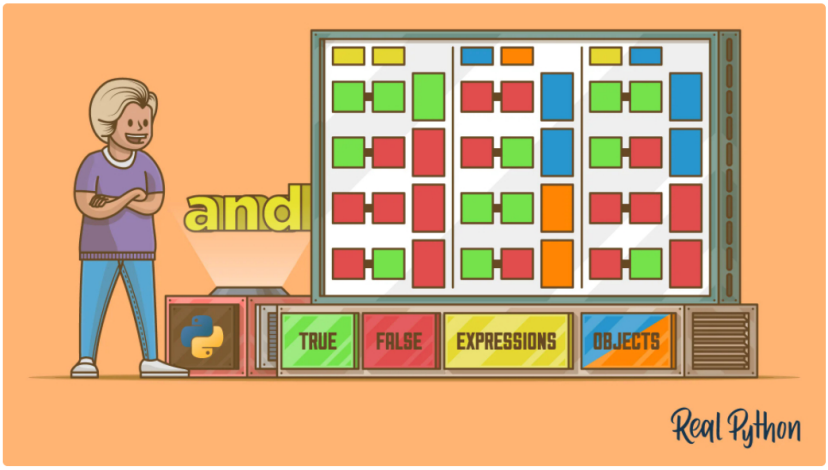


Using the "and" Boolean Operator in Python

by Leodanis Pozo Ramos Sep 20, 2021 3 Comments basics best-practices python

Table of Contents

Python has three Boolean operators, or **logical operators**: `and`, `or`, and `not`. You can use them to check if certain conditions are met before deciding the execution path your programs will follow. In this tutorial, you'll learn about the `and` operator and how to use it in your code.

In this tutorial, you'll learn how to:

- Understand the logic behind Python's **`and` operator**
- Build and understand **Boolean** and **non-Boolean expressions** that use the `and` operator
- Use the `and` operator in **Boolean contexts** to decide the **course of action** of your programs
- Use the `and` operator in **non-Boolean contexts** to make your code more concise

Table of Contents

- Working With Boolean Logic in Python
- Getting Started With Python's and Operator
- Using Python's and Operator in Boolean Contexts
- Using Python's and Operator in Non-Boolean Contexts
- Putting Python's and Operator Into Action
- Conclusion

Mark as Completed

Tweet Share Email

- Use the `and` operator in **non-Boolean contexts** to make your code more concise

You'll also code a few practical examples that will help you understand how to use the `and` operator to approach different problems in a Pythonic way. Even if you don't use all the features of `and`, learning about them will allow you to write better and more accurate code.

Working With Boolean Logic in Python

Back in 1854, George Boole authored The Laws of Thought, which contains what's known as Boolean algebra. This algebra relies on two values: **true** and **false**. It also defines a set of Boolean operations, also known as logical operations, denoted by the generic operators `AND`, `OR`, and `NOT`.

These Boolean values and operators are pretty helpful in programming. For example, you can construct arbitrarily complex Boolean expressions with the operators and determine their resulting truth value as true or false. You can use the **truth value** of Boolean expressions to decide the course of action of your programs.

In Python, the Boolean type `bool` is a subclass of `int` and can take the values `True` or `False`:

Python >>>

```
>>> issubclass(bool, int)
True
>>> help(bool)
Help on class bool in module builtins:

class bool(int)
    ...

>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>

>>> isinstance(True, int)
True
>>> isinstance(False, int)
True

>>> int(True)
1
>>> int(False)
0
```

As you can see in this code, Python implements `bool` as a subclass of `int` with two possible values, `True` and `False`. These values are built-in constants in Python. They're internally implemented as integer numbers with the value `1` for `True` and `0` for `False`. Note that both `True` and `False` must be capitalized.

