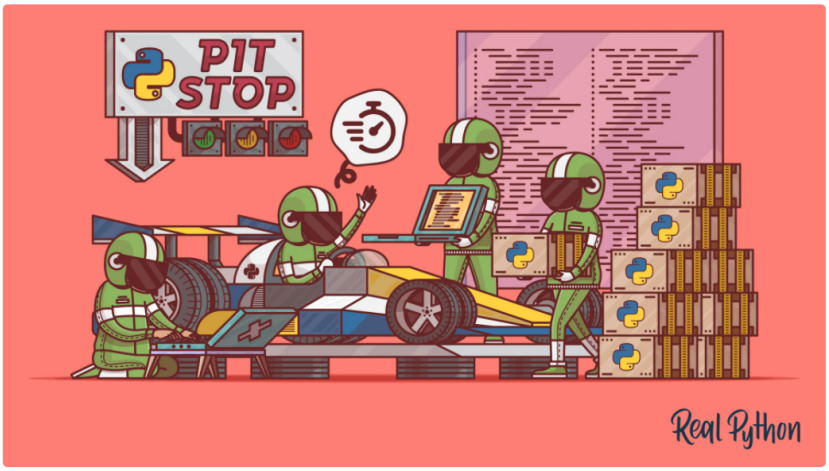

Python '!=' Is Not 'is not': Comparing Objects in Python

by Joska de Langen Jan 29, 2020 7 Comments best-practices intermediate python

Mark as Completed

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Comparing Python Objects the Right Way: "is" vs "=="**

There's a subtle difference between the Python identity operator (`is`) and the equality operator (`==`). Your code can run fine when you use the Python `is` operator to compare numbers, until it suddenly doesn't. You might have heard somewhere that the Python `is` operator is faster than the `==` operator, or you may feel that it looks more Pythonic. However, it's crucial to keep in mind that these operators don't behave quite the same.

The `==` operator compares the value or **equality** of two objects, whereas the Python `is` operator checks whether two variables point to the same object in memory. In the vast majority of cases, this means you should use the equality operators `==` and `!=`, except when you're comparing to `None`.

In this tutorial, you'll learn:

- What the difference is between **object equality and identity**
- When to use equality and identity operators to **compare objects**
- What these **Python operators** do under the hood

Table of Contents

- Comparing Identity With the Python is and is not Operators
- Comparing Equality With the Python == and != Operators
- Comparing the Python Comparison Operators
- Conclusion

Mark as Completed

Tweet Share Email

- Why using `is` and `is not` to compare values leads to **unexpected behavior**
- How to write a **custom `__eq__()` class method** to define equality operator behavior

Python Pit Stop: This tutorial is a **quick** and **practical** way to find the info you need, so you'll be back to your project in no time!

Free Bonus: Click here to get a Python Cheat Sheet and learn the basics of Python 3, like working with data types, dictionaries, lists, and Python functions.

Comparing Identity With the Python is and is not Operators

The Python `is` and `is not` operators compare the **identity** of two objects. In CPython, this is their memory address. Everything in Python is an object, and each object is stored at a specific memory location. The Python `is` and `is not` operators check whether two variables refer to the same object in memory.

Note: Keep in mind that objects with the same value are usually stored at separate memory addresses.

You can use `id()` to check the identity of an object:

Python >>>

```
>>> help(id)
Help on built-in function id in module builtins:

id(obj, /)
    Return the identity of an object.

    This is guaranteed to be unique among simultaneously existing objects.
    (CPython uses the object's memory address.)

>>> id(id)
2570892442576
```

The last line shows the memory address where the built-in function `id` itself is stored.

There are some common cases where objects with the same value will have the same id by default. For example, the numbers -5 to 256 are **interned** in CPython. Each number is stored at a singular and fixed place in memory, which saves memory for commonly-used integers.

You can use `sys.intern()` to intern strings for performance. This function allows you to compare their memory addresses rather than comparing the strings character-by-character:

Python >>>

```
>>> from sys import intern
>>> a = 'hello world'
>>> b = 'hello world'
>>> a is b
False
>>> id(a)
1603648396784
>>> id(b)
1603648426160

>>> a = intern(a)
>>> b = intern(b)
>>> a is b
True
>>> id(a)
1603648396784
>>> id(b)
1603648396784
```

The variables `a` and `b` initially point to two different objects in memory, as shown by their different IDs. When you intern them, you ensure that `a` and `b` point to the same object in

memory. Any new string with the value `'hello world'` will now be created at a new memory location, but when you intern this new string, you make sure that it points to the same memory address as the first `'hello world'` that you interned.

> **Note:** Even though the memory address of an object is unique at any given time, it varies between runs of the same code, and depends on the version of CPython and the machine on which it runs.

Other objects that are interned by default are `None`, `True`, `False`, and simple strings. Keep in mind that most of the time, different objects with the same value will be stored at separate memory addresses. **This means you should not use the Python `is` operator to compare values.**

When Only Some Integers Are Interned

Behind the scenes, Python interns objects with commonly-used values (for example, the integers -5 to 256) to save memory. The following bit of code shows you how only some integers have a fixed memory address:

Python >>>

```
>>> a = 256
>>> b = 256
>>> a is b
True
>>> id(a)
1638894624
>>> id(b)
1638894624

>>> a = 257
>>> b = 257
>>> a is b
False

>>> id(a)
2570926051952
>>> id(b)
2570926051984
```

Initially, `a` and `b` point to the same interned object in memory, but when their values are outside the range of **common integers** (ranging from -5 to 256), they're stored at separate memory addresses.

