

# Python's Mutable vs Immutable Types: What's the Difference?

by [Leodanis Pozo Ramos](#) · Mar 08, 2023 · 3 Comments

[data-structures](#) [intermediate](#) [python](#)

Mark as Completed

Share Share Email

## Table of Contents

- [Mutability vs Immutability](#)
  - [Variables and Objects](#)
  - [Objects, Value, Identity, and Type](#)
  - [Mutable and Immutable Objects in Python](#)
  - [Considerations for Working With Mutable vs Immutable Objects](#)
- [Immutable Built-in Data Types in Python](#)
  - [Numbers](#)
  - [Booleans](#)
  - [Strings](#)
  - [Bytes](#)
  - [Tuples](#)
- [Mutable Built-in Data Types in Python](#)
  - [Lists](#)
  - [Dictionaries](#)
  - [Sets](#)
- [Opposite Variations of Sets and Bytes](#)
  - [Frozen Sets](#)
  - [Byte Arrays](#)
- [Mutability in Built-in Types: A Summary](#)
- [Common Mutability-Related Gotchas](#)
  - [Aliasing Variables](#)
  - [Mutating Arguments in Functions](#)
  - [Using Mutable Default Values](#)
  - [Making Copies of Lists](#)
  - [Getting None From Mutator Methods](#)
  - [Storing Mutable Objects in Tuples](#)
  - [Concatenating Many Strings](#)
- [Mutability in Custom Classes](#)
  - [Mutability of Classes and Instances](#)

— FREE Email Series —

Python Tricks

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

Email...

Get Python Tricks »

No spam. Unsubscribe any time.

Browse Topics

Guided Learning Paths

Basics

Intermediate

Advanced

[api](#)

[best-practices](#)

[career](#)

[community](#)

[databases](#)

[data-science](#)

[data-structures](#)

[data-viz](#)

[devops](#)

[django](#)

[docker](#)

[editors](#)

[flask](#)

[front-end](#)

[gamedev](#)

[gui](#)

[machine-learning](#)

[numpy](#)

[projects](#)

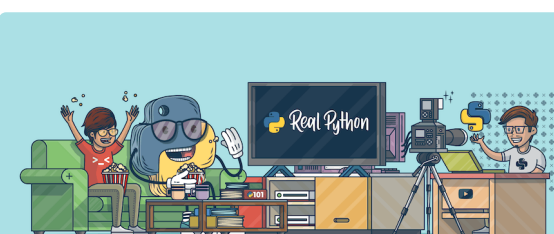
[python](#)

[testing](#)

[tools](#)

[web-dev](#)

[web-scraping](#)



## Master Real-World Python Skills With a Community of Experts

Level Up With Unlimited Access to Our Vast Library of Python Tutorials and Video Lessons

Watch Python Tutorials »

## Table of Contents

- [Mutability vs Immutability](#)
- [Immutable Built-in Data Types in Python](#)
- [Mutable Built-in Data Types in Python](#)
- [Opposite Variations of Sets and Bytes](#)
- [Mutability in Built-in Types: A](#)

- [Mutability of Attributes](#)
- [Techniques to Control Mutability in Custom Classes](#)
  - [Defining a `\_\_slots\_\_` Class Attribute](#)
  - [Providing Custom `\_\_setattr\_\_\(\)` and `\_\_delattr\_\_\(\)` Methods](#)
  - [Using Read-Only Properties and Descriptors](#)
  - [Building Immutable Tuple-Like Objects](#)
  - [Creating Immutable Data Classes](#)
- [Conclusion](#)



**Master Real-World Python Skills**  
With a Community of Experts  
Level Up With Unlimited Access to Our Vast Library  
of Python Tutorials and Video Lessons

**Watch Now »**

 [Remove ads](#)

As a Python developer, you'll have to deal with **mutable** and **immutable** objects sooner or later. Mutable objects are those that allow you to change their value or data in place without affecting the object's identity. In contrast, immutable objects don't allow this kind of operation. You'll just have the option of creating new objects of the same type with different values.

In Python, mutability is a characteristic that may profoundly influence your decision when choosing which data type to use in solving a given programming problem. Therefore, you need to know how mutable and immutable objects work in Python.

#### In this tutorial, you'll:

- Understand how **mutability** and **immutability** work under the hood in Python
- Explore immutable and mutable **built-in data types** in Python
- Identify and avoid some common **mutability-related gotchas**
- Understand and control how mutability affects your **custom classes**

To dive smoothly into this fundamental Python topic, you should be familiar with how [variables](#) work in Python. You should also know the basics of Python's built-in [data types](#), such as [numbers](#), [strings](#), [tuples](#), [lists](#), [dictionaries](#), [sets](#), and others. Finally, knowing how [object-oriented programming](#) works in Python is also a good starting point.

**Free Sample Code:** [Click here to download the free sample code](#) that you'll use to explore mutable vs immutable data types in Python.

## Mutability vs Immutability

In programming, you have an **immutable object** if you can't change the object's [state](#) after you've created it. In contrast, a **mutable object** allows you to modify its internal state after creation. In short, whether you're able to change an object's state or contained data is what defines if that object is mutable or immutable.

Immutable objects are common in [functional programming](#), while mutable objects are widely used in [object-oriented programming](#). Because Python is a multiparadigm programming language, it provides mutable and immutable objects for you to choose from when solving a problem.

To understand how mutable and immutable objects work in Python, you first need to understand a few related concepts. To kick things off, you'll take a look at variables and objects.

#### [Summary](#)

- [Common Mutability-Related Gotchas](#)
- [Mutability in Custom Classes](#)
- [Techniques to Control Mutability in Custom Classes](#)
- [Conclusion](#)

Mark as Completed



 **Share**

 **Share**

 **Email**



**High Quality**  
Python Video Courses  
**Watch Now »**



 [Remove ads](#)

## Variables and Objects

In Python, [variables](#) don't have an associated type or size, as they're labels attached to objects in memory. They point to the memory position where concrete objects live. In other words, a Python variable is a name that *refers to* or holds a *reference to* a concrete object. In contrast, Python **objects** are concrete pieces of information that live in specific memory positions on your computer.

The main takeaway here is that variables and objects are two different animals in Python:

- **Variables** hold references to objects.
- **Objects** live in concrete memory positions.

Both concepts are independent of each other. However, they're closely related. Once you've created a variable with an [assignment statement](#), then you can access the referenced object throughout your code by using the variable name. If the referenced object is mutable, then you can also perform **mutations** on it through the variable. Mutability or immutability is intrinsic to objects rather than to variables.

However, if the referenced object is immutable, then you won't be able to change its internal state or contained data. You'll just be able to make your variable reference a different object that, in Python, may or may not be of the same type as your original object.

If you don't have a reference (variable) to an object, then you can't access that object in your code. If you lose or remove all the references to a given object, then Python will [garbage-collect](#) that object, freeing the memory for later use.

Now that you know that there are differences between variables and objects, you need to learn that all Python objects have three core properties: identity, type, and value.

## Objects, Value, Identity, and Type

In Python, everything is an object. For example, [numbers](#), [strings](#), [functions](#), [classes](#), and [modules](#) are all objects. Every Python object has three core characteristics that define it at a foundational level. These characteristics are:

1. Value
2. Identity
3. Type

Arguably, the **value** is probably the most familiar object characteristic that you've dealt with. An object's value consists of the concrete piece or pieces of **data** contained in the object itself. A classic example is a numeric value like an [integer](#) or [floating-point](#) number:

Python



```
>>> 42
42
>>> isinstance(42, int)
True

>>> 3.14
3.14
>>> isinstance(3.14, float)
True
```

These numeric values, 42 and 3.14, are both objects. The first number is an instance of the built-in `int` class, while the second is an instance of `float`. In both examples, you confirm the object's type using the built-in `isinstance()` function.

**Note:** In this tutorial, you'll use the term **value** to refer to an object's data. Your objects will have custom and meaningful values only if you add those values yourself. Sometimes, you'll create or find objects that don't have meaningful values. However, all Python objects contain built-in attributes that also hold data. So, in a way, each Python object has an implicit value.

For example, all Python objects will have special methods and attributes that the language adds automatically under the hood:

Python



```
>>> obj = object()
>>> dir(obj)
[
    '__class__',
    '__delattr__',
    ...
    '__str__',
    '__subclasshook__'
]
```

In this example, you created a generic object using the built-in `object` class. You haven't added any attributes to the object. So, from your programming perspective, this object doesn't have a meaningful value. So, you may think it only has identity and type, which you'll explore in just a moment.

However, because all objects have built-in attributes, you can mutate them by adding new attributes and data dynamically, as you'll learn in the [Mutability in Custom Classes](#) section of this tutorial.

Python also supports objects with way more complex values that hold several data points. For example, you can have a list of numbers like the following:

Python

```
>>> [1, 2, 3]
[1, 2, 3]
>>> isinstance([1, 2, 3], list)
True
```

In this case, you have an instance of the built-in `list` class, and its value is `[1, 2, 3]`. Of course, the examples in this section are just a sample of the infinite range of values and data that Python objects can store. That's especially true if you consider that you can also create your own classes and objects in Python.

