**Comparison Operators**

Welcome back, budding Pythonistas! You have already learned a bit about different Python types.

**NumPy recap**

Among those was bool, short for boolean. Do you remember the bmi array from the intro course? Here it is again. Using the greater than sign, we could find out which values in bmi were above 23. Next, I used the resulting Boolean array to actually select that value. In this video, we'll dive a little deeper into the world of comparison operators, like this greater than sign. Comparison operators are operators that can tell how two Python values relate, and result in a boolean.

**Numeric comparisons**

In the simplest sense, you can use these operators on numbers. Say, for example, that you want to check if 2 is smaller than 3. You type 2 less than sign 3, and hit Enter. Because this is the case, you get True. You can also check if two values are equal, with a double equals sign. From this call, we see that 2 equals equals 3 gives us False. Makes sense, because 2 is not equal to 3. You can also make a combination of equality and smaller than. Have a look at this command that checks if 2 is smaller than or equal to 3. It's TRUE, but also 3 smaller than or equal to 3 is True. Of course, you can also use comparison operators directly on variables that represent these integers, like I did in this example.

**Other comparisons**

All these operators also work for strings. Let's check if "carl" is smaller than "chris". According to the alphabet, carl comes before chris, so the result is True. Do you think that comparing a string and an integer can work? Let's try to see if the integer 3 is smaller than the string chris. We get an error. Typically, Python can't tell how two objects with different types relate. Different numeric types, such as floats and integers, are exceptions as this example shows: no error this time. In general, always make sure that you make comparisons between objects of the same type.

**Other comparisons**

Another exception arises when we move back to the example we started with, where we compared the NumPy array, bmi, with an integer, 23. This works perfectly. NumPy figures out that you want to compare every element in bmi with 23, and returns corresponding booleans. Behind the scenes, NumPy builds a numpy array of the same size filled with the number 23, and then performs an element-wise comparison. This is concise yet very efficient code, something data scientists love!

**Comparators**

Have a look at this table that summarizes all comparison operators. You already know about some of these. They're all pretty straightforward, except for the last one maybe. The exclamation mark followed by an equals sign stands for inequality. It's basically the opposite of equality.

**Boolean Operators**

You're doing great! And now that you can produce booleans by performing comparison operations, the next step is combining these booleans.

**Boolean Operators**

You can use boolean operators for this. The three most common ones are and, or, and not.

**and**

The and operator works just as you would expect. It takes two booleans and returns True only if both the booleans themselves are True. This means that True and True evaluates True, but False and True, True and False and False and False all evaluate to False. Instead of using booleans, we can of course use the results of comparisons. Suppose we have a variable x, equal to 12. To check if this variable is greater than 5 but less than 15, we can use x greater than 5 and x less than 15. As you already learned, the first part will evaluate to True. The second part, will also evaluate to True. So the result of this expression, True and True, is True. This makes sense, because 12 lies between 5 and 15.

**or**

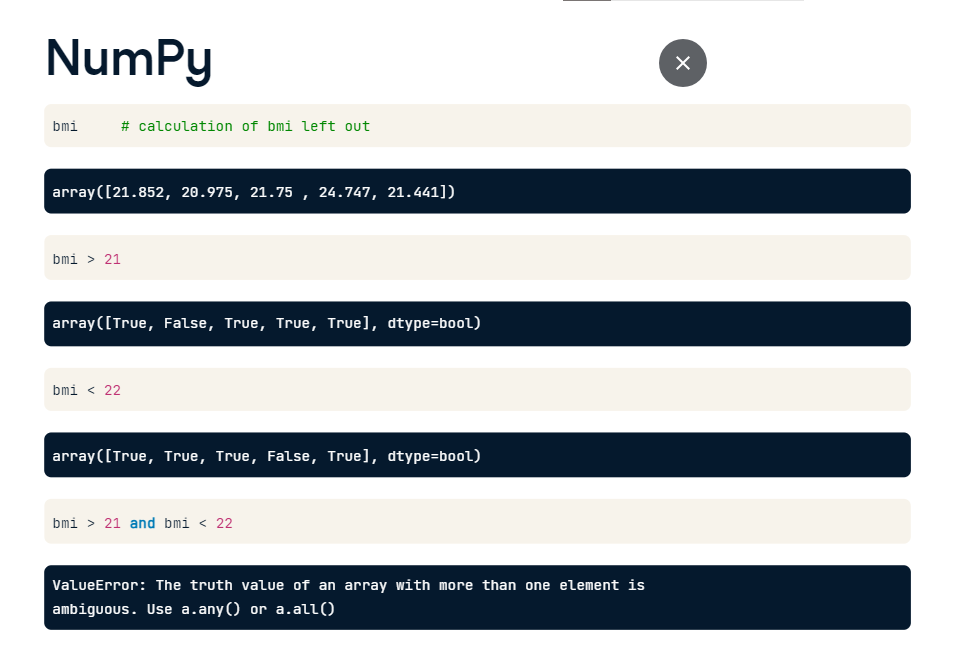
The or operator works similarly, but the difference is that only at least one of the booleans it uses should be True. This means that, True or True equals True, but that also False or True and True or False evaluate to True. Only False or False results in False. Also here you can make combinations with variables, like this example that checks if a variable y, which is equal to 5, is less than 7 or above 13. 5 less than 7 is True, 5 greater than 13 is False. The or operation thus returns True.