Along with the `bool` type, Python provides three Boolean operators, or logical operators, that allow you to combine Boolean expressions and objects into more elaborate expressions. Those operators are the following:

Operator	Logical Operation
`and`	Conjunction
`or`	Disjunction
`not`	Negation

With these operators, you can connect several Boolean expressions and objects to build your own expressions. Unlike other languages, Python uses English words to denote

Boolean operators. These words are **keywords** of the language, so you can't use them as identifiers.

In this tutorial, you'll learn about Python's `and` operator. This operator implements the logical `AND` operation. You'll learn how it works and how to use it either in a Boolean or non-Boolean context.

Getting Started With Python's `and` Operator

Python's `and` operator takes two **operands**, which can be Boolean expressions, objects, or a combination. With those operands, the `and` operator builds more elaborate expressions. The operands in an `and` expression are commonly known as **conditions**. If both conditions are true, then the `and` expression returns a true result. Otherwise, it returns a false result:

```
Python                                                    >>>

>>> True and True
True

>>> False and False
False

>>> True and False
False

>>> False and True
False
```

These examples show that an `and` expression only returns `True` when both operands in the expressions are true. Since the `and` operator takes two operands to build an expression, it's a **binary operator**.

The quick examples above show what's known as the `and` operator's truth table:

`operand1`	`operand2`	`operand1 and operand2`
True	True	True
True	False	False
False	False	False
False	True	False

This table summarizes the resulting **truth value** of a Boolean expression like `operand1 and operand2`. The result of the expression depends on the truth values of its operands. It'll be true if both are true. Otherwise, it'll be false. This is the general logic behind the `and` operator. However, this operator can do more than that in Python.

In the following sections, you'll learn how to use `and` for building your own expressions with different types of operands.

Using Python's `and` Operator With Boolean Expressions

You'll typically use logical operators to build **compound Boolean expressions**, which are combinations of variables and values that produce a Boolean value as a result. In other words, Boolean expressions return `True` or `False`.

Comparisons and equality tests are common examples of this type of expression:

```
Python                                                    >>>

>>> 5 == 3 + 2
```

```
True
>>> 5 > 3
True
>>> 5 < 3
False
>>> 5 != 3
True

>>> [5, 3] == [5, 3]
True

>>> "hi" == "hello"
False
```

All these expressions return `True` or `False`, which means they're Boolean expressions. You can combine them using the `and` keyword to create compound expressions that test two—or more–subexpressions at a time:

Python >>>

```
>>> 5 > 3 and 5 == 3 + 2
True

>>> 5 < 3 and 5 == 5
False

>>> 5 == 5 and 5 != 5
False

>>> 5 < 3 and 5 != 5
False
```

Here, when you combine two `True` expressions, you get `True` as a result. Any other combination returns `False`. From these examples, you can conclude that the syntax for creating compound Boolean expressions with the `and` operator is the following:

Python

```
expression1 and expression2
```

If both subexpressions `expression1` and `expression2` evaluate to `True`, then the compound expression is `True`. If at least one subexpression evaluates to `False`, then the result is `False`.

There's no limit to the number of `and` operators you can use when you're building a compound expression. This means that you can combine more than two subexpressions in a single expression using several `and` operators:

Python >>>

```
>>> 5 > 3 and 5 == 3 + 2 and 5 != 3
True

>>> 5 < 3 and 5 == 3 and 5 != 3
False
```

Again, if all the subexpressions evaluate to `True`, then you get `True`. Otherwise, you get `False`. Especially as expressions get longer, you should keep in mind that Python evaluates the expressions sequentially from left to right.

Short-Circuiting the Evaluation

Python's logical operators, such as `and` and `or`, use something called short-circuit evaluation, or **lazy evaluation**. In other words, Python evaluates the operand on the right *only* when it needs to.

To determine the final result of an `and` expression, Python starts by evaluating the left

operand. If it's false, then the whole expression is false. In this situation, there's no need to evaluate the operand on the right. Python already knows the final result.

Having a false left operand automatically makes the whole expression false. It would be a waste of CPU time to evaluate the remaining operand. Python prevents this by short-circuiting the evaluation.