An object's value lives in a concrete position or address in your computer's memory. In Python's [CPython implementation](#), this specific memory address is what you'll know as the object's **identity**. This identity is a unique identifier that distinguishes



one object from others.

An object's identity is a read-only property, which means that you can't change an object's identity once the object has been created.

You can use the built-in `id()` function to obtain an object's identity:

Python



```
>>> id(42)
4343440904

>>> id(3.14)
4376764112

>>> id([1, 2, 3])
4399577728
```

If you call `id()` with a given object, then you get a number that acts as the object's identity. Again, in CPython, this number represents the memory address where the object lives in your computer. However, in other [Python implementations](#), this number may have a different origin.

**Note:** You'll get different results from `id()` on your computer after running the examples above. So, you shouldn't expect the exact same output as in the examples.

Before jumping into the third characteristic of Python objects, take a minute to think of the object's identity. Does it remind you of something you learned a moment ago? Yes, you can also access an object's identity through a variable that points to that object:

Python



```
>>> number = 42
>>> id(number)
4343440904
```

In this example, the variable `number` points to an integer object with a value of 42. From this point on, you can access the object, 42, through the variable, `number`.

**Note:** In this tutorial, you'll use the built-in `id()` function a lot. So, take a moment to understand what this function does and how it works. For extra support, you can turn to the function's [documentation](#).

The final characteristic of every Python object is its **type**. The type of an object determines which class that object derives from. You can use the built-in `type()` function to learn about an object's type:

Python



```
>>> type(42)
<class 'int'>

>>> type("Hello, World!")
<class 'str'>

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

Here, you called `type()` with three different objects. The first object derives from the `int` class, so `int` is its type. The second object is a string, so its type is `str`. Finally, you have an object of type `list`.

**Note:** You can also determine the type of an object by accessing its `.__class__` attribute directly.

Here's how you'd do that:

Python



```
>>> (42).__class__  
<class 'int'>  
  
>>> "Hello, World!".__class__  
<class 'str'>  
  
>>> [1, 2, 3].__class__  
<class 'list'>
```

As you can see, the result is the same as when you used `type()`. That's because `type()` falls back to returning the value stored in `.__class__`. Even though `.__class__` works, you should use `type()` to retrieve the type of a given object and avoid direct access to special attributes as much as possible, especially in cases where a dedicated built-in function exists.

Note that to access `.__class__` on an integer value, you need to enclose the value in a pair of parentheses because, otherwise, the dot will turn the integer into a floating-point number.

An object's type defines the operations that you can perform with or on that object. It also defines the accepted values. For example, you can perform arithmetic operations with numbers:

Python



```
>>> 5 + 2  
7  
  
>>> 10 - 5  
5  
  
>>> 21 * 2  
42  
  
>>> 15 / 3  
5
```

As expected, numbers support the four elementary arithmetic operations and many other operations, such as exponentiation, modulo, square root, and absolute value. They also support other operations, such as comparison.

Similarly, other types of objects support other types of operations. For example, `list` objects allow you to determine their number of items using the `len()` function:

Python



```
>>> len([1, 2, 3])  
3
```

In this example, you use the built-in `len()` function to get the number of items contained in a list object. In this case, the input list has three values in it. Now note how you can't use `len()` with a number:

Python



```
>>> len(42)
Traceback (most recent call last):
...
TypeError: object of type 'int' has no len()
```

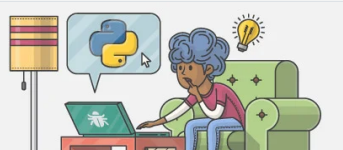
If you call `len()` with a number as an argument, then you get a `TypeError` because numeric types don't support this specific operation.

When it comes to mutable and immutable objects, the object's type is crucial. In the end, it's the object's type that determines the operations that you can do on that object. If the type supports mutations, then you'll have mutable derived objects. Otherwise, you'll have immutable derived objects.

Now that you know a few things about Python objects, it's time for you to dive into the core topic of this tutorial: *mutable and immutable objects in Python*.

## Learn Python Programming, By Example

realpython.com



[Remove ads](#)

## Mutable and Immutable Objects in Python

In Python, the identity of an object is the only read-only characteristic. You'll set it when you first create the object itself. The type of an object is generally fixed but not read-only. You can change an object's type by reassigning its `__class__` class attribute to a different class:

Python



```
>>> class Person:
...     def __init__(self, name):
...         self.name = name
...

>>> class Student(Person):
...     def __init__(self, name, major):
...         super().__init__(name)
...         self.major = major
...

>>> john = Student("John", "Computer Science")
>>> type(john)
<class '__main__.Student'>

>>> john.__class__ = Person
>>> type(john)
<class '__main__.Person'>

>>> john.name
John
>>> john.major
Computer Science
```

In this example, you create two classes. `Student` inherits from `Person` and extends it with an additional attribute, `major`. Then you create an instance of `Student` with appropriate arguments. Finally, you reassign the object's `__class__` attribute to point to the `Person` class. Note that this change modifies the object's type but not the object's current attributes.

**Note:** Python allows you to change the type of an object dynamically. This doesn't mean that you should. Changing an object's type dynamically could cause unexpected behaviors. So, try to avoid this practice in your code.

An object's value is probably the only characteristic that you'd want to change in your code. An object that allows you to change its values without changing its identity is a **mutable** object. The changes that you can perform on a mutable object's value are known as **mutations**.

In contrast, an object that doesn't allow changes in its value is an **immutable** object. You have no way to perform mutations on this kind of object. You just have the option of creating a new object of the same type but with a different value, which you can do through a new assignment.

In Python, you'll have several built-in types at your disposal. Most of them are immutable, and a few are mutable. Single-item data types, such as integers, floats, complex numbers, and [Booleans](#), are always immutable. So, you have no way to change the value of these types. You just have the option of creating a new object with a new value and throwing away the old one.

When it comes to **collection** or **container** types, such as strings, lists, tuples, sets, and dictionaries, things are a bit different. Like all other objects, a collection has a unique identity. However, collections internally keep references to all their individual values. This opens the opportunity for mutability.

According to the definition of a mutable object, if you can change the value of that object without changing the object's identity, then you have a mutable object. In a collection type, you have the collection's identity and probably several item identities. You can modify the value—and, therefore, the identity—of some of these items while leaving the collection's identity unchanged. In that case, you have a mutable collection.

Python has both mutable and immutable collection data types. Strings and tuples are immutable, while lists, dictionaries, and sets are mutable. This distinction is crucial for you as a Python developer. You'll explore it in more detail later in this tutorial. Before that, you'll learn about some facts that you should keep in mind when working with mutable and immutable objects.

## Considerations for Working With Mutable vs Immutable Objects

When working with mutable and immutable data types in Python, you must consider how using one or the other category of objects would impact your code.

First, to work with immutable objects, you may need *more memory* because you can't mutate the data directly in the object itself. You need to create new objects instead, which may lead to many different but related copies of the same underlying data.

Second, mutable objects are known to make working with [threads](#) a challenging task. Threads that mutate an object's value without proper synchronization can corrupt the data contained in that object. To work around this issue, you need to use synchronization, which may end up causing deadlocks. Most importantly, excessive synchronization weakens concurrency.

Finally, immutable objects facilitate reasoning about your code. To understand a piece of code that constantly mutates an object, you must remember all the possible moments in which the underlying data was mutated. This isn't true with immutable objects because you can't mutate their data or value.

With all this theoretical knowledge about mutable and immutable objects, you're ready to dive into Python's built-in immutable data types.



# Immutable Built-in Data Types in Python

As you've already learned, Python provides a rich set of immutable built-in data types. Numeric types, such as `int`, `float`, and `complex`, can hold single-item objects that you can't mutate during your code execution. These types are immutable. In Python, the `bool` class, which supports the Boolean type, is a subclass of `int`. So, this type is also immutable.

When it comes to collection types, `str`, `bytes`, and `tuple` are also immutable even though they can store multiple-item values.

As you can conclude, the vast majority of built-in data types in Python are immutable. In the following sections, you'll dive a bit deeper into how each of these immutable types works and how their immutability can affect your code.

**Write Cleaner & More Pythonic Code**

realpython.com



 [Remove ads](#)

## Numbers

Numbers are the first and most common example of immutable built-in types in Python. You'll have no way to access the interior of a numeric value. Because of this limitation, you won't be able to mutate its value without altering its identity. This is true for all the built-in numeric types—`int`, `float`, and `complex`—as you already learned.

In Python, when you want to use a given object throughout your code, you typically assign that object to a variable and then reuse that variable whenever you need to. Now consider the following example of a variable that points to a numeric value:

Python



```
>>> number = 314
>>> number
314

>>> id(number)
9402151432
```

In this example, you use an assignment statement to create a new variable that holds a reference to an integer number. The `id()` function returns the number's identity.

Now say that you don't know how Python variables work and you think *I can probably change the value of this integer variable using another assignment*:

Python