When Multiple Variables Point to the Same Object

When you use the assignment operator (`=`) to make one variable equal to the other, you make these variables point to the same object in memory. This may lead to unexpected behavior for mutable objects:

Python >>>

```
>>> a = [1, 2, 3]
>>> b = a
>>> a
[1, 2, 3]
>>> b
[1, 2, 3]

>>> a.append(4)
>>> a
[1, 2, 3, 4]
>>> b
[1, 2, 3, 4]
```

```
>>> id(a)
2570926056520
>>> id(b)
2570926056520
```

What just happened? You add a new element to `a`, but now `b` contains this element too! Well, in the line where `b = a`, you set `b` to point to the same memory address as `a`, so that both variables now refer to the same object.

If you define these lists independently of each other, then they're stored at different memory addresses and behave independently:

Python >>>

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
>>> id(a)
2356388925576
>>> id(b)
2356388952648
```

Because `a` and `b` now refer to different objects in memory, changing one doesn't affect the other.

Comparing Equality With the Python == and != Operators

Recall that objects with the **same value** are often stored at **separate memory addresses**. Use the equality operators `==` and `!=` if you want to check whether or not two objects have the same value, regardless of where they're stored in memory. In the vast majority of cases, this is what you want to do.

When Object Copy Is Equal but Not Identical

In the example below, you set `b` to be a copy of `a` (which is a mutable object, such as a list or a dictionary). Both variables will have the same value, but each will be stored at a different memory address:

Python >>>

```
>>> a = [1, 2, 3]
>>> b = a.copy()
>>> a
[1, 2, 3]
>>> b
[1, 2, 3]

>>> a == b
True
>>> a is b
False

>>> id(a)
2570926058312
>>> id(b)
2570926057736
```

`a` and `b` are now stored at different memory addresses, so `a is b` will no longer return `True`. However, `a == b` returns `True` because both objects have the same value.

How Comparing by Equality Works

The magic of the equality operator `==` happens in the `__eq__()` class method of the object to the left of the `==` sign.

> **Note:** This is the case unless the object on the right is a **subclass** of the object on the left. For more information, check the official documentation.

This is a magic class method that's called whenever an instance of this class is compared against another object. If this method is not implemented, then `==` compares the memory addresses of the two objects by default.

As an exercise, make a `SillyString` class that inherits from `str` and implement `__eq__()` to compare whether the length of this string is the same as the length of the other object:

Python

```
class SillyString(str):
    # This method gets called when using == on the object
    def __eq__(self, other):
        print(f'comparing {self} to {other}')
        # Return True if self and other have the same length
        return len(self) == len(other)
```

Now, a SillyString `'hello world'` should be equal to the string `'world hello'`, and even to any other object with the same length:

Python >>>

```
>>> # Compare two strings
>>> 'hello world' == 'world hello'
False

>>> # Compare a string with a SillyString
>>> 'hello world' == SillyString('world hello')
comparing world hello to hello world
True

>>> # Compare a SillyString with a list
>>> SillyString('hello world') == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
comparing hello world to [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
True
```

This is, of course, silly behavior for an object that otherwise behaves as a string, but it does illustrate what happens when you compare two objects using `==`. The `!=` operator gives the inverse response of this unless a specific `__ne__()` class method is implemented.

The example above also clearly shows you why it is good practice to use the Python `is` operator for comparing with `None`, instead of the `==` operator. Not only is it faster since it compares memory addresses, but it's also safer because it doesn't depend on the logic of any `__eq__()` class methods.

Comparing the Python Comparison Operators

As a rule of thumb, you should always use the equality operators `==` and `!=`, except when you're comparing to `None`:

- **Use the Python `==` and `!=` operators to compare object equality**. Here, you're generally comparing the value of two objects. This is what you need if you want to compare whether or not two objects have the same contents, and you don't care about where they're stored in memory.

- **Use the Python `is` and `is not` operators when you want to compare object identity**. Here, you're comparing whether or not two variables point to the same object in memory. The main use case for these operators is when you're comparing to `None`. It's faster and safer to compare to `None` by memory address than it is by using class methods.

Variables with the same value are often stored at separate memory addresses. This means that you should use `==` and `!=` to compare their values and use the Python `is` and `is not` operators only when you want to check whether two variables point to the same memory

address.

Conclusion

In this tutorial, you've learned that `==` and `!=` **compare the value of two objects**, whereas the Python `is` and `is not` operators compare whether two variables **refer to the same object in memory**. If you keep this distinction in mind, then you should be able to prevent unexpected behavior in your code.

If you want to read more about the wonderful world of **object interning** and the Python `is` operator, then check out Why you should almost never use "is" in Python. You could also have a look at how you can use `sys.intern()` to optimize memory usage and comparison times for strings, although the chances are that Python already automatically handles this for you behind-the-scenes.

Now that you've learned what the **equality and identity operators** do under the hood, you can try writing your own `__eq__()` class methods, which define how instances of this class are compared when using the `==` operator. Go and apply your newfound knowledge of these Python comparison operators!

About **Joska de Langen**

Joska is an Ordina Pythoneer who writes for Real Python.

» More about Joska

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:

 Geir Arne

 Jaya

 Joanna

 Mike

What Do You Think?

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.