In contrast, the `and` operator evaluates the operand on the right only if the first operand is true. In this case, the final result depends on the right operand's truth value. If it's true, then the whole expression is true. Otherwise, the expression is false.

To demonstrate the short-circuiting feature, take a look at the following examples:

Python >>>

```
>>> def true_func():
...     print("Running true_func()")
...     return True
...

>>> def false_func():
...     print("Running false_func()")
...     return False
...

>>> true_func() and false_func()  # Case 1
Running true_func()
Running false_func()
False

>>> false_func() and true_func()  # Case 2
Running false_func()
False

>>> false_func() and false_func()  # Case 3
Running false_func()
False

>>> true_func() and true_func()  # Case 4
Running true_func()
Running true_func()
True
```

Here's how this code works:

- **Case 1**: Python evaluates `true_func()`, which returns `True`. To determine the final result, Python evaluates `false_func()` and gets `False`. You can confirm this by seeing both functions' output.
- **Case 2**: Python evaluates `false_func()`, which returns `False`. Python already knows that the final result is `False`, so it doesn't evaluate `true_func()`.
- **Case 3**: Python runs `false_func()` and gets `False` as a result. It doesn't need to evaluate the repeated function a second time.
- **Case 4**: Python evaluates `true_func()` and gets `True` as a result. It then evaluates the function again. Since both operands evaluate to `True`, the final result is `True`.

Python processes Boolean expressions from left to right. It stops when it no longer needs to evaluate any further operands or subexpressions to determine the final outcome. To sum up this concept, you should remember that if the left operand in an `and` expression is false, then the right operand won't be evaluated.

Short-circuit evaluation can have a significant impact on your code's performance. To take advantage of that, consider the following tips when you're building `and` expressions:

- Place time-consuming expressions on the right of the `and` keyword. This way, the costly expression won't run if the short-circuit rule takes effect.
- Place the expression that is more likely to be false on the left of the `and` keyword. This way, it's more likely that Python can determine if the whole expression is false by evaluating the left operand only.

Sometimes you may want to avoid lazy evaluation in a specific Boolean expression. You can do so by using the bitwise operators (`&`, `|`, `~`). These operators also work in Boolean

do so by using the bitwise operators (&, |, ^). These operators also work in Boolean expressions, but they evaluate the operands eagerly:

```
Python >>>
>>> def true_func():
...     print("Running true_func()")
...     return True
...

>>> def false_func():
...     print("Running false_func()")
...     return False
...

>>> # Use logical and
>>> false_func() and true_func()
Running false_func()
False

>>> # Use bitwise and
>>> false_func() & true_func()
Running false_func()
Running true_func()
False
```

In the first expression, the `and` operator works lazily, as expected. It evaluates the first function, and since the result is false, it doesn't evaluate the second function. In the second expression, however, the bitwise AND operator (`&`) calls both functions eagerly even though the first function returns `False`. Note that in both cases, the final result is `False`.

Even though this trick works, it's generally discouraged. You should use bitwise operators to manipulate bits, and Boolean operators to work with Boolean values and expressions. For a deeper dive into bitwise operators, check out Bitwise Operators in Python.

Using Python's `and` Operator With Common Objects

You can use the `and` operator to combine two Python objects in a single expression. In that situation, Python internally uses `bool()` to determine the truth value of the operands. As a result, you get a specific object rather than a Boolean value. You only get `True` or `False` if a given operand explicitly evaluates to `True` or `False`:

```
Python >>>
>>> 2 and 3
3

>>> 5 and 0.0
0.0

>>> [] and 3
[]

>>> 0 and {}
0

>>> False and ""
False
```

In these examples, the `and` expression returns the operand on the left if it evaluates to `False`. Otherwise, it returns the operand on the right. To produce these results, the `and` operator uses Python's internal rules to determine an object's truth value. The Python documentation states these rules like this:

> By default, an object is considered true unless its class defines either a `__bool__()` method that returns `False` or a `__len__()` method that returns zero, when called with the object. Here are most of the built-in objects considered false:
>
> - 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)`

> (Source)

With these rules in mind, look again at the code above. In the first example, the integer number `2` is true (nonzero), so `and` returns the right operand, `3`. In the second example, `5` is true, so `and` returns the right operand even though it evaluates to `False`.