```
>>> number = 7
>>> number
7

>>> id(number)
4343439784
```

If you compare the return value of `id()` in this example with the value you got in the previous example, then you'll note that they're different. So, instead of updating the value of `number`, you've created a completely new value and made your variable refer to it.

Once you've changed `number` to refer to 7, then the old 314 gets garbage-collected,

which frees up the memory assigned to storing the object.

When it comes to numbers, Python just allows you to reassign the variable name to point to a new numeric value that lives at a completely different memory address. It doesn't allow you to change the value stored at the original memory address. That's why 314 has the identity 9402151432, and 7 has the identity 4343439784. This behavior is the base of immutability in numeric types.

Because numeric types are immutable, it doesn't matter how many variables you have pointing to a given object. Every reference will point to the same value until you explicitly reassign the variable name:

Python



```
>>> word_counter = letter_counter = 0
>>> id(word_counter)
4343439560
>>> id(letter_counter)
4343439560

>>> word_counter += 1
>>> word_counter
1
>>> id(word_counter)
4343439592

>>> letter_counter
0
>>> id(letter_counter)
4343439560
```

In this example, you use a [multiple assignment](#) to create two variables that refer to the same number. After that, both variables hold the same reference and therefore point to the same instance of the number 0.

If you reassign either variable to hold a new number, then the second variable will still point to the original number. That's why after doing `word_counter += 1`, the `id()` function returns a different identity number. Note how `letter_counter` kept the original data.

## Booleans

Python has a built-in `bool` type that can take two possible values: `True` or `False`. These values are [built-in constants](#) set to 1 and 0, respectively. The `bool` type is a subclass of `int`. To confirm these internal features of `bool`, go ahead and run the code below:

Python



```
>>> isinstance(bool, int)
True

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

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

Here, you first call the built-in `isinstance()` function to check that `bool` is actually a subclass of `int`. Then you call `isinstance()` to check that both `True` and `False` are instances of `int`. Finally, you call `int()` to uncover that `True` has a value of 1 and

False has a value of 0.

Because the `bool` type is a subclass of `int`, it inherits the parent's immutability:

Python

```
>>> has_permission = True
>>> id(has_permission)
4342432520

>>> has_permission = False
>>> id(has_permission)
4342432552
```

In this example, you didn't change the value of `has_permission` from `True` to `False`. You actually reassigned `has_permission`, which now refers to a different object with a different identity.

This code works like this because consecutive assignments to a variable make that variable refer to completely different objects every time rather than mutating the object in place. Note that this behavior doesn't depend on an object's mutability but on the way that assignments work in Python.

## Python Tricks The Book

A Buffet of Awesome Python Features

[Get Your Free Sample Chapter](#)



[Remove ads](#)

## Strings

Python strings are [sequences](#) of individual characters. Because they're sequences, you can efficiently access any individual character in a given string using the **indexing operator** (`[]`) and a zero-based integer index:

Python

```
>>> greeting = "Hello!"

>>> greeting[0]
'H'
>>> greeting[1]
'e'
>>> greeting[2]
'l'
```

In these examples, you use indexing on a string object to access individual characters. Because you can access characters this way, you can also use `id()` to learn the identity of each character:

Python

```
>>> id(greeting[0])
4343484232
>>> id(greeting[1])
4343485856
>>> id(greeting[2])
4343486248
```

As you can see in these examples, every unique character in a string has its own identity, which is completely independent of the identity of the string object itself:

Python

```
>>> id(greeting)
4398429680
```

With this result, it may seem like Python strings can be mutated [in place](#). However, they can't. Strings are immutable objects in Python.

**Note:** In-place algorithms consist of transforming the content of a given [data structure](#) by operating on the data structure itself and without using or creating any auxiliary similar data structure. These algorithms play an important role in Python's mutable built-in data types, as you'll learn throughout this tutorial.

If you try to use the indexing syntax to change the value of a specific character in place with the help of an assignment, then you get a `TypeError` exception:

Python



```
>>> greeting[1] = "E"
Traceback (most recent call last):
  ...
TypeError: 'str' object does not support item assignment
```

The message in this exception [traceback](#) is pretty enlightening. String objects don't support item assignments in Python. In other words, you can't reassign the value of a given character in an existing string. Once you create a string object, there's no way to modify that object's value without affecting the object's identity.

**Note:** Sometimes, having a mutable string can be beneficial. As you've learned, changing the value of strings many times may cause an important memory and performance burden because you'll have to create many new string objects just to throw them away.

You can use a `bytearray` object to emulate a mutable string, which will help you deal with those memory and performance issues:

Python



```
>>> greeting = "Hello!"

>>> mutable_greeting = bytearray(greeting.encode("utf-8"))
>>> id(mutable_greeting)
140293973558064

>>> mutable_greeting[1] = ord("E")
>>> mutable_greeting
bytearray(b'Hello')

>>> id(mutable_greeting)
140293973558064
```

The `bytearray` type is a decent tool for emulating mutable strings. You'll learn more about this type in the [Byte Arrays](#) section.

Additionally, it may seem possible to change the value of an existing string object by using an assignment. However, that's not doable. This kind of assignment creates a completely new object:

Python



```
>>> greeting = "Hello!"
>>> id(greeting)
4391270704

>>> greeting = "Hello, World!"
>>> id(greeting)
4391910000
```

Here, you've used an assignment to switch the reference of `greeting` from `"Hello!"` to `"Hello, World!"`. Note how the identities of the objects are different now.

Strings are quite useful in Python programming. They allow you to process textual data, which is commonplace in computer programming. Python strings implement a rich set of formatting methods, allowing you to transform your textual data as needed.

Because string objects are immutable, changing a string requires creating a new, modified copy of the original string. That's why most [string methods](#) work by returning new string objects rather than by modifying the target objects in place. The new string objects contain the original string with the desired modifications.

**Note:** The list of currently available string methods is rather long. It provides many interesting transformations that you can efficiently and quickly use in your code. So, check out the [str class documentation](#) for more details.

Consider the following examples that show how to use some popular string formatting methods:

Python



```
>>> greeting = "Hello, Pythonistas!"
>>> greeting
'Hello, Pythonistas!'
>>> id(greeting)
4376850912

>>> greeting = greeting.upper()
>>> greeting
'HELLO, PYTHONISTAS!'
>>> id(greeting)
4376845632

>>> greeting = greeting.lower()
>>> greeting
'hello, pythonistas!'
>>> id(greeting)
4395590768

>>> greeting = greeting.title()
>>> greeting
'Hello, Pythonistas!'
>>> id(greeting)
4376411504
```

In these examples, you've performed a few transformations on your original string object, `"Hello, Pythonistas!"`. Note how every formatter method returns a completely new and transformed string object. You can confirm this behavior by inspecting the object's identity as usual.



Your [Practical Introduction to Python 3](#) »

[Remove ads](#)

## Bytes

The built-in [bytes](#) type is also an immutable type in Python. As its name suggests, bytes stores binary data. Byte literals may look pretty similar to string literals because their syntax is mostly the same as that of string literals. You just need to add a `b` prefix:



Python



```
>>> greeting_str = "Hello, World!"
>>> type(greeting_str)
<class 'str'>
```

```
>>> greeting_bytes = b"Hello, World!"
>>> type(greeting_bytes)
<class 'bytes'>
```

In the first example, you create a normal string object using some text as a literal. In the second example, you use the same text to create a bytes object. To do this, you add a `b` prefix to your text. The rest of the syntax is completely compatible between strings and bytes, including using different quote combinations.

A significant difference between strings and bytes is that the latter only allows for [ASCII](#) characters in its literals. If you ever need to store a binary value over 127 in a bytes object, then you must enter it using the appropriate escape sequence. Alternatively, you can turn a Python string into a bytes object using a specific character encoding and the `.encode()` method:

Python



```
>>> bytes("Español".encode("utf-8"))
b'Espa\xc3\xba1o1'
```

In this example, the character "ñ" has a [Unicode](#) point greater than 127. Still, you can store the value in a bytes object using the `encode` method with the appropriate character encoding.

Like regular strings, bytes are also immutable. If you try to change a character in a bytes object, then you get an error:

Python



```
>>> greeting_bytes = b"Hello, World!"
>>> greeting_bytes[0] = "h"
Traceback (most recent call last):
  ...
TypeError: 'bytes' object does not support item assignment
```

You can't mutate an existing bytes object in place. If you try to do that, then you get a `TypeError`, as in the example above. Note that the error message is mostly the same as the message you get when you try to do a similar operation on a string.

## Tuples

Python's tuples are another example of an immutable collection of data. You can't modify them once created. Like strings, tuples are sequences. However, unlike strings, tuples allow you to store any type of object rather than just characters. This feature probably seems great at first glance. However, it can cause issues when the stored objects are mutable, as you'll learn later in this tutorial.

**Note:** As you've learned, Python tuples can store any type of object, including mutable ones. This characteristic of tuples may be the source of errors and bugs. You'll learn more about it in the [Mutable Objects in Tuples](#) section.

To create a tuple object, you can use a literal that consists of a comma-separated series of values:

Python



```
>>> letters = ("a", "b", "c", "d")
>>> type(letters)
<class 'tuple'>
```

In this example, you create a `letters` tuple using a literal. This specific tuple holds string objects. Note how the `type()` function says that your `letters` variable refers to an instance of the `tuple` class.

**Note:** The pair of parentheses isn't necessary when defining tuples. You can remove them and get the same result:

Python



```
>>> letters = "a", "b", "c", "d"
>>> type(letters)
<class 'tuple'>
```

Even though the parentheses aren't required for you to define a tuple, using them is common practice in the Python community because they make the code clearer and more readable.

