

```
def canBuildDam(fastRiver, hasWood):
    if not fastRiver and hasWood:
        return True
    else:
        return False
```

In the case where `not fastRiver and hasWood` is `True`, the function immeadetly returns `True` and ends. So we don't actually need the `else` as there's no risk of the `return False` code running when it shouldn't. So the following code also works:

```
def canBuildDam(fastRiver, hasWood):
    if not fastRiver and hasWood:
        return True
    return False
```

Believe it or not, we can simplify even further. When `not fastRiver and hasWood` is `True`, we return `True`. When `not fastRiver and hasWood` is not `True` (aka `False`), we return `False`. So we can actually write:

```
def canBuildDam(fastRiver, hasWood):
    return not fastRiver and hasWood
```

and this code will work perfectly fine as well.

Do not feel pressured to simplify if you don't feel comfortable yet! I'm writing this section so that you understand code you see in the wild that does

drop `else` or `if` entirely. All the solutions above were entirely correct code, and you should be proud with whatever answer you come up with!

TODO: Write example with simplification of multiple if statements.

CodingBat Setup

TODO

Homework

TODO (but make sure include simplifying the isDry problem.) (Also make sure it has a 3 variable problem.)