python workshop : Operators

Table Of Contents

Hello 👋

Welcome to another (not so boring 😆) workshop day in Python. Lets move forward 👇

Comparison Operators

Operator	Example	Meaning	Result
==	a == b	Equal to	True if the value of a is equal to the value of b False otherwise
!=	a != b	Not equal to	True if a is not equal to b False otherwise
<	a < b	Less than	True if a is less than b
False otherwise			
<=	a <= b	Less than or equal to	True if a is less than or equal to b False otherwise
>	a > b	Greater than	True if a is greater than b
False otherwise			
>=	a >= b	Greater than or equal to	True if a is greater than or equal to b
False otherwise			

Here are examples of the comparison operators in use:

```
>>> a = 10
>>> b = 20
>>> a == b
False
>>> a != b
True
>>> a <= b
True
>>> a >= b
False

>>> a = 30
>>> b = 30
>>> a == b
True
>>> a <= b
True
>>> a >= b
True
```

Comparison operators are typically used in Boolean contexts like conditional and loop statements to direct program flow,as you will see later.

Equality Comparison on Floating-Point Values

Floating-point numbers that the value stored internally for a float object may not be precisely what you'd think it would be. For that reason, it is poor practice to compare floating-point values for exact equality. Consider this example:

```
>>> x = 1.1 + 2.2
>>> x == 3.3
False
```

😞Yikes! The internal representations of the addition operands are not exactly equal to 1.1 and 2.2, so you cannot rely on x to compare exactly to 3.3.

The preferred way to determine whether two floating-point values are "equal" is to compute whether they are close to one another, given some tolerance. Take a look at this example:

```
>>> tolerance = 0.00001
>>> x = 1.1 + 2.2
>>> abs(x - 3.3) < tolerance
True
```

abs() returns absolute value. If the absolute value of the difference between the two numbers is less than the specified tolerance, they are close enough to one another to be considered equal.

Logical Operators

The logical operators **not**, **or**, and **and** modify and join together expressions evaluated in Boolean context to create more complex conditions.

Logical Expressions Involving Boolean Operands

As you have seen, some objects and expressions in Python actually are of Boolean type. That is, they are equal to one of the Python objects **True** or **False**. Consider these examples:

```
>>> x = 5
>>> x < 10
True
>>> type(x < 10)
<class 'bool'>

>>> t = x > 10
>>> t
False
>>> type(t)
<class 'bool'>

>>> callable(x)
False
>>> type(callable(x))
<class 'bool'>

>>> t = callable(len)
>>> t
True
>>> type(t)
<class 'bool'>
```

In the examples above, x < 10, callable(x), and t are all Boolean objects or expressions.

Interpretation of logical expressions involving not, or, and and is straightforward when the operands are Boolean:

Operator	Example	Meaning

Operator	Example	Meaning
not	not x	True if x is False

False if x is True
(Logically reverses the sense of x)|
|or |x or y |True |if either x or y is True
False otherwise|
|and| x and y| True if both x and y are True
False otherwise|

Take a look at how they work in practice below.

“not” and Boolean Operands

```
x = 5
not x < 10
False
not callable(x)
True
```

Operand	Value	Logical Expression	Value
x < 10	True	not x < 10	False
callable(x)	False	not callable(x)	True

“or” and Boolean Operands

```
x = 5
x < 10 or callable(x)
True
x < 0 or callable(x)
False
```

Operand	Value	operand	value	Logical Expression	Value
x < 10	True	callable(x)	False	x < 10 or callable(x)	True
x < 0	False	callable(x)	False	x < 0 or callable(x)	False

“and” and Boolean Operands

```
x = 5
x < 10 and callable(x)
False
x < 10 and callable(len)
True
```

Operand	Value	Operand	Value	Logical Expression	Value
x < 10	True	callable(x)	False	x < 10 and callable(x)	False
x < 10	True	callable(len)	True	x < 10 or callable(len)	True

Evaluation of Non-Boolean Values in Boolean Context

Many objects and expressions are not equal to True or False. Nonetheless, they may still be evaluated in Boolean context and determined to be “truthy” or “falsy.”