Like with strings and other sequences, you can use the indexing operator to access individual items from an existing tuple. However, you can't use indexing on the left side of an assignment because tuples are immutable:

Python



```
>>> letters[0]
'a'
>>> letters[1]
'b'
>>> letters[2]
'c'

>>> letters[0] = "A"
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

In these examples, note that you can access the items of a tuple object using integer indices that start from 0. However, you can't use this syntax to change the value of a particular item in the underlying tuple. This operation is forbidden and raises a `TypeError`.

In practice, you'll use tuples when you need an ordered sequence of values that never changes during its lifetime. Good examples of where to use tuples include records from a [SQL](#) database and lines from a [CSV](#) file. Tuples like these tell someone reading your code that you don't intend for that sequence of values to change.

Because tuples are immutable, they don't provide methods to operate and transform the underlying data. Tuples are pretty lightweight objects with only two methods:

- `.count()` returns the number of occurrences of a particular item in the target tuple.
- `.index()` returns the first index of a given item in the target tuple. It raises a `ValueError` if the item isn't present in the tuple.

Unlike strings, tuples have no methods for transforming the data because their data can be of multiple types rather than a single type. Finally, tuples don't have methods to perform data transformations in place because they're immutable.

 [Remove ads](#)

## Mutable Built-in Data Types in Python

Mutable data types are another face of the built-in types in Python. The language provides a few useful mutable collection types that you can use in many situations. These types allow you to change the value of specific items without affecting the identity of the container object.

In the following sections, you'll learn about lists, which are arguably the classic example of a mutable type in Python. To complete your tool kit of mutable types, you'll learn about dictionaries and sets. Understanding the basics of these data types will help you decipher how mutability works under the hood.

### Lists

Python lists are a classic example of a mutable data type. Like tuples, lists are sequences of arbitrary objects. In a list, however, you can change the value of any item without altering the list's identity. In other words, you can change the value of a list object in place.

Because lists are sequences, you can use indexing and [slicing](#) operations to access individual data items:

Python



```
>>> digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> digits[0]
0
>>> digits[9]
9

>>> digits[3:7]
[3, 4, 5, 6]
>>> digits[2::2]
[2, 4, 6, 8]
>>> digits[1::2]
[1, 3, 5, 7, 9]
```

In this example, you create a list object to hold the digits. Then you use the indexing operator to access the digit at index 0 and the digit at index 9. Again, the indices are integer values that start from 0.

Then you use the slicing syntax to retrieve several items from your input list in one go. The first slice returns the items from index 3 up to index 7 without including the last one. The second slice gets the items from index 2 to the end of the list but jumps through two items every time. The final slice does a similar job.

Because lists are mutable, you can use the indexing operator (`[index]`) not only to access individual items but also to mutate them. To do this, you'll use an assignment statement like the following:

Python



```
>>> numbers = [1, 2, 314]
>>> id(numbers)
4390459520
>>> id(numbers[2])
9402151432

>>> numbers[2] = 3 # Mutation
>>> id(numbers)
4390459520
>>> id(numbers[2])
4343439656


>>> numbers
[1, 2, 3]
```

The original value of `numbers` was `[1, 2, 314]`. Because lists are mutable, you can change the value of `numbers` in place. That's what you did with the assignment `numbers[2] = 3`.

Note how the list's identity doesn't change when you do the actual mutation. However, the identity of the mutated data item has changed. Now, index 2 holds a reference to a new memory position when you store a new number object.

The net result is that you've replaced 314 with 3 in your list of numbers. After this operation, you've lost all the references to your old 314 value. Because of this, Python garbage-collects the old 314 and frees the corresponding memory.

You can also use the slicing operator (`[start:stop:step]`) on the left side of an assignment to mutate multiple items in a list at a time:

```
Python 
>>> letters = ["A", "B", "c", "d"]
>>> letters[2:] = ["C", "D"]
>>> letters
['A', 'B', 'C', 'D']
```

In this example, you mutate your `letters` list by replacing the letters from index 2 up to the end of the list with uppercase letters.

In summary, you'll be able to perform the following mutations on an existing list using the assignment operator:

Mutation	Description	Syntax
Item assignment	Replaces the data item stored at a given index with a new data item, <code>new_value</code>	<code>a_list[index] = new_value</code>
Slice assignment	Replaces the data items within a given slice of the list	<code>a_list[start:stop:step] = new_values</code>
Item deletion	Deletes the data item at a given index	<code>del a_list[index]</code>

Mutation	Description	Syntax
Slice deletion	Deletes the data items within a slice of the list	<code>del a_list[start:stop:step]</code>

You’ve already learned about the first two ways to mutate a list. The last two ways involve shrinking the list and reducing the number of items using the [del statement](#). These two operations open a new dimension of mutations that you can perform on list objects. Yes, just like removing items from a list, you can also add new items to it.

The following methods allow you to do several different mutations on `list` objects. They allow you to deal with adding and removing items from your list objects:

Method	Description
<code>a_list.append(item)</code>	Appends <code>item</code> to the end of <code>a_list</code> .
<code>a_list.clear()</code>	Removes all items from <code>a_list</code> .
<code>a_list.extend(iterable)</code>	Extends <code>a_list</code> with the contents of <code>iterable</code> .
<code>a_list.insert(index, item)</code>	Inserts <code>item</code> into <code>a_list</code> at <code>index</code> .
<code>a_list.pop(index)</code>	Returns and removes the item at <code>index</code> . With no argument, it returns the last item.
<code>a_list.remove(item)</code>	Removes the first occurrence of <code>item</code> from <code>a_list</code> .
<code>a_list.reverse()</code>	Reverses the items of <code>a_list</code> in place.
<code>a_list.sort(key=None, reverse=False)</code>	Sorts the items of <code>a_list</code> in place.

A critical point to note about `list` methods is that because lists are mutable, their methods change the underlying list in place and return `None`, instead of returning a new list. You should remember this behavior because it differs from string methods, which return a new string.

Consider the following examples:

Python

```
>>> numbers = [2, 4, 3, 1]

>>> last_added = numbers.append(5)
>>> print(last_added)
None
>>> numbers
[2, 4, 3, 1, 5]

>>> sorted_numbers = numbers.sort()
>>> print(sorted_numbers)
None
>>> numbers
[1, 2, 3, 4, 5]
```



All the list-mutating methods perform their expected transformations in place. So, instead of returning a new list object, they return `None`. This behavior may cause issues for Python beginners who expect to get a different result from calling a list method.

List objects support two additional operations that you can perform with two operators:

1. **Concatenation**, which uses the plus operator `+`
2. **Repetition**, which uses the multiplication operator `*`

What do these two operations have to do with mutability? They have [augmented syntax](#) variations that perform their changes in place. These variations use the following augmented operators:

- `+=` for concatenation
- `*=` for repetition

Concatenation consists of adding together multiple list objects to get a list containing all the items from the original lists:

Python

```
>>> # Regular operator
>>> numbers = [1, 2, 3] + [4, 5, 6]
>>> numbers
[1, 2, 3, 4, 5, 6]

>>> # Augmented operator
>>> numbers = [1, 2, 3]
>>> id(numbers)
4376770304
>>> numbers += [4, 5, 6]
>>> numbers
[1, 2, 3, 4, 5, 6]
>>> id(numbers)
4376770304
```

In the first example, you concatenate two lists of numbers using the plus operator. Note that the operator returns a new list object that you assign to `numbers`. In the second example, you have an initial list. You use the augmented concatenation operator to append the content of a new list to `numbers`. Note how the identity of `numbers` doesn't change in the process.

**Note:** Both mutable and immutable sequences support the `+=` and `*=` augmented assignment operators, but they do so differently. Mutable sequences like lists support the `+=` operator through the `.__iadd__()` method, which performs an in-place addition.

Similarly, mutable sequences support the `*=` operator through the `.__imul__()` method, which also performs the operation in place, modifying the underlying sequence.

In contrast, immutable sequences, such as tuples and strings, don't have the `.__iadd__()` and `.__imul__()` methods. Instead, augmented concatenations and repetitions fall back to calling `.__add__()` and `.__mul__()`, respectively. These two methods don't modify the underlying sequence in place but return new sequences.

Repetition consists of repeating the data contained in a given list several times. To do this operation, you use the following syntax:

Python



```
>>> # Regular operator
>>> letters = ["A", "B", "C"] * 3
>>> letters
['A', 'B', 'C', 'A', 'B', 'C', 'A', 'B', 'C']