The next example uses an empty list (`[]`) as the left operand. Since empty lists evaluate to false, the `and` expression returns the empty list. The only case where you get `True` or `False` is the one where you use a Boolean object explicitly in the expression.

> **Note:** If you need to get `True` or `False` from an `and` expression involving common objects rather than Boolean expressions, then you can use `bool()`. This built-in function explicitly returns `True` or `False` depending on the truth value of the specific object you provide as an argument.

Here's how you can summarize the behavior of the `and` operator when you use it with common Python objects instead of Boolean expressions. Note that Python uses the truth value of each object to determine the final result:

`object1`	`object2`	`object1 and object2`
False	False	`object1`
False	True	`object1`
True	True	`object2`
True	False	`object2`

In general, if the operands in an `and` expression are objects instead of Boolean expressions, then the operator returns the object on the left if it evaluates to `False`. Otherwise, it returns the object on the right, even when it evaluates to `False`.

Mixing Boolean Expressions and Objects

You can also combine Boolean expressions and common Python objects within an `and` expression. In that situation, the `and` expression still returns the left operand if it's false, or else it returns the right operand. The returned value could be `True`, `False`, or a regular object, depending on which part of the expression provides that result:

Python >>>

```
>>> 2 < 4 and 2
2
>>> 2 and 2 < 4
True

>>> 2 < 4 and []
[]
>>> [] and 2 < 4
[]

>>> 5 > 10 and {}
False
>>> {} and 5 > 10
{}

>>> 5 > 10 and 4
False
>>> 4 and 5 > 10
False
```

These examples use a combination of Boolean expressions and common objects. In each pair of examples, you see that you can get either a non-Boolean object or a Boolean value, `True` or `False`. The result will depend on which part of the expression provides the final result.

Here's a table that summarizes the behavior of the `and` operator when you combine Boolean expressions and common Python objects:

expression	object	expression and object
`True`	True	`object`
`True`	False	`object`
`False`	False	`False`
`False`	True	`False`

To find out what's returned, Python evaluates the Boolean expression on the left to get its Boolean value (`True` or `False`). Then Python uses its internal rules to determine the truth value of the object on the right.

As an exercise to test your understanding, you could try to rewrite this table by swapping the order of the operands in the third column to `object and expression`. Try to predict what will be returned in each row.

Combining Python Logical Operators

As you've seen earlier in this tutorial, Python provides two additional logical operators: the `or` operator and the `not` operator. You can use them along with the `and` operator to create more complex compound expressions. If you want to make accurate and clear expressions with multiple logical operators, then you need to consider the precedence of each operator. In other words, you need to consider the order in which Python executes them.

Python's logical operators have low precedence when compared with other operators. However, sometimes it's healthy to use a pair of parentheses (`()`) to ensure a consistent and readable result:

```
Python                                                    >>>

>>> 5 or 2 and 2 > 1
5

>>> (5 or 3) and 2 > 1
True
```

These examples combine the `or` operator and the `and` operator in a compound expression. Just like the `and` operator, the `or` operator uses short-circuit evaluation. Unlike `and`, however, the `or` operator stops once it finds a true operand. You can see this in the first example. Since `5` is true, the `or` subexpression immediately returns `5` without evaluating the rest of the expression.

In contrast, if you enclose the `or` subexpression in a pair of parentheses, then it works as a single true operand, and `2 > 1` gets evaluated as well. The final result is `True`.

The takeaway is that if you're using multiple logical operators in a single expression, then you should consider using parentheses to make your intentions clear. This trick will also help you get the correct logical result.