In Python, it is well-defined. All the following are considered false when evaluated in Boolean context:

The Boolean value False

- Any value that is numerically zero (0, 0.0, 0.0+0.0j)
- An empty string
- An object of a built-in composite data type which is empty (see below)
- The special value denoted by the Python keyword `None`
- Virtually any other object built into Python is regarded as true.

You can determine the “truthiness” of an object or expression with the built-in bool() function. bool() returns True if its argument is truthy and False if it is falsy.

Numeric Value

- A zero value is false.
- A non-zero value is true.

```
>>> print(bool(0), bool(0.0), bool(0.0+0j))
False False False

>>> print(bool(-3), bool(3.14159), bool(1.0+1j))
True True True
```

String

An empty string is false.
A non-empty string is true.

```
>>> print(bool(''), bool(""), bool("""""""))
False False False

>>> print(bool('foo'), bool(" "), bool(''' '''))
True True True
```

Built-In Composite Data Object

Python provides built-in composite data types called list, tuple, dict, and set. These are “container” types that contain other objects. An object of one of these types is considered false if it is empty and true if it is non-empty.

The examples below demonstrate this for the list type. (Lists are defined in Python with square brackets.)

For more information on the list, tuple, dict, and set types, see the upcoming tutorials.

```
>>> type([])
<class 'list'>
>>> bool([])
False

>>> type([1, 2, 3])
<class 'list'>
>>> bool([1, 2, 3])
True
```

The “None” Keyword

None is always false:

```
>>> bool(None)
False
```

Logical Expressions Involving Non-Boolean Operands

Non-Boolean values can also be modified and joined by not, or and, and. The result depends on the “truthiness” of the operands.

“not” and Non-Boolean Operands

Here is what happens for a non-Boolean value x:

If x is	not x is
“truthy”	False
“falsy”	True

Here are some concrete examples:

```
>>> x = 3
>>> bool(x)
True
>>> not x
False

>>> x = 0.0
>>> bool(x)
False
>>> not x
True
```

“or” and Non-Boolean Operands

This is what happens for two non-Boolean values x and y:

If x is	x or y is
truthy	x
falsy	y

Note that in this case, the expression x or y does not evaluate to either True or False, but instead to one of either x or y:

```
>>> x = 3
>>> y = 4
>>> x or y
3

>>> x = 0.0
>>> y = 4.4
>>> x or y
4.4
```

Even so, it is still the case that the expression x or y will be truthy if either x or y is truthy, and falsy if both x and y are falsy.

“and” and Non-Boolean Operands

Here’s what you’ll get for two non-Boolean values x and y:

If x is	x and y is
“truthy”	y
“falsy”	x

```
>>> x = 3
>>> y = 4
>>> x and y
4

>>> x = 0.0
>>> y = 4.4
>>> x and y
0.0
```

As with or, the expression x and y does not evaluate to either True or False, but instead to one of either x or y. x and y will be truthy if both x and y are truthy, and falsy otherwise.

Avoiding an Exception

Suppose you have defined two variables a and b, and you want to know whether (b / a) > 0:

```
>>> a = 3
>>> b = 1
>>> (b / a) > 0
True
```

But you need to account for the possibility that a might be 0, in which case the interpreter will raise an exception:

```
>>> a = 0
>>> b = 1
>>> (b / a) > 0
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    (b / a) > 0
ZeroDivisionError: division by zero
```

You can avoid an error with an expression like this:

```
>>> a = 0
>>> b = 1
>>> a != 0 and (b / a) > 0
False
```

When a is 0, a != 0 is false. Short-circuit evaluation ensures that evaluation stops at that point. (b / a) is not evaluated, and no error is raised.

If fact, you can be even more concise than that. When a is 0, the expression a by itself is falsy. There is no need for the explicit comparison a != 0:

```
>>> a = 0
>>> b = 1
>>> a and (b / a) > 0
0
```

Selecting a Default Value

Another idiom involves selecting a default value when a specified value is zero or empty. For example, suppose you want to assign a variable s to the value contained in another variable called string. But if string is empty, you want to supply a default value.