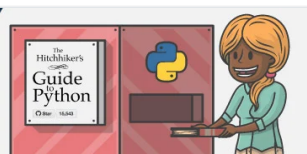
>>> # Augmented operator
>>> letters = ["A", "B", "C"]
>>> id(letters)
4379483328
>>> letters *= 3
>>> letters
['A', 'B', 'C', 'A', 'B', 'C', 'A', 'B', 'C']
>>> id(letters)
4379483328
```

In the initial example, you create a new list by repeating the data in your input list, `["A", "B", "C"]`, three times. Again, this operation returns a new list object that you store back in `letters`. In the second example, you use the augmented repetition operator to repeat the content of `letters` three times and store it back in `letters`. Again, the list's identity doesn't change in the process.

Because augmented concatenations and repetitions allow you to change the value of list objects without affecting the identity of the list, these two operations are actual mutations.

**Your Guide to the Python Programming Language and a Best Practices Handbook**

[python-guide.org](https://python-guide.org)



[Remove ads](#)

## Dictionaries

Dictionaries are the only [mapping](#) type among Python's built-in data types. They allow you to store a collection of key-value pairs. Like lists, dictionaries are also mutable, so you can change the value of a dictionary—specifically, its key-value pairs—without altering the dictionary's identity.

The keys of a dictionary work as unique identifiers that hold references to specific values. In other words, keys are like variables defined within a dictionary. You can use keys to access and mutate the values stored in a given dictionary.

In general, you can perform at least three types of mutations on an existing dictionary. You can:

1. Change the value associated with an **existing key**
2. Add **new key-value** pairs
3. Remove an **existing key-value** pair

All these possible mutations involve using keys to change the underlying dictionary in place. Here are a few examples of how these mutations work in practice:

Python



```
>>> inventory = {"apple": 100, "orange": 80, "banana": 120}
>>> inventory
{"apple": 100, "orange": 80, "banana": 120}

>>> inventory["orange"] = 140 # Change
>>> inventory
{'apple': 100, 'orange': 140, 'banana': 120}

>>> inventory["lemon"] = 200 # Add
>>> inventory
{'apple': 100, 'orange': 140, 'banana': 120, 'lemon': 200}

>>> del inventory["banana"] # Remove
>>> inventory
{'apple': 100, 'orange': 140, 'lemon': 200}

>>> del inventory["grape"]
Traceback (most recent call last):
  ...
KeyError: 'grape'
```

In the first example, you use the `dict[key] = value` syntax to change the value stored under the `orange` key. In the second example, you use the same syntax to add a nonexistent key, `lemon`, to your dictionary. This new key will hold a value of 200.

Finally, you use the `del dict[key]` syntax to remove `"banana": 120` from the dictionary. It's important to note that if you use the `del` statement with a nonexistent key as an argument, then you get a [KeyError](#) exception because the target key isn't present.

**Note:** Unfortunately, Python doesn't have a built-in immutable dictionary-like type. This type would be beneficial in situations where you need to [cache](#) function calls that take dictionaries as arguments. Caching requires the arguments to be hashable, which implies they must be immutable. You'll learn more about hashability in just a moment.

A quick solution to this caching issue would be to transform the dictionaries into tuples of key-value pairs with the `.items()` method.

Like lists, Python dictionaries also provide a few methods that allow you to perform mutations. Here's a summary of them:

Operation	Description
<code>a_dict.clear()</code>	Removes all key-value pairs from <code>a_dict</code> .
<code>a_dict.pop(key[, default])</code>	Removes the key-value pair under <code>key</code> and returns the value, or <code>default</code> if the key doesn't exist.
<code>a_dict.popitem()</code>	Removes and returns the most recently added key-value pair as a tuple like <code>(key, value)</code> .
<code>a_dict.setdefault(key[, default])</code>	Inserts a new key-value pair with <code>default</code> as its value and <code>key</code> as its key if <code>key</code> doesn't exist. Then returns the value associated with <code>key</code> .

Operation	Description
<code>a_dict.update([other])</code>	Updates the dictionary with the key-value pairs from <code>other</code> , overwriting existing keys and creating new keys for missing ones.

The `.pop()` method raises a `KeyError` exception if the target key isn't present in the dictionary and `default` isn't provided. The `.popitem()` method returns key-value pairs in [LIFO \(last in, first out\)](#) order.

As noted before, the keys in a dictionary must be unique. You can't have duplicate keys. Keys also have another important constraint. They must be [hashable](#) objects.

For an object to be hashable, you must be able to pass it to a [hash function](#) and get a unique hash code. To achieve this, your object must be unchangeable. In other words, the object's value must never change during its lifetime.

According to this definition, immutable types, such as numbers, Booleans, and strings, are hashable. That means you can use them as dictionary keys.

There's an important exception to this statement about immutable types. Tuples are only hashable when all their items are also hashable. Remember that tuples can store mutable objects, in which case the underlying tuple won't be hashable. You'll learn more about this specific behavior in the section [Mutable Objects in Tuples](#).

In contrast, mutable types, such as lists, dictionaries, and sets, can't work as dictionary keys because they're not hashable. The reason? Their values can change during their lifetime.

**Note:** To dive deeper into the underlying differences between immutable and hashable types, check out the [Hashability vs Immutability](#) section in the tutorial [Build a Hash Table in Python With TDD](#).

Finally, dictionaries also support what's called the **union operator**, represented by the pipe symbol (`|`). This operator allows you to create a new dictionary by merging key-value pairs from two existing dictionaries. Yes, this operator has an augmented version that mutates the target dictionary in place:

Python

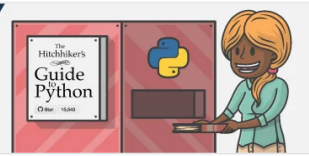
```
>>> # Regular operator
>>> inventory = {"apples": 42} | {"bananas": 24}
>>> inventory
{'apples': 42, 'bananas': 24}


>>> # Augmented operator
>>> inventory = {"apples": 42}
>>> id(inventory)
4381513984
>>> inventory |= {"bananas": 24}
>>> inventory
{'apples': 42, 'bananas': 24}
>>> id(inventory)
4381513984
```

In the first example, you use the union operator to merge two dictionaries together. The operator returns a new dictionary that gets stored in `inventory`. In the second example, you start with an existing dictionary and use the augmented union operator to merge it with a second dictionary. This operation mutates `inventory` in place, so the dictionary's identity doesn't change.

## A Python Best Practices Handbook

python-guide.org



 [Remove ads](#)

## Sets

Python's [sets](#) are another commonly used container data type. They represent an unordered container of hashable objects. Like lists and dictionaries, sets are also mutable.

If you compare sets to lists, then you'll find two main differences. First, sets don't keep their data in any specific order, so you can't use indices to access individual items. Second, sets don't keep duplicate items, while lists do.

Sets and dictionaries are closely related, though. In Python, a set works as a special dictionary that contains only keys instead of key-value pairs. Because of this characteristic, the items in a set must be hashable and unique.

Considering the above differences and similarities, you'll be able to make the following changes to the value of a given set:

- Add **new items**.
- Remove **existing items**.

Unlike lists and dictionaries that allow you to use indices and keys to change individual data items in an existing object, sets only allow mutations through specific methods:

Operation	Description
<code>a_set.add(element)</code>	Adds <code>element</code> to <code>a_set</code> .
<code>a_set.update(*others)</code>	Updates <code>a_set</code> , adding elements from one or more sets unpacked from <code>others</code> . Equivalent to <code>a_set  = other_1   other_2   ...   other_n</code> .
<code>a_set.remove(element)</code>	Removes <code>element</code> from <code>a_set</code> , raising a <code>KeyError</code> if <code>element</code> doesn't exist.
<code>a_set.discard(element)</code>	Removes <code>element</code> from <code>a_set</code> , skipping the <code>KeyError</code> if <code>element</code> doesn't exist.
<code>a_set.pop()</code>	Removes and returns an arbitrary element from <code>a_set</code> , raising a <code>KeyError</code> if the set is empty.
<code>a_set.clear()</code>	Removes all elements from <code>a_set</code> .

All these methods allow you to perform mutations on an existing set object. However, these aren't the only mutations that you can perform on sets. Note that methods like `.update()` and `.add()` will just add new elements to a given set because sets can't have duplicate elements:

Python






```
>>> fruits = {"apple", "orange", "banana"}
>>> fruits.add("lemon")
>>> fruits
{'apple', 'orange', 'lemon', 'banana'}
>>> fruits.add("orange")
>>> fruits
{'apple', 'orange', 'lemon', 'banana'}

>>> fruits.update({"grape", "orange"})
>>> fruits
{'apple', 'lemon', 'banana', 'grape', 'orange'}
```

Note that calling `.add()` with a nonexistent element like `"lemon"` effectively adds this element to the set. However, calling `.add()` with an object already stored in the underlying set doesn't cause any effect. Something similar happens with `.update()`. This method only adds new items, skipping existing ones.

The examples below show how the rest of the methods work:

```
Python 

>>> fruits.remove("apple")
>>> fruits
{'lemon', 'banana', 'grape', 'orange'}
>>> fruits.remove("mango")
Traceback (most recent call last):
  ...
KeyError: 'mango'