Using Python's `and` Operator in Boolean Contexts

Like all of Python's Boolean operators, the `and` operator is especially useful in **Boolean contexts**. Boolean contexts are where you'll find most of the real-world use cases of Boolean operators.

Two main structures define Boolean contexts in Pvthon:

1. `if` statements let you perform **conditional execution** and take different courses of action based on the result of some initial conditions.
2. `while` loops let you perform **conditional iteration** and run repetitive tasks while a given condition is true.

These two structures are part of what you'd call control flow statements. They help you decide your programs' execution path.

You can use Python's `and` operator to construct compound Boolean expressions in both `if` statements and `while` loops.

`if` Statements

Boolean expressions are commonly known as **conditions** because they typically imply the need for meeting a given requirement. They're pretty useful in the context of conditional statements. In Python, this type of statement starts with the `if` keyword and continues with a condition. A conditional statement can additionally include `elif` and `else` clauses.

Python conditional statements follow the logic of conditionals in English grammar. If the condition is true, then the `if` code block executes. Otherwise, the execution jumps to a different code block:

Python >>>

```
>>> a = -8

>>> if a < 0:
...     print("a is negative")
... elif a > 0:
...     print("a is positive")
... else:
...     print("a is equal to 0")
...
a is negative
```

Since `a` holds a negative number, the condition `a < 0` is true. The `if` code block runs, and you get the message `a is negative` printed on your screen. If you change the value of `a` to a positive number, however, then the `elif` block runs and Python prints `a is positive`. Finally, if you set `a` to zero, then the `else` code block executes. Go ahead and play with `a` to see what happens!

Now say you want to make sure that two conditions are met—meaning that they're both true—before running a certain piece of code. To try this out, suppose you need to get the age of a user running your script, process that information, and display to the user their current life stage.

Fire up your favorite code editor or IDE and create the following script:

Python

```
# age.py

age = int(input("Enter your age: "))

if age >= 0 and age <= 9:
    print("You are a child!")
elif age > 9 and age <= 18:
    print("You are an adolescent!")
elif age > 18 and age <= 65:
    print("You are an adult!")
elif age > 65:
    print("Golden ages!")
```

Here, you get the user's age using `input()` and then convert it to an integer number with

`int()`. The `if` clause checks if `age` is greater than or equal to `0`. In the same clause, it checks if `age` is less than or equal to `9`. To do this, you build an `and` compound Boolean expression.

The three `elif` clauses check other intervals to determine the life stage associated with the user's age.

If you run this script from your command line, then you get something like this:

Shell

```
$ python age.py
Enter your age: 25
You are an adult!
```

Depending on the age you enter at the command line, the script takes one course of action or another. In this specific example, you provide an age of 25 years and get the message `You are an adult!` printed to your screen.

`while` Loops

The `while` loop is the second construct that can use `and` expressions to control a program's execution flow. By using the `and` operator in the `while` statement header, you can test several conditions and repeat the loop's code block for as long as all conditions are met.

Say you're prototyping a control system for a manufacturer. The system has a critical mechanism that should work with a pressure of 500 psi or lower. If the pressure goes over 500 psi while staying under 700 psi, then the system has to run a given series of standard safety actions. For pressures greater than 700 psi, there are a whole new set of safety actions that the system must run.

To approach this problem, you can use a `while` loop with an `and` expression. Here's a script that simulates a possible solution:

Python

```
1  # pressure.py
2
3  from time import sleep
4  from random import randint
5
6  def control_pressure():
7      pressure = measure_pressure()
8      while True:
9          if pressure <= 500:
10             break
11
12         while pressure > 500 and pressure <= 700:
13             run_standard_safeties()
14             pressure = measure_pressure()
15
16         while pressure > 700:
17             run_critical_safeties()
18             pressure = measure_pressure()
19
20     print("Wow! The system is safe...")
21
22 def measure_pressure():
23     pressure = randint(490, 800)
24     print(f"psi={pressure}", end="; ")
25     return pressure
26
27 def run_standard_safeties():
28     print("Running standard safeties...")
29     sleep(0.2)
30
31 def run_critical_safeties():
32     print("Running critical safeties...")
33     sleep(0.7)
34
35 if __name__ == "__main__":
36     control_pressure()
```

Inside `control_pressure()`, you create an infinite `while` loop on line 8. If the system is stable

and the pressure is below 500 psi, the conditional statement breaks out of the loop and the program finishes.