Here is a concise way of expressing this using short-circuit evaluation:

```
s = string or '<default_value>'
```

If string is non-empty, it is truthy, and the expression string or '<default_value>' will be true at that point. Evaluation stops, and the value of string is returned and assigned to s:

```
>>> string = 'foo bar'
>>> s = string or '<default_value>'
>>> s
'foo bar'
```

On the other hand, if string is an empty string, it is falsy. Evaluation of string or '<default_value>' continues to the next operand, '<default_value>', which is returned and assigned to s:

```
>>> string = ''
>>> s = string or '<default_value>'
>>> s
'<default_value>'
```

Chained Comparisons

Comparison operators can be chained together to arbitrary length. For example, the following expressions are nearly equivalent:

```
x < y <= z
x < y and y <= z
```

They will both evaluate to the same Boolean value. The subtle difference between the two is that in the chained comparison x < y <= z, y is evaluated only once. The longer expression x < y and y <= z will cause y to be evaluated twice.

More generally, if op1, op2, …, opn are comparison operators, then the following have the same Boolean value:

x1 op1 x2 op2 x3 … xn-1 opn xn

x1 op1 x2 and x2 op2 x3 and … xn-1 opn xn

In the former case, each xi is only evaluated once. In the latter case, each will be evaluated twice except the first and last, unless short-circuit evaluation causes premature termination.

Bitwise Operators

Bitwise operators treat operands as sequences of binary digits and operate on them bit by bit. The following operators are supported:

Operator	Example	Meaning	Result
&	a & b	bitwise AND Each bit position in the result is the logical AND of the bits in the corresponding position of the operands.	(1 if both are 1, otherwise 0.)
|	a | b	bitwise OR Each bit position in the result is the logical OR of the bits in the corresponding position of the operands.	(1 if either is 1, otherwise 0.)
~	~a	bitwise negation Each bit position in the result is the logical negation of the bit in the corresponding position of the operand.	(1 if 0, 0 if 1.)

Operator	Example	Meaning	Result
^	a ^ b	bitwise XOR (exclusive OR) Each bit position in the result is the logical XOR of the bits in the corresponding position of the operands.	(1 if the bits in the operands are different, 0 if they are the same.)
>>	a >> n	Shift right n places Each bit is shifted right n places	
<<	a << n	Shift left n places Each bit is shifted left n places.	

Here are some examples:

```
>>> '0b{:04b}'.format(0b1100 & 0b1010)
'0b1000'
>>> '0b{:04b}'.format(0b1100 | 0b1010)
'0b1110'
>>> '0b{:04b}'.format(0b1100 ^ 0b1010)
'0b0110'
>>> '0b{:04b}'.format(0b1100 >> 2)
'0b0011'
>>> '0b{:04b}'.format(0b0011 << 2)
'0b1100'
```

Note: The purpose of the '0b{:04b}'.format() is to format the numeric output of the bitwise operations, to make them easier to read. You will see the format() method in much more detail later. For now, just pay attention to the operands of the bitwise operations, and the results.

Identity Operators

Python provides two operators, **is** and **is not**, that determine whether the given operands have the same identity—that is, refer to the same object. This **is not** the same thing as equality, which means the two operands refer to objects that contain the same data but are not necessarily the same object.

Here is an example of two object that are equal but not identical:

```
>>> x = 1001
>>> y = 1000 + 1
>>> print(x, y)
1001 1001

>>> x == y
True
>>> x is y
False
```

Here, x and y both refer to objects whose value is 1001. They are equal. But they do not reference the same object, as you can verify:

```
>>> id(x)
60307920
>>> id(y)
60307936
```

x and y do not have the same identity, and x is y returns False.

You saw previously that when you make an assignment like x = y, Python merely creates a second reference to the same object, and that you could confirm that fact with the id() function. You can also confirm it using the is operator:

```
>>> a = 'I am a string'
>>> b = a
>>> id(a)
55993992
>>> id(b)
55993992

>>> a is b
True
>>> a == b
True
```

In this case, since a and b reference the same object, it stands to reason that a and b would be equal as well.

Unsurprisingly, the opposite of is is is not:

```
>>> x = 10
>>> y = 20
>>> x is not y
True
```

Variable, literal, operators and expressions.

Debugging a Python program.