Boolean Expressions and Conditionals

What if you only wanted code to execute if a certain condition is met (or in some cases, not met)? Well you can with the very neatly-named conditionals. Everytime you say something along the lines of "if {this is true} then do {this}", that is when you would use a conditional.

A very basic example is if you were deciding where to eat for lunch. This was my thought process yesterday as psuedocode:

```
if hungry
    if craving tacos and have money
        go to velvet taco
    else, if craving sandwiches
        go to potbelly
    else
        cook at home
else
    force myself to eat lunch at closest restaurant
```

So you can see that I have a lot of thought processes: I have to decide if I'm hungry or not and based on that decision, I have some more decisions to make to decide what to actually do. Python has keywords to represent each scenario keyword:

- `if` : For the `if` statements... literally
- `elif` : For other choices that you decide if the original `if` is not what you want (the `else, if`)
- `else` : For the choice you default to if you decide against all the other conditions

I'll write the code for above as Example 4-1, and you can play around with the boolean variables to see how they influence the results.

In [ ]:
```
# Example 4-1: Anthony's Weird Lunch Decision Tree
hungry = True
craving_tacos = True
craving_sandwiches = True
have_money = False

if hungry:
    if craving_tacos and have_money:
        print("We getting Velvet Taco!")
    elif craving_sandwiches:
        print("We getting Potbelly!")
    else:
        print("We got food at home, :copium:")
else:
    print("Go eat for once bruh")
```

```
We getting Potbelly!
```

Boolean Expressions

These are any expressions that evaluate to `True` or `False` , hence the name, "boolean" expressions. With math, you can perform basic mathematical operations to get numeric outputs (rarely strings). With booleans, they have their own set of operations you can use to create boolean expressions.

Before I dive into these operations (and their associated operators), I want to do a quick English example. Let's say I went shopping and I only want to buy a sandwich if it meets these conditions:

- Less than 5 USD in cost
- Hasn't expired yet
- Is actually a sandwich or at least has bread as a major component

As one sentence, I would say that I only get a sandwich if and only if it is less than 5 USD in cost, and it hasn't expired yet, and either it is a sandwich or has bread as a major component. As psuedocode, this would be:

```
if (cost less than 5 USD) and (has not expired) and (is sandwich or has bread)
```

Note that however I write this, there are only two possibilities, either the entire condition is `True` or the entire condition is `False` . In English, it is pretty obvious how I can calculate the final result of the above expression: just substitute in `True` or `False` for the conditions and simplify. For example, if I bought a 3 USD bread soup that hasn't expired, I would get:

```
if True and True and (False or True)
if (True and True) and (False or True)
if (True) and (True) => True
```

You can determine for yourself given the information that I would indeed buy the bread soup since it fits all my criteria. Now, this is a very basic example: you can create (and later simplify) complex expressions. These expressions may not be necessarily complex in terms of difficulty but rather in length.

Relational and Boolean Operators

You have two types of operators: relational and boolean. Despite their different names, they both are used to form conditionals and boolean expressions. In other words, they will output a boolean after evaluation. These operators are as follows:

Relational:

- < (less than)
- <= (less than or equal to)
- == (equal to)
- > (greater than)
- >= (greater than or equal to)

Boolean:

- `not` (negation)
- `and`
- `or`

All these operators are more or less self-explanatory. For the relational operators, if the expression is true, then `True` is the result, otherwise, the result is `False`. For the boolean operators:

- `not` is attached to the start of an expression and reverses the output
- `and` returns `True` if and only if both values to the left and right of it evaluate to `True`, else it returns `False`
- `or` returns `True` if and only if at least one value out of the two (the one to its left and the one to its right) is `True`

The truth tables are shown below:

a	b	not a	a or b	a and b
T	T	F	T	T
T	F	F	T	F
F	T	T	T	F
F	F	T	F	F

The tricky thing with the relational operators is that different types have different behaviors. Comparing two numbers works exactly the same as real life (for example, `9 < 10` is `True`). However, what happens if we compare two strings using a relational operator?

In [ ]:
```
# Example 4-2: String comparison
print("2" > "10")
print("12" < "3")
```

```
True
True
```

As seen with Example 4-2, they both evaluate to `True`! Numerically, this is false, however, we are not comparing numeric values but rather text values. I will not discuss how Python orders strings, but this is a common source of error for students: they forget to cast their `input()` to `int` or `float` before comparing the value to some other number.

