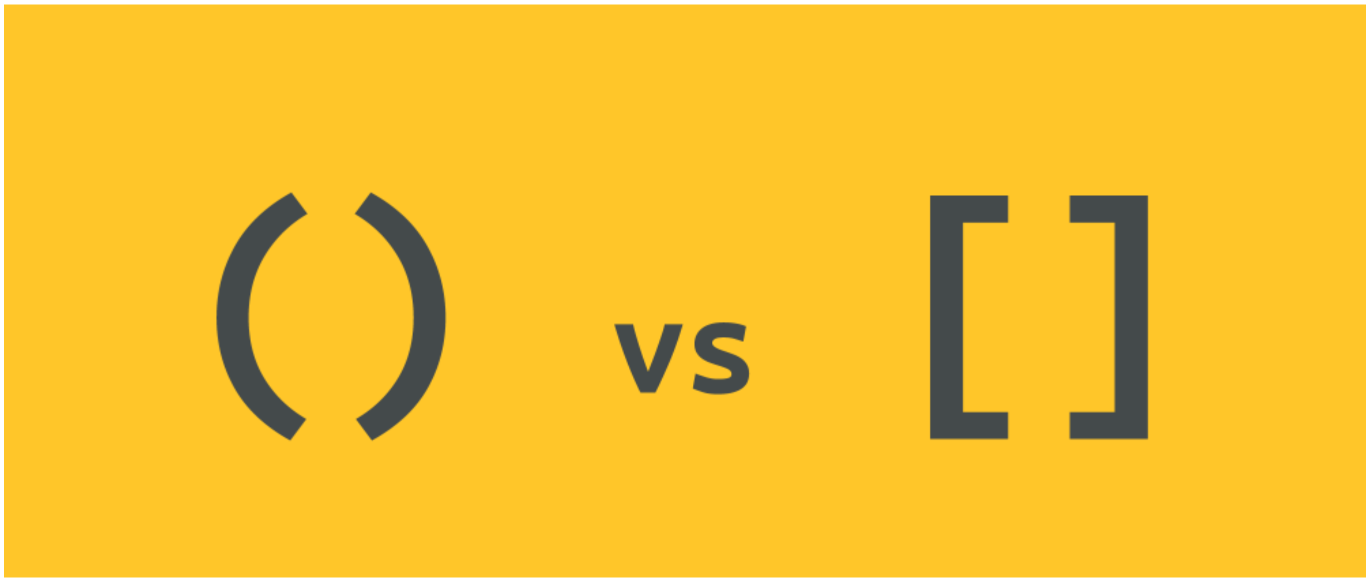

Immutable vs Mutable Data Types in Python

Learn the difference between mutable and immutable data types and how to find out which is which!

Lorraine Li Follow
Oct 17, 2019 · 5 min read ★

By now you may have heard the phrase "everything in Python is an object". **Objects** are abstraction for data, and Python has an amazing variety of data structures that you can use to represent data, or combine them to create your own custom data.

A first fundamental distinction that Python makes on data is about whether or not the value of an object changes. If the value can change, the object is called **mutable**, while if the value cannot change, the object is called **immutable**.

In this crash course, we will explore:

- The difference between mutable and immutable types
- Different data types and how to find out whether they are mutable or immutable

It is very important that you understand the distinction between mutable and immutable because it affects the code you write.

Let’s get started!

> *This crash course is adapted from Next Tech’s **Learn Python Programming** course that uses a mix of theory and practicals to explore Python and its features, and progresses from beginner to being skilled in Python. It includes an in-browser sandboxed environment with all the necessary software and libraries pre-installed. You can get started for free here!*

. . .

Mutable vs Immutable

To get started, it’s important to understand that every object in Python has an ID (or identity), a type, and a value, as shown in the following snippet:

```
age = 42
print(id(age))    # id
print(type(age))  # type
print(age)        # value

[Out:]
10966208
<class ‘int’>
42
```

Once created, the ID of an object never changes. It is a unique identifier for it, and it is used behind the scenes by Python to retrieve the object when we want to use it.

The type also never changes. The type tells what operations are supported by the object and the possible values that can be assigned to it.

The value can either change or not. If it can, the object is said to be mutable, while when it cannot, the object is said to be immutable.

Let’s take a look at an example:

```
age = 42
print(id(age))
print(type(age))
print(age)

age = 43
print(age)
print(id(age))

[Out:]
10966208
<class ‘int’>
42
43
10966240
```

Has the value of `age` changed? Well, no. `42` is an integer number, of the type `int`, which is immutable. So, what happened is really that on the first line, `age` is a name that is set to point to an `int` object, whose value is `42`.

When we type `age = 43`, what happens is that another object is created, of the type `int` and value `43` (also, the `id` will be different), and the name `age` is set to point to it. So, we didn’t change that `42` to `43`. We actually just pointed `age` to a different location.

As you can see from printing `id(age)` before and after the second object named `age` was created, they are different.