>>> fruits.discard("mango")
>>> fruits
{'lemon', 'banana', 'grape', 'orange'}

>>> fruits.pop()
'lemon'
>>> fruits
{'banana', 'grape', 'orange'}
```

The `.remove()` method allows you to delete a specific element from a set. If the element isn't present in the target set, then the method raises a `KeyError`. In contrast, `.discard()` has the same effect as `.remove()` but doesn't raise a `TypeError` if the item doesn't exist in the underlying set.

When it comes to `.pop()`, note that you won't know beforehand which element the method will return and remove from the set because sets are unordered data types. So, the method returns and removes an arbitrary element each time you call it.

Python's sets also implement operations from the original [mathematical sets](#), including [union](#), [intersection](#), [difference](#), and [symmetric difference](#). All these operations return a new set object rather than modify the target set in place. However, you'll find a few methods that mutate the target set in place:

Method	Description
<code>a_set.intersection_update(*others)</code>	Updates <code>a_set</code> in place, keeping only elements found in it and all others
<code>a_set.difference_update(*others)</code>	Updates <code>a_set</code> in place, removing elements found in others

Method	Description
<a href="#">a_set.symmetric_difference_update(other)</a>	Updates a_set in place, keeping only elements found in either set but not in both

These three methods represent mutations that you can perform on your set objects to transform them in place according to your needs. To learn more about them, check out the corresponding documentation.

Python sets also support some operators that allow you to perform set operations on two existing sets. Like with dictionaries and lists, these set operators also have augmented assignment versions that mutate the target set in place:

Python



```
>>> # Regular operators
>>> {"apple", "orange"} | {"banana"} # Union
{'orange', 'apple', 'banana'}
>>> {"apple", "orange"} & {"apple"} # Intersection
{'apple'}
>>> {"apple", "orange"} - {"apple", "banana"} # Difference
{'orange'}
>>> {"apple", "orange"} ^ {"apple", "banana"} # Symmetric difference
{'banana', 'orange'}

>>> # Augmented operators
>>> fruits = {"apple", "orange"}
>>> id(fruits)
4337163104
>>> fruits |= {"banana"} # Augmented union
>>> fruits
{'orange', 'apple', 'banana'}
>>> id(fruits)
4337163104

>>> fruits = {"apple", "orange"}
>>> id(fruits)
4337168928
>>> fruits &= {"apple"} # Augmented intersection
>>> fruits
{'apple'}
>>> id(fruits)
4337168928
```

In this code snippet, you can observe the regular set operators in action. They always return a new set object with the result. You can also see how the augmented union and intersection operators mutate the target set in place. Go ahead and try out augmented difference and symmetric difference for yourself.

[Real Python for Teams »](#) [Remove ads](#)

## Opposite Variations of Sets and Bytes

Python also provides two lesser-known built-in types that provide variations for the set and bytes types. For example, if you need an immutable set-like data type, then you can take advantage of the built-in [frozenset](#) type. Similarly, if you need a mutable bytes-like type, then you can use the built-in [bytearray](#) type.

In the following sections, you'll learn how to use these two built-in types in your code. To kick things off, you'll start by exploring frozen sets.

## Frozen Sets

The built-in `frozenset` data type provides an immutable version of regular sets. Frozen sets don't support methods like `.add()`, `.remove()`, and `.pop()`. In consequence, you can't modify the value of a frozen set once you've created it.

You can only perform mathematical set operations on frozen sets:

Python

```
>>> fruits = frozenset(["apple", "orange", "banana"])

>>> dir(fruits)
[
    ...
    'difference',
    'intersection',
    'isdisjoint',
    'issubset',
    'issuperset',
    'symmetric_difference',
    'union'
]
```

In Python, frozen sets only support operations like **union**, **intersection**, **difference**, and **symmetric difference**. All of these operations return new `frozenset` objects rather than mutating the underlying frozen set in place.

Like regular sets, frozen sets implement methods like [.isdisjoint\(\)](#), [.issubset\(\)](#), and [.issuperset\(\)](#). These methods return `True` or `False` according to the operation's result.

It's important to remember that the items in a frozen set must be hashable. So, if you ever need to create a set of sets, then the inner sets must be frozen sets. Frozen sets are hashable too, so you can use them as dictionary keys.

## Byte Arrays

Python's `bytes` type is immutable, as you've already learned. This type has a mutable variant that's called `bytearray`, which supports in-place changes using indices. Like the `bytes` type, `bytearray` stores binary data. If your data matches the ASCII characters set, then you can use the data directly when creating `bytearray` objects.

If you want to store a binary value over 127 in a `bytearray` object, then you must enter it using the appropriate escape sequence in the literal.

Python doesn't provide a specific literal syntax to define `bytearray` objects. If you want to create a `bytearray` object, then you need to use the `bytearray()` constructor:

Python

```
>>> bytearray(b"Hello, World!")
bytearray(b'Hello, World!')
```

Note that to create a byte array with an ASCII string, you should prefix that string with a `b` to signal that it's a bytes literal. There are several other ways to create `bytearray` objects. Check the [documentation](#) for more detail.

Remember that `bytearray` objects are mutable. They support [mutable](#) sequence operations just like lists. However, they only accept integers between 0 and 255, inclusive. That's because there are only 256 values that you can store on a single byte:

Python



```
>>> greeting = bytearray(b"Hello, World!")

>>> greeting[1] = 69
>>> greeting
bytearray(b'HEllo, World!')

>>> greeting[2] = "L"
Traceback (most recent call last):
  ...
TypeError: 'str' object cannot be interpreted as an integer

>>> greeting[2] = b"L"
Traceback (most recent call last):
  ...
TypeError: 'bytes' object cannot be interpreted as an integer

>>> greeting[2] = bytearray(b"L")
Traceback (most recent call last):
  ...
TypeError: 'bytearray' object cannot be interpreted as an integer
```

In this case, you turn the lowercase e at index 1 into an uppercase E. To do that, you use the integer 69, which represents the ASCII code for E. The next examples try to mutate the character at index 2. They fail because bytearray objects only accept integer numbers as input for mutations.

You can use the built-in [ord\(\)](#) function to get an integer representing the [Unicode](#) code point of a given character, which you can use as the target of a mutation on a byte array:

Python



```
>>> greeting[2] = ord("L")
>>> greeting
bytearray(b'HELlo, World!')

>>> greeting[3] = ord("л")
Traceback (most recent call last):
  ...
ValueError: byte must be in range(0, 256)
```

In the first example, you successfully change the character at index 2 in place. In the second example, the mutation fails because the character "л" isn't in the allowed range.

Byte arrays are kind of a hybrid of immutable bytes and mutable lists. They support the same mutations as a regular list. However, they have limitations regarding the data that you can use in these mutations.

[Online Python Training for Teams »](#) [Remove ads](#)

## Mutability in Built-in Types: A Summary

As a Python developer, you should be able to identify which built-in types are mutable and which are immutable. This distinction will become increasingly important as you go deeper in your Python learning journey.

Here's a summary of which built-in types are mutable and which are immutable:

Data Type	Built-in Class	Mutable
Numbers	int, float, complex	✗
Strings	str	✗
Tuples	tuple	✗
Bytes	bytes	✗
Booleans	bool	✗
Frozen sets	frozenset	✗
Lists	list	✓
Dictionaries	dict	✓
Sets	set	✓
Byte arrays	bytearray	✓

There are some extra considerations for you to keep in mind about some of the types in this table:

- Tuples can contain **mutable types**.
- Booleans can only hold `True` or `False`.
- Sets and frozen sets only accept **unique, hashable** items.
- Dictionaries allow **unique, hashable** keys only.

Distinguishing between mutable and immutable built-in data types will make your life more pleasant when choosing the appropriate data type for a given job. This knowledge will allow you to make informed decisions on the appropriate data type to solve a specific problem.

## Common Mutability-Related Gotchas

In Python, you may find yourself working with mutable built-in data types a lot. For example, lists and dictionaries may be present in almost every piece of Python code you write. They’re great tools! However, because of mutability, they have a few gotchas that can cause bugs and errors in your code if you’re unaware of them.

In the following sections, you’ll learn about some of the most common gotchas of mutable data types in Python. You’ll also learn how to avoid them and make your code more robust and reliable.

## Aliasing Variables

Whenever you assign an existing variable to a new one, you create an **alias** for the original variable. Variable aliases share a reference to the same memory address, pointing to the same object or piece of data:

Python





```
>>> number = 7
>>> lucky_number = number
>>> id(number)
4334100904
>>> id(lucky_number)
4334100904
```

Python variables don't hold the data itself but references to the memory address where the data is stored. When you create an alias of an existing variable, both variables will hold the same reference.

When you create aliases of variables pointing to immutable objects like numbers, strings, and tuples, you don't have to worry about mutations. With mutable types, mutation on a given alias affects the rest of the aliases:

Python



```
>>> number = 42
>>> another_number = number
>>> number += 1
>>> number
43
>>> another_number
42

>>> fruits = ["apple"]
>>> another_fruits = fruits
>>> fruits += ["banana"]
>>> fruits
['apple', 'banana']
>>> another_fruits
['apple', 'banana']
```

You can't change the value of an immutable object without changing the object's identity. So, when it comes to immutable objects, creating aliases is completely safe. In contrast, aliases of mutable objects are affected by mutations in other aliases.

## Mutatating Arguments in Functions

When creating a [function](#) in Python, you may need it to accept arguments in the call. These arguments will work as the function's input and can be the starting point for the function's specific computation.

If you call a function with a [global variable](#) as an argument, then the function's formal parameter becomes an alias of the global variable. Both will point to the same memory address and object.

When you call a function with a mutable object as an argument, the function may perform mutations on that object. These mutations will affect the original object outside the function. This behavior can be a source of subtle bugs.

Therefore, when coding functions that receive mutable objects as arguments, you must carefully consider whether the caller expects to change the arguments passed.

For example, say you want to write a function that takes a list of numbers and returns a list of square values. You can think of a function like the following:

Python




```
>>> def squares_of(numbers):  
...     for i, number in enumerate(numbers):  
...         numbers[i] = number ** 2  
...     return numbers  
...  
  
