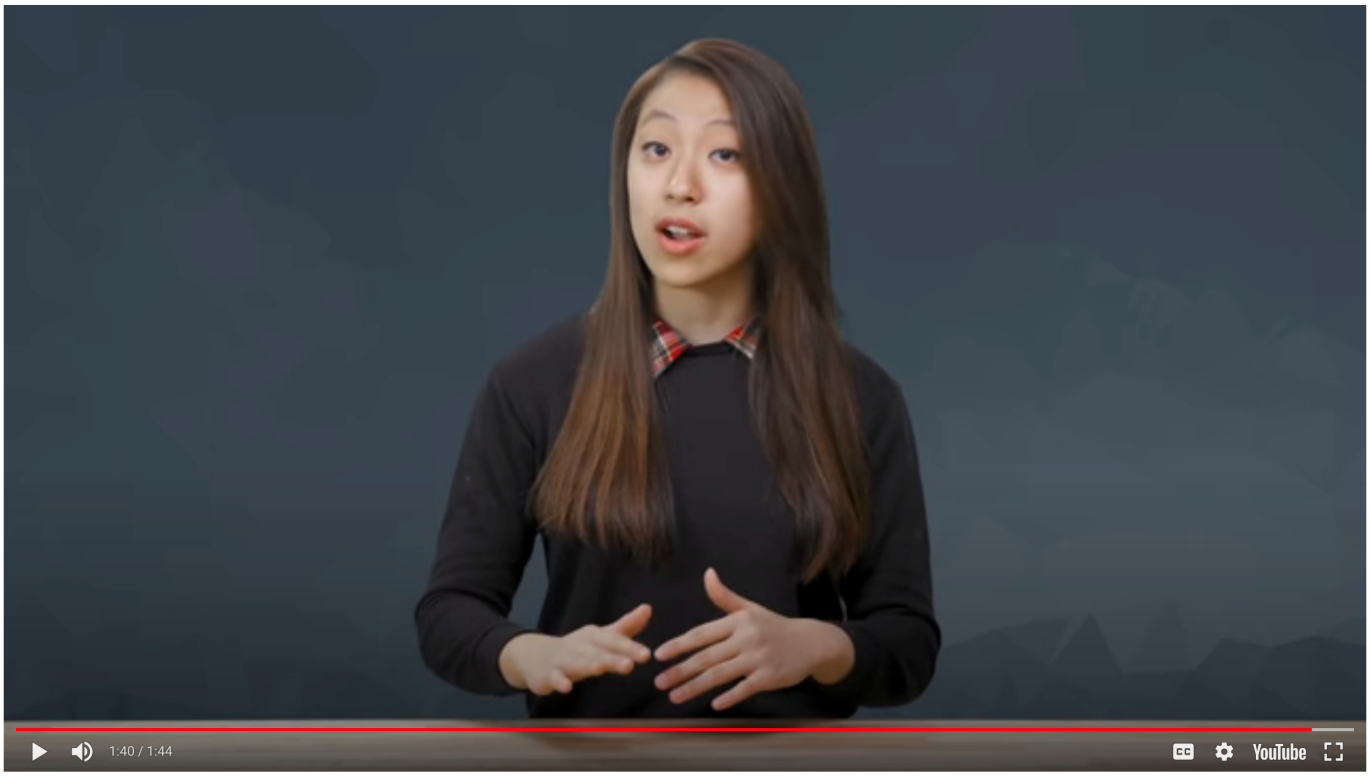
Boolean Expressions for Conditions



Complex Boolean Expressions

`If` statements sometimes use more complicated boolean expressions for their conditions. They may contain multiple comparisons operators, logical operators, and even calculations. Examples:

```
if 18.5 <= weight / height**2 < 25:
    print("BMI is considered 'normal'")

if is_raining and is_sunny:
    print("Is there a rainbow?")

if (not unsubscribed) and (location == "USA" or location == "CAN"):
    print("send email")
```

For really complicated conditions you might need to combine some `and`s, `or`s and `not`s together. Use parentheses if you need to make the combinations clear.

However simple or complex, the condition in an `if` statement must be a boolean expression that evaluates to either True or False and it is this value that decides whether the indented block in an `if` statement executes or not.

Good and Bad Examples

Here are some things to keep in mind while writing boolean expressions for your `if` statements.

1. Don't use `True` or `False` as conditions

```
## Bad example
if True:
    print("This indented code will always get run.")
```

While "True" is a valid boolean expression, it's not useful as a condition since it always evaluates to True, so the indented code will always get run. Similarly, `if False` is not a condition you should use either - the statement following this `if` statement would never be executed.

```
## Another bad example
if is_cold or not is_cold:
    print("This indented code will always get run.")
```

Similarly, it's useless to use any condition that you know will always evaluate to True, like this example above. A boolean variable can only be True or False, so either `is_cold` or `not is_cold` is always True, and the indented code will always be run.

2. Be careful writing expressions that use logical operators

Logical operators `and`, `or` and `not` have specific meanings that aren't quite the same as their meanings in plain English. Make sure your boolean expressions are being evaluated the way you expect them to.

```
## Bad example
if weather == "snow" or "rain":
    print("Wear boots!")
```

This code is valid in Python, but it is not a boolean expression, although it reads like one. The reason is that the expression to the right of the `or` operator, `"rain"`, is not a boolean expression - it's a string! Later we'll discuss what happens when you use non-boolean-type objects in place of booleans.

3. Don't compare a boolean variable with `== True` or `== False`

This comparison isn't necessary, since the boolean variable itself is a boolean expression.

```
## Bad example
if is_cold == True:
    print("The weather is cold!")
```

This is a valid condition, but we can make the code more readable by using the variable itself as the condition instead, as below.

```
## Good example
if is_cold:
    print("The weather is cold!")
```

If you want to check whether a boolean is False, you can use the `not` operator.

Truth Value Testing

If we use a non-boolean object as a condition in an `if` statement in place of the boolean expression, Python will check for its truth value and use that to decide whether or not to run the indented code. By default, the truth value of an object in Python is considered True unless specified as False in the documentation.

Here are most of the built-in objects that are considered False in Python:

- constants defined to be false: `None` and `False`
- zero of any numeric type: `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`
- empty sequences and collections: `""`, `()`, `[]`, `{}`, `set()`, `range(0)`

Example:

```
errors = 3
if errors:
    print("You have {} errors to fix!".format(errors))
else:
    print("No errors to fix!")
```

In this code, `errors` has the truth value True because it's a non-zero number, so the error message is printed. This is a nice, succinct way of writing an `if` statement.

← Previous Next → Give Page Feedback