**not**

Finally, there's the not operator. It simply negates the boolean value you use it on. not True is False, not False is True. The not operation is typically useful if you're combining different boolean operations and then want to negate that result; you'll see some examples in the exercises.

**NumPy**

Now, for NumPy arrays, things are different. Retaking the bmi example from the intro course, we can try to find out which bmis are higher than 21, but lower than 22. The output of bmi greater than 21 is easily found, so is the one for the bmi lower than 22. Let's now try to combine those with the and operator I just introduced. Oops, an error. The truth value of an array with more than one element is ambiguous. and clearly doesn't like an array of booleans to work on.



After some digging in the numpy documentation, you can find the functions logical\_and, logical\_or and logical\_not, the "array equivalents" of and or and not. To find out which bmis are between 21 and 22, we thus need this call. Again, as we expect from NumPy, the and operation is performed element-wise: True and True give True, like these ones, but False and True or True and False give False, like for these elements. To actually select only these bmis from the bmi array, we can use the resulting array of booleans in square brackets. Again, NumPy wins when it comes to writing short yet very expressive Python code. I can hear you asking, "Cool, but how does this work for Pandas DataFrames, the de facto standard for dataset manipulation?" That's something you'll find out later in this chapter.



**if, elif, else**

**Overview**

such as less than and greater than, and you also know how to combine the boolean results, using boolean operators such as and and or. Things get really interesting when you can actually use these concepts to change how your program behaves. Depending on the outcome of your comparisons, you might want your Python code to behave differently. You can do this with conditional statements in Python: if, else and elif.

**if**

Let's start working in a script, control.py. Suppose you have a variable z, equal to 4. If the value is even, you want to print out: "z is even". This code does the trick. modulo operator 2 will return 0 if z is even. If you run this, Python checks if the condition holds. It's true, so the corresponding code is executed: "z is even" gets printed out. Let's compare this to the general recipe for an if statement. It reads as follows: if condition, execute expression. Notice the colon at the end, and the fact that you simply have to indent the Python code with four spaces (or a tab) to tell Python what to do in the case the condition succeeds.

**if**

To exit the if statement, simply continue with some Python code without indentation, and Python will know that it's not part of the if statement.

It's perfectly possible to have more lines inside the if statement, like this for example. The script now prints out two lines if you run it. If the condition does not pass, the expression is not executed.

You can see this if we change z to be 5 and rerun the code: there's no output. Suppose now that you want to print out "z is odd" in this case. How to do this?

**else**

Well, you can simply use an else statement, like this. If we run it with z equal to 5, the condition is not true, so the expression for the else statement gets printed out. The general recipe looks like this: for the else statement, you don't need to specify a condition. The corresponding expression gets run if the condition of the if statement it belongs to does not hold.

**elif**

02:24 - 02:54

You can think of cases where even more customized behavior is necessary. Say you want different printouts for numbers that are divisible by 2 and by 3. You can throw some elifs in there to get the job done. Take this example. Can you tell what this script will print out if you run it? If z equals 3, the first condition is False, so it goes over to the next condition. This condition, does hold, so the corresponding print statement is executed.

**elif**

Suppose now that z equals 6. Both the if and elif conditions hold in this case. Will two printouts occur? Nope. As soon as Python bumps into a condition that is true, it executes the corresponding code and then leaves the control structure after that. This means the second condition, corresponding to the elif, is never reached so there's no corresponding printout.