On line 12, the first nested `while` loop runs the standard safety actions while the system pressure stays between 500 psi and 700 psi. In each iteration, the loop gets a new pressure measurement to test the condition again in the next iteration. If the pressure grows beyond 700 psi, then the second loop on line 16 runs the critical safety actions.

Note: The implementation of `control_pressure()` in the example above is intended to show how the `and` operator can work in the context of a `while` loop.

However, this isn't the most efficient implementation you can write. You can refactor `control_pressure()` to use a single loop without using `and`:

Python

```
def control_pressure():
    while True:
        pressure = measure_pressure()
        if pressure > 700:
            run_critical_safeties()
        elif 500 < pressure <= 700:
            run_standard_safeties()
        elif pressure <= 500:
            break
    print("Wow! The system is safe...")
```

In this alternative implementation, instead of using `and`, you use the chained expression `500 < pressure <= 700`, which does the same as `pressure > 500 and pressure <= 700` but is cleaner and more Pythonic. Another advantage is that you call `measure_pressure()` only once, which ends up being more efficient.

To run this script, open up your command line and enter the following command:

Shell

```
$ python pressure.py
psi=756; Running critical safeties...
psi=574; Running standard safeties...
psi=723; Running critical safeties...
psi=552; Running standard safeties...
psi=500; Wow! The system is safe...
```

The output on your screen should be a little different from this sample output, but you can still get an idea of how the application works.

Using Python's `and` Operator in Non-Boolean Contexts

The fact that `and` can return objects besides just `True` and `False` is an interesting feature. For example, this feature allows you to use the `and` operator for **conditional execution**. Say you need to update a `flag` variable if the first item in a given list is equal to a certain expected value. For this situation, you can use a conditional statement:

Python >>>

```
>>> a_list = ["expected value", "other value"]
>>> flag = False

>>> if len(a_list) > 0 and a_list[0] == "expected value":
...     flag = True
...

>>> flag
```

```
True
```

Here, the conditional checks if the list has at least one item. If so, it checks if the first item in the list is equal to the `"expected value"` string. If both checks pass, then `flag` changes to `True`. You can simplify this code by taking advantage of the `and` operator:

Python >>>

```
>>> a_list = ["expected value", "other value"]
>>> flag = False

>>> flag = len(a_list) > 0 and a_list[0] == "expected value"

>>> flag
True
```

In this example, the highlighted line does all the work. It checks both conditions and makes the corresponding assignment in one go. This expression takes the `and` operator out of the `if` statement you used in the previous example, which means that you're not working in a Boolean context any longer.

The code in the example above is more concise than the equivalent conditional statement you saw before, but it's less readable. To properly understand this expression, you'd need to be aware of how the `and` operator works internally.

Putting Python's `and` Operator Into Action

So far, you've learned how to use Python's `and` operator for creating compound Boolean expressions and non-Boolean expressions. You've also learned how to use this logical operator in Boolean contexts like `if` statements and `while` loops.

In this section, you'll build a few practical examples that'll help you decide when to use the `and` operator. With these examples, you'll learn how to take advantage of `and` for writing better and more Pythonic code.

Flattening Nested `if` Statements

One principle from the Zen of Python states that "Flat is better than nested." For example, while code that has two levels of nested `if` statements is normal and totally okay, your code really starts to look messy and complicated when you have more than two levels of nesting.