>>> sample = [2, 3, 4]  
>>> squares_of(sample)  
[4, 9, 16]  
>>> sample  
[4, 9, 16]
```

Your `squares_of()` function takes a list of numbers as an argument. Inside the function, you take the `numbers` argument and change it in place to contain square values based on the original input data.

Because you mutated the argument, these mutations affect the input data. Now `sample` contains square values rather than the original data. This may be the wrong final result because you're losing your original data.

To prevent this kind of issue, you must avoid mutating arguments in your functions:

Python 

```
>>> def squares_of(numbers):  
...     result = []  
...     for number in numbers:  
...         result.append(number ** 2)  
...     return result  
...  
  
>>> sample = [2, 3, 4]  
>>> squares_of(sample)  
[4, 9, 16]  
>>> sample  
[2, 3, 4]
```

In this new implementation of `squares_of()`, you don't run any mutations on its `numbers` argument. Instead, you create a new `list` object to hold the transformed data. Now the original data is safe, as you can confirm by accessing `sample` after the function call.

**Note:** The above implementation of `squares_of()` has another cool advantage compared to the first implementation. Your function can now accept any [iterable](#) holding numbers. It's not limited to `list` objects.

In contrast, if you pass an immutable object as an argument to a function, then you can reassign that argument inside the function. Changes won't affect the original input object:

Python 

```
>>> counter = 0
>>> id(counter)
4334100680

>>> def increment(value):
...     value += 1
...     print(id(value))
...     return value
...

>>> increment(counter)
4334100712
1
>>> increment(counter)
4334100712
1

>>> counter
0
>>> id(counter)
4334100680
```

In this example, you use `counter` as an argument to `increment()`. The function receives a reference to the content of `counter` and assigns it to `value`, which becomes a local alias of `counter`. Because both aliases point to an immutable object, reassigning the argument inside the function doesn't affect the global object. It just creates a new local object with the same name.

**Note:** You can use Python's [global](#) statement if you need the assignment to a global variable inside a function to affect the variable outside.

For example, if you want your `increment()` function to update the value of `counter` outside the function, then you can do something like this:

Python



```
>>> counter = 0

>>> def increment():
...     global counter
...     counter += 1
...

>>> increment()
>>> increment()
>>> counter
2
```

This time, `increment()` doesn't take any arguments. Instead, it uses the global `counter` variable to perform its computation. Note how changes affect the global variable itself.

In general, you should avoid mutating arguments inside your functions. A function should provide its result through an appropriate return value rather than through an altered argument that affects the value of objects in the caller's namespace.

## Using Mutable Default Values

[Optional arguments](#) with default values are a great feature of Python functions. Default argument values allow you to write powerful, flexible functions. A common recommendation in Python is to avoid mutable objects as default argument values. Why?

Most Python developers will argue that using mutable types as default argument

values is a dangerous practice because the default value is defined when the Python interpreter first parses the function. This means that if you call the function multiple times while relying on the default value, then you'll be using the same object every time. The function becomes [stateful](#). It retains state between calls.

The following function—or some variation of it—is the classic example to uncover this behavior:

Python



```
>>> def append_to(item, target=[]):
...     target.append(item)
...     return target
... 
```

This toy function appends `item` to the end of `target`, which defaults to an empty list. At first glance, it may seem that consecutive calls to `append_to()` will return single-item lists like in the following hypothetical code:

Python



```
>>> # What you might expect to happen:
>>> append_to(1)
[1]
>>> append_to(2)
[2]
>>> append_to(3)
[3]
```

Because Python defines the default argument value when it first parses the function and doesn't overwrite it in every call, you'll be working with the same instance every time. Then, you don't get the above output. Instead, you get the following output:

Python



```
>>> # What actually happens:
>>> append_to(1)
[1]
>>> append_to(2)
[1, 2]
>>> append_to(3)
[1, 2, 3]
```

Wow! The list remembers the data between calls. This happens because you're relying on the same empty list used as the default value of `target`. This issue with mutable default argument values may be one of the reasons why [None](#) is such a common default value in Python functions.

The classic solution to the issue looks something like this:

Python



```
>>> def append_to(item, target=None):
...     if target is None:
...         target = []
...     target.append(item)
...     return target
... 
```

```
>>> append_to(1)
[1]
>>> append_to(2)
[2]
>>> append_to(3)
[3]
```

Great! You've solved the issue. Now your function returns single-item lists as expected. Now the function doesn't retain state between calls.

Even though using mutable default argument values is widely considered a Python gotcha, you may find situations in which this rather weird behavior may be useful.

For example, say that you want to create a function that computes the cumulative mean of a data stream. In this case, you'll need a way to make your function remember the previous data points between calls. To do this, you can take advantage of a mutable default argument like in the following code:

Python



```
>>> def cumulative_mean(value, sample=[]):
...     sample.append(value)
...     return sum(sample) / len(sample)
...

>>> cumulative_mean(10)
10.0
>>> cumulative_mean(12)
11.0
>>> cumulative_mean(11)
11.0
>>> cumulative_mean(13)
11.5
>>> cumulative_mean(14)
12.0
```

Wow! That was nice! It turns out that mutable default argument values aren't completely bad. Well, that may be true. But you should still avoid them as much as you can. Make your functions safer by returning an appropriate value, and avoid mutating the function's arguments in its body.

## Making Copies of Lists

Python's lists are mutable. Because of this, it's often useful to make a copy of a given list before performing operations that would mutate the list in place. You can copy a list in at least two ways. You can make:

- A **shallow copy**, which you create using the slicing operator (`[ : ]`), the `.copy()` method, or the `copy.copy()` function
- A **deep copy**, which you can create using the `copy.deepcopy()` function

When you make a shallow copy of an existing list, you create a new list of objects with a different identity. However, the internal components or data items in the new list are just aliases of those in the original list. On the other hand, if you make a deep copy, then you create a completely new copy of the original list.

Here are some examples that illustrate the described behavior:

Python



```
>>> import copy

>>> matrix = [[1, 2, 3], [4, 5, 6], [6, 7, 8]]

>>> shallow_copy = copy.copy(matrix) # Same as: matrix.copy() or matrix[:]
>>> deep_copy = copy.deepcopy(matrix)

>>> id(matrix)
4334079424

>>> id(shallow_copy)
4381951232

>>> id(deep_copy)
4369317568

>>> id(matrix[0]) == id(shallow_copy[0])
True
>>> id(matrix[0]) == id(deep_copy[0])
False

>>> matrix[1][1] = 555

>>> matrix
[[1, 2, 3], [4, 555, 6], [6, 7, 8]]


>>> shallow_copy
[[1, 2, 3], [4, 555, 6], [6, 7, 8]]

>>> deep_copy
[[1, 2, 3], [4, 5, 6], [6, 7, 8]]
```

In this example, `matrix` and its two copies are completely independent objects. They have different identities. The values stored in `shallow_copy` share the same identity as those in `matrix`. The behavior of shallow copies can save a lot of memory when your lists are quite large. You won't need to use the same amount of memory to store copies of all the data. You just need memory to store the `list` objects.

In the meantime, the values in `deep_copy` have different identities from those in the original object, `matrix`. The inner lists are now completely different objects. They don't have the same identity. Because of this behavior, deep copies may duplicate the memory usage because they need to store an entire copy of the data.

An important detail to note is that mutations on a shallow copy don't affect the original list and vice versa:

```
Python 

>>> matrix[1] = [444, 555, 666]
>>> matrix
[[1, 2, 3], [444, 555, 666], [6, 7, 8]]
>>> shallow_copy
[[1, 2, 3], [4, 555, 6], [6, 7, 8]]

>>> shallow_copy[0] = [111, 222, 333]
>>> shallow_copy
[[111, 222, 333], [4, 555, 6], [6, 7, 8]]
>>> matrix
[[1, 2, 3], [444, 555, 666], [6, 7, 8]]
```

In this example, you mutate `matrix` directly by replacing the list stored at index 1. This mutation doesn't affect the data stored in `shallow_copy`. From this point on, that item doesn't share the memory address that's in `matrix` and `shallow_copy`. The second example works similarly.



## Getting None From Mutator Methods

You'll find an important behavior difference between methods of immutable objects and methods of mutable objects. For example, string formatting methods always return a new string object because they can't perform their formatting in place. You'll typically use these methods in assignments:

Python



```
>>> greeting = "Hello, World!"
>>> greeting = greeting.upper()
>>> greeting
'HELLO, WORLD!'
```

The `.upper()` method, like all the string formatting methods, returns a new string. You must use an assignment to store the new string in a variable if you want to keep a reference to it.

You can take advantage of the above behavior when you need to chain method calls, which is only possible when the method returns the same kind of object:

Python



```
>>> text = '<html lang="en">'

>>> text.removeprefix("<").removesuffix(">").upper().center(20)
'    HTML LANG="EN"    '
```

In this example, you create a chain of method calls that preforms a sequence of transformations in the original string contained in `text`. Note that you can use this technique with methods that return new objects, like the string formatting methods do.

If you're starting with Python, then you may be getting used to this behavior and then try to do the same with list objects:

Python