If you want to chain multiple conditions, you can use a boolean operator. For example, if you wanted to ensure a variable is between two numbers, you would do:

```
x < num_max and x > num_min
```

for example. There is one detail that does separate Python from other languages though: you can directly chain the relational operators. In other words, I can actually rewrite the above as:

```
num_min < x < num_max
```

and the two expressions would be the same. This exists for boolean operators too, mainly for `and` and `or`:

```
a and b and c or d and e
```

Now that does get confusing so typically parentheses are used to help determine what actually is the left side of an operator and what actually is the right side of that same operator. Also note that these operators are processed in a certain order:

```
Parentheses -> Mathematical operators -> Relational operators -> Boolean operators (not -> and
-> or)
```

For example:

1. `not ((1 + 2) > 3) and (3 < 5 or 1 + 3 * 9 > 3) or 3 > 10`
2. `not (3 > 3) and (3 < 5 or 28 > 3) or 3 > 10`
3. `not (False) and (True or True) or 3 > 10`
4. `not (False) and True or 3 > 10`
5. `not False and True or False`
6. `True and True or False`
7. `True or False`
8. `True`

The color coding is kinda jank, but hopefully it helps with reading the text. As seen with Example 4-3, my result is correct. My steps are as follows:

1. Initial expression
2. Evaluate math in parentheses
3. Evaluate the relational operations in parentheses
4. Evaluate the boolean operation (`or`) in parentheses
5. Evaluate the remaining relation operation (not in parentheses)
6. Evaluate the `not`
7. Evaluate the `and`
8. Evaluate the `or`

As with math, anything in parentheses takes precedence. It's normal PEDMAS except you add relational operators and then boolean operators at the end.

In [ ]:
```
# Example 4-3: Verifying my result
print(not ((1 + 2) > 3) and (3 < 5 or 1 + 3 * 9 > 3) or 3 > 10)
```

```
True
```

If/Elif/Else

To let Python know to make decisions, we have to use specific statements:

- `if` : the initial condition
- `elif` : other conditions to check if the initial condition is `False`
- `else` : executed when all the above conditions are incorrect

You do not need `elif` or `else` in all cases, but you must always have `if` since `elif` and `else` have a dependency on `if` . The structures for the three can be represented with Example 4-4.

In [ ]:
```
# Example 4-4: If-elif-else, all combinations

a, b, c = True, True, True
# just if
if a:
    print("Hi!")

# if + elif
if not a:
    print("Hi")
elif b:
    print("Hello!")

# if + mutliple elif
if not a:
    print("Hi")
elif not b:
    print("Hello!")
elif c:
    print("Howdy!")

# if + elif + else
if not a:
    print("Hi")
elif not b:
    print("Hello!")
else:
    print("bye")

# if + multiple elif + else
if not a:
    print("Hi")
elif not b:
    print("Hello!")
elif c:
    print("Howdy!")
else:
    print("bye")
```

```
Hi!
Hello!
Howdy!
bye
Howdy!
```

Note that every `if` statement starts a decision that Python has to make: if you want Python to make alternate decisons based on other conditions in case the `if` statement, use `elif` , and if you want Python to make a decision if all the alternates and main conditions fail, use `else` .

Furthermore, note that the `print` statements are all indented. In order for Python to know what belongs in the `if` / `elif` / `else`

block, the code must be one indentation level higher than the `if` / `elif` / `else`. Once you go back to a regular indentation level, that's how Python knows that's the end of the code block.

In [ ]:
```
# Example 4-5: If blocks
a, b, c = True, True, True

if not a:
    print("Hi")
else:
    print("Yo!")
if not b:
    print("Howdy!")
elif c:
    print("Hello!")
else:
    print("Meow!")
```

```
Yo!
Hello!
```

As demonstrated by Example 4-5, the `elif` and `else` belong to the nearest `if` statement above it. The first `else` belongs to and depends on the first `if` and the first `elif` and second `else` belongs to and depend on the first `if`.

"pass" keyword

The `pass` keyword is a placeholder: it tells Python to go pass by it and go to the next line of code. This is really handy if you just want to make the `if` / `elif` / `else` structure without having to worry about Python yelling at you for indentation error or not having any executable code in your conditionals. Example 4-6 highlights `pass` in action.

In [ ]:
```
# Example 4-6: pass keyword

if 4 > 3:
    pass
```

As you can see, nothing is outputted or done although the conditional's expression evaluates to `True`.