Say you need to test if a given number is positive. Then, once you confirm that it's positive, you need to check if the number is lower than a given positive value. If it is, you can proceed with a specific calculation using the number at hand:

Python >>>

```
>>> number = 7

>>> if number > 0:
...     if number < 10:
...         # Do some calculation with number...
...         print("Calculation done!")
...
Calculation done!
```

Cool! These two nested `if` statements solve your problem. You first check if the number is positive and then check if it's lower than `10`. In this small example, the call to `print()` is a placeholder for your specific calculation, which runs only if both conditions are true.

Even though the code works, it'd be nice to make it more Pythonic by removing the nested `if`. How can you do that? Well, you can use the `and` operator to combine both conditions in a single compound condition:

Python >>>

```
>>> number = 7

>>> if number > 0 and number < 10:
...     # Do some calculation with number...
```

```
...     print("Calculation done!")
...
Calculation done!
```

Logical operators like the `and` operator often provide an effective way to improve your code by removing nested conditional statements. Take advantage of them whenever possible.

In this specific example, you use `and` to create a compound expression that checks if a number is in a given range or interval. Python provides an even better way to perform this check by chaining expressions. For example, you can write the condition above as `0 < number < 10`. That's a topic for the following section.

Checking Numeric Ranges

With a close look at the example in the section below, you can conclude that Python's `and` operator is a convenient tool for checking if a specific numeric value is inside a given interval or range. For example, the following expressions check if a number `x` is between `0` and `10`, both inclusive:

Python >>>

```
>>> x = 5
>>> x >= 0 and x <= 10
True

>>> x = 20
>>> x >= 0 and x <= 10
False
```

In the first expression, the `and` operator first checks if `x` is greater than or equal to `0`. Since the condition is true, the `and` operator checks if `x` is lower than or equal to `10`. The final result is true because the second condition is also true. This means that the number is within the desired interval.

In the second example, the first condition is true, but the second is false. The general result is false, which means the number isn't in the target interval.

You can enclose this logic in a function and make it reusable:

Python >>>

```
>>> def is_between(number, start=0, end=10):
...     return number >= start and number <= end
...

>>> is_between(5)
True
>>> is_between(20)
False

>>> is_between(20, 10, 40)
True
```

In this example, `is_between()` takes `number` as an argument. It also takes `start` and `end`, which define the target interval. Note that these arguments have default argument values, which means they're optional arguments.

Your `is_between()` function returns the result of evaluating an `and` expression that checks if `number` is between `start` and `end`, both inclusive.

> **Note:** Unintentionally writing `and` expressions that always return `False` is a common mistake. Suppose you want to write an expression that excludes values between `0` and `10` from a given computation.
>
> To achieve this result, you start with two Boolean expressions:
>
> 1. `number < 0`
> 2. `number > 10`
>
> With these two expressions as a starting point, you think of using `and` to combine them

in a single compound expression. However, no number is lower than `0` and greater than `10` at the same time, so you end up with an always-false condition:

```
Python >>>
>>> for number in range(-100, 100):
...     included = number < 0 and number > 10
...     print(f"Is {number} included?", included)
...
Is -100 included? False
Is -99 included? False

...

Is 0 included? False
Is 1 included? False

...

Is 98 included? False
Is 99 included? False
```

In this case, `and` is the wrong logical operator to approach the problem at hand. You should use the `or` operator instead. Go ahead and give it a try!

Even though using the `and` operator allows you to check gracefully if a number is within a given interval, there's a more Pythonic technique to approach the same problem. In mathematics, you can write $0 < x < 10$ to denote that x is between 0 and 10.

In most programming languages, this expression doesn't make sense. In Python, however, the expression works like a charm:

```
Python >>>
>>> x = 5
>>> 0 < x < 10
True

>>> x = 20
>>> 0 < x < 10
False
```

In a different programming language, this expression would start by evaluating `0 < x`, which is true. The next step would be to compare the true Boolean with `10`, which doesn't make much sense, so the expression fails. In Python, something different happens.

Python internally rewrites this type of expression to an equivalent `and` expression, such as `x > 0 and x < 10`. It then performs the actual evaluation. That's why you get the correct result in the example above.