Now, let’s see the same example using a mutable object.

```
x = [1, 2, 3]
print(x)
print(id(x))

x.pop()
print(x)
print(id(x))
```

```
[Out:]
[1, 2, 3]
139912816421064
[1, 2]
139912816421064
```

For this example, we created a list named `m` that contains 3 integers, `1` , `2` , and `3` . After we change `m` by "popping" off the last value `3` , the ID of `m` stays the same!

So, objects of type `int` are immutable and objects of type `list` are mutable. Now let's discuss other immutable and mutable data types!

Mutable Data Types

Mutable sequences can be changed after creation. Some of Python's mutable data types are: **lists**, **byte arrays**, **sets**, and **dictionaries**.

Lists

As you saw earlier, lists are mutable. Here's another example using the `append()` method:

```
a = list(('apple', 'banana', 'clementine'))
print(id(a))

a.append('dates')
print(id(a))
```

```
[Out:]
140372445629448
140372445629448
```

Byte Arrays

Byte arrays represent the mutable version of `bytes` objects. They expose most of the usual methods of mutable sequences as well as most of the methods of the `bytes` type. Items are integers in the range [0, 256).

Let's see a quick example with the `bytearray` type to show that it is mutable:

```
b = bytearray(b'python')
print(id(b))

b.replace(b'p', b'P')
print(id(b))


[Out:]
139963525979808
139963525979808
```

Sets

Python provides two set types, `set` and `frozenset`. They are unordered collections of immutable objects.

```
c = set((‘San Francisco’, ‘Sydney’, ‘Sapporo’))
print(id(c))
c.pop()
print(id(c))

[Out:]
140494031990344
140494031990344
```

As you can see, `set`s are indeed mutable. Later, in the **Immutable Data Types** section, we will see that `frozenset`s are immutable.

Dictionaries

```
d = {
    'a': 'alpha',
    'b': 'bravo',
    'c': 'charlie',
    'd': 'delta',
    'e': 'echo'
}
print(id(d))

d.update({
    'f': 'foxtrot'
})
print(id(d))
```

```
[Out:]
14007111431940
14007111431940
```

Immutable Data Types

Immutable data types differ from their mutable counterparts in that they can not be changed after creation. Some immutable types include **numeric data types**, **strings**, **bytes**, **frozen sets**, and **tuples**.

Numeric Data Types

You have already seen that integers are immutable; similarly, Python's other built-in numeric data types such as booleans, floats, complex numbers, fractions, and decimals are also immutable!

Strings and Bytes

Textual data in Python is handled with `str` objects, more commonly known as **strings**. They are immutable sequences of **Unicode code points**. Unicode code points can represent a character.

When it comes to storing textual data though, or sending it on the network, you may want to encode it, using an appropriate encoding for the medium you're using. The result of an encoding produces a `bytes` object, whose syntax and behavior is similar to that of strings.

Both strings and bytes are immutable, as shown in the following snippet:

```
# string
e = 'Hello, World!'
print(id(e))

e = 'Hello, Mars!'
print(id(e))
```

```
[Out:]
140595675113648
140595675113776
```

```
# bytes
unicode = 'This is üŋíc0de'  # unicode string: code points
print(type(unicode))
f = unicode.encode('utf-8')  # utf-8 encoded version
print(type(f))
print(id(f))

f = b'A bytes object'        # a bytes object
print(id(f))


[Out:]
<class 'str'>
<class 'bytes'>
140595675068152
140595675461360
```

In the bytes section, we first defined `f` as an encoded version of our `unicode` string. As you can see from `print(type(f))` this is a `bytes` type. We then create another `bytes` object named `f` whose value is `b'A bytes object'`. The two `f` objects have different IDs, which shows that bytes are immutable.

Frozen Sets

As discussed in the previous section, `frozenset` s are similar to `set` s. However, `frozenset` objects are quite limited in respect of their mutable counterpart since they cannot be changed. Nevertheless, they still prove very effective for membership test, union, intersection, and difference operations, and for performance reasons.

Tuples

The last immutable sequence type we're going to see is the tuple. A tuple is a sequence of arbitrary Python objects. In a tuple, items are separated by commas. These, too, are immutable, as shown in the following example:

```
g = (1, 3, 5)
print(id(g))

g = (42, )
print(id(g))


[Out:]
```

```
139952252343784
139952253457184
```

. . .

I hope you enjoyed this crash course on the difference between immutable and mutable objects and how to find out which an object is! Now that you understand this fundamental concept of Python programming, you can now explore the methods you can use for each data type.

. . .

If you'd like to learn about this and continue to advance your Python skills, Next Tech has a full ***Learn Python Programming*** *course that covers:*

- *Functions*
- *Conditional programming*
- *Comprehensions and generators*
- *Decorators, object-oriented programming, and iterators*
- *File data persistence*
- *Testing, including a brief introduction to test-driven development*
- *Exception handling*
- *Profiling and performances*

You can get started here for free!

Programming Python Crash Course Coding