Just like you can chain several subexpressions with multiple `and` operators, you can also chain them without explicitly using any `and` operators:

```
Python >>>
>>> x = 5
>>> y = 15

>>> 0 < x < 10 < y < 20

>>> # Equivalent and expression
>>> 0 < x and x < 10 and 10 < y and y < 20
True
```

You can also use this Python trick to check if several values are equal:

```
Python >>>
>>> x = 10
>>> y = 10
>>> z = 10

>>> x == y == z
```

```
True

>>> # Equivalent and expression
>>> x == y and y == z
True
```

Chained comparison expressions are a nice feature, and you can write them in various ways. However, you should be careful. In some cases, the final expression can be challenging to read and understand, especially for programmers coming from languages in which this feature isn't available.

Chaining Function Calls Conditionally

If you've ever worked with Bash on a Unix system, then you probably know about the `command1 && command2` construct. This is a handy technique that allows you to run several commands in a chain. Each command runs if and only if the previous command was successful:

Shell

```
$ cd /not_a_dir && echo "Success"
bash: cd: /not_a_dir: No such file or directory

$ cd /home && echo "Success"
Success
```

These examples use Bash's short-circuit AND operator (`&&`) to make the execution of the `echo` command dependent on the success of the `cd` command.

Since Python's `and` also implements the idea of lazy evaluation, you can use it to emulate this Bash trick. For example, you can chain a series of function calls in a single `and` expression like the following:

Python

```
func1() and func2() and func3() ... and funcN()
```

In this case, Python calls `func1()`. If the function's return value evaluates to a true value, then Python calls `func2()`, and so on. If one of the functions returns a false value, then Python won't call the rest of the functions.

Here's an example that uses some `pathlib` functions to manipulate a text file:

Python >>>

```
>>> from pathlib import Path
>>> file = Path("hello.txt")
>>> file.touch()

>>> # Use a regular if statement
>>> if file.exists():
...     file.write_text("Hello!")
...     file.read_text()
...
6
'Hello!'

>>> # Use an and expression
>>> file.exists() and file.write_text("Hello!") and file.read_text()
'Hello!'
```

Nice! In a single line of code, you run three functions conditionally without the need for an `if` statement. In this specific example, the only visible difference is that `.write_text()` returns the number of bytes it wrote to the file. The interactive shell automatically displays that value to the screen. Keep in mind that this difference isn't visible when you run the code

as a script.

Conclusion

Python's `and` operator allows you to construct compound Boolean expressions that you can use to decide the course of action of your programs. You can use the `and` operator to solve several problems both in Boolean or non-Boolean contexts. Learning about how to use the `and` operator properly can help you write more Pythonic code.

In this tutorial, you learned how to:

- Work with Python's **`and` operator**
- Build **Boolean** and **non-Boolean expressions** with Python's `and` operator
- Decide the **course of action** of your programs using the `and` operator in Boolean contexts
- Make your code more concise using the `and` operator in **non-Boolean contexts**

Going through the practical examples in this tutorial can help you get a general idea of how to use the `and` operator to make decisions in your Python code.

About **Leodanis Pozo Ramos**

Leodanis is an industrial engineer who loves Python and software development. He's a self-taught Python developer with 6+ years of experience. He's an avid technical writer with a growing number of articles published on Real Python and other sites.

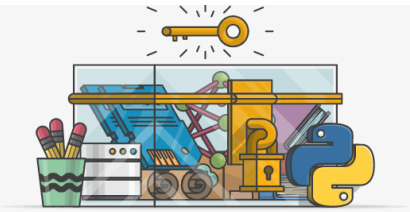
» More about Leodanis

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

 Aldren

 Bartosz

 Sadie



Join us and get access to hundreds of tutorials, hands-on video courses, and a community of expert Pythonistas:

Level Up Your Python Skills »

What Do You Think?

Tweet Share Email

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.