```
>>> numbers = [3, 4, 2, 6, 1]
>>> sorted_numbers = numbers.sort()
>>> print(sorted_numbers)
None
```

Wow! What just happened? The methods that change a mutable object in place return `None`. This behavior may be a common source of confusion and errors. So, keep it in mind when working with mutator methods.

## Storing Mutable Objects in Tuples

Python's `tuple` is quite an interesting data type. Tuples are immutable by definition, so you'd probably conclude that you can't change the value of a tuple in place. That's a half-truth. The problem is that tuples can hold mutable objects.

Consider the following example, in which you use a tuple to store a color:

Python



```
>>> red = ("RED", [255, 0, 0])

>>> red[0] = "red"
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment

>>> red[1] = [0, 0, 0]
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment

>>> red[1][0] = 0
>>> red
('RED', [0, 0, 0])
```

You can't mutate the items contained in `red`. However, since the second item is a list object, you can use a second index to mutate individual items in that list, as you did in the last assignment above.

If the items in a tuple are mutable, then you'll be able to change them even if the tuple itself is immutable. In other words, the immutability of Python tuples refers to the references it directly holds. It doesn't extend to the referenced objects themselves.

In general, putting mutable objects in tuples is a bad idea. It kind of breaks the tuple's immutability. It also makes your tuples unhashable, which prevents you from using them as dictionary keys:

```
Python 

>>> apple = ["Apple", "123"]
>>> orange = ["Orange", "456"]

>>> prices = {
...     ("New York", apple): 1.00,
...     ("New York", orange): 1.50,
...     ("Chicago", apple): 1.25,
...     ("Chicago", orange): 1.75
... }
Traceback (most recent call last):
  ...
TypeError: unhashable type: 'list'
```

In this example, you've tried to use tuples to represent the keys of your `prices` dictionary. The idea is that your dictionary keys store the city and the product, while the dictionary values will represent the keys. However, you've used list objects to represent your products, with each list containing the product's name and ID.

Since lists aren't hashable, they make their containing tuples unhashable as well, so building your dictionary fails when using these keys. To work around this issue, you can turn your products into tuples.

Whenever possible, avoid storing mutable objects in your tuples unless you explicitly need to take advantage of this specific behavior of Python tuples. Always keep in mind that your code may continue evolving, and at some point, this behavior can cause undesired issues.

## Concatenating Many Strings

Sometimes, when working with strings, you'll need to create new strings by joining several smaller strings together. To [concatenate](#) two or more string objects, you can use the concatenation operator (+) like in the following code:

Python



```
>>> "Hello" + "," + " " + "World" + "!"  
'Hello, World!'
```

This kind of expression may provide a quick solution to the problem of concatenating a few strings together.

However, concatenating many strings using the plus operator—or its augmented variation (`+=`)—would imply creating several temporary string objects because you can't perform the mutation in place. All these intermediate strings will be discarded when added to the next string.

This behavior can represent a considerable waste of memory resources when the number of intermediate strings is large. It can also imply a processing time burden because creating new string objects takes time.

If you ever need to concatenate several string objects to build a final string, then use the [.join\(\)](#) method with an appropriate separator string:

Python



```
>>> "".join(["Hello", ",", " ", "World", "!"])  
'Hello, World!'
```

The `.join()` method takes an iterable of string objects and efficiently joins them together in a final string. This join uses a specific separator string that you must use to call the method itself. In the above example, that separator is an empty string.

## Mutability in Custom Classes

When you're creating your own classes in Python, you must keep in mind that they're mutable by default. Yes, instances of user-defined classes are mutable too. You can mutate them in several ways. For example, you can:

- **Add** or **delete** class and instance attributes dynamically
- **Change** or **reassign** the value of class and instance attributes

In the following sections, you'll learn how mutability operates on your custom classes and their instances in Python.

## Mutability of Classes and Instances

To demonstrate why classes and instances are mutable by default in Python, consider the following `User` class definition:

Python



```
>>> class User:  
...     pass  
...
```

This class is pretty minimal. It only defines `User` with the [class](#) keyword and uses a [pass](#) statement as the class body. If you use the built-in [dir\(\)](#) function to [introspect](#) the class, then you'll get a relatively long list of attributes that Python added under the hood:


Python



```
>>> dir(User)
[
    '__class__',
    '__delattr__',
    '__dict__',
    ...
    '__weakref__'
]
```

User inherits all these attributes from the [object](#) class, which is the default parent class of every Python class. Even though User has many inherited attributes, the class itself isn't that useful. It doesn't have custom methods and attributes, so it has minimal functionality and state.

To make User more feature-rich, you can provide [class attributes](#) and methods dynamically using **dot notation** and an assignment like in `User.attr = value`. Here's an example:

```
Python 
>>> User.company = "Example" # Class attribute

>>> def __init__(self, name, job):
...     self.name = name
...     self.job = job
...

>>> User.__init__ = __init__ # Instance method

>>> dir(User)
[
    '__class__',
    '__delattr__',
    '__dict__',
    '__dir__',
    ...
    '__weakref__',
    'company'
]
```

After running this code, your class will have a new class attribute called `.company`. The value of a class attribute will be common to all the instances of that class.

User also has a custom implementation of `__init__()`, the [instance initializer](#). This new `__init__()` provides your class with two new instance attributes, `.name` and `.job`. You can initialize these attributes by passing appropriate arguments to the [class constructor](#) at instantiation:

```
Python 
>>> john = User("John Doe", "Python Dev")
>>> john.name
'John Doe'
>>> john.job
'Python Dev'

>>> dir(john)
[
    '__class__',
    '__delattr__',
    '__dict__',
    ...
    '__weakref__',
    'company',
    'job',
    'name'
]
```

Cool! The constructor of `User` now accepts the `name` and `job` arguments, which are automatically stored in `.name` and `.job` under the current instance. You can access the values of these attributes through the instance using dot notation.

Being able to add attributes to your classes and instances dynamically, as you've done in the above examples, implies that classes and instances are mutable. You can change their value in place, without changing the class or instance identity.

Custom classes and their instances are mutable because they keep their attributes and methods in a special dictionary called `__dict__`. Both the class and the instance will have a `__dict__` dictionary:

Python



```
>>> User.__dict__
mappingproxy({
  '__module__': '__main__',
  '__dict__': <attribute '__dict__' of 'User' objects>,
  '__weakref__': <attribute '__weakref__' of 'User' objects>,
  '__doc__': None,
  'company': 'Example',
  '__init__': <function __init__ at 0x102b50860>,
  '__getattr__': <slot wrapper '__getattr__' of 'object' ob...
})

>>> john.__dict__
{'name': 'John Doe', 'job': 'Python Dev'}
```

The `User.__dict__` dictionary is a private [namespace](#) for the class object. There you'll find class attributes and methods, such as `.company` and `__init__()`, respectively. Similarly, `john.__dict__` holds instance attributes and their values for the current instance, `john`.

Finally, you can also delete an attribute from a class or an instance using the `del` statement:

Python



```
>>> del john.job
>>> john.__dict__
{'name': 'John Doe'}
```

In this example, you use the `del` statement to remove the `.job` attribute from `john`. When you inspect the content of the instance `__dict__` dictionary, you see that it doesn't have the `job` key any longer.

## Mutability of Attributes

The second dimension of class and instance mutability is the possibility of changing the value of class and instance attributes, by either mutating them or reassigning them. For example, in the `Book` class below, there's nothing to stop you from changing the `.title` of an existing instance like `harry_potter`:

Python



```
>>> class Book:
...     def __init__(self, title):
...         self.title = title
...

>>> harry_potter = Book("Harry Potter")

>>> harry_potter.title = "Harry Potter and the Sorcerer's Stone"
>>> harry_potter.title
"Harry Potter and the Sorcerer's Stone"
```

With this assignment, you mutated the current value of `harry_potter` without altering the object's identity. So, this is another mutability dimension of Python classes. In the following section, you'll learn about some of the most common techniques and tools that you can use to control mutability in your own classes.

## Techniques to Control Mutability in Custom Classes

When creating your own classes, you have to take extra actions in order to make them and their instances immutable or partially immutable. In this section, you'll focus on techniques that allow you to control the mutability of instances and instance attributes.

Python offers several techniques and tools that you can use to prevent people from changing the value of instance attributes or even from adding new instance attributes dynamically. You can use some of the following approaches:

- Defining a `__slots__` class attribute
- Providing custom `__setattr__()` and `__delattr__()` methods
- Using [read-only properties](#)
- Relying on [descriptors](#) with an appropriate `__set__()` method
- Using an immutable class, such as a [named tuple](#) or a [data class](#)

In the following sections, you'll write some examples of how to use these techniques in your code to prevent your users from changing the value of instances and instance attributes.

### Defining a `__slots__` Class Attribute

Python allows you to create classes that prevent the addition of new instance attributes. To do this, you can use the `__slots__` class attribute, which lets you specify the allowed attributes for a given object:

Python

```
>>> class Book:
...     __slots__ = ("title",)
...     def __init__(self, title):
...         self.title = title
...

>>> harry_potter = Book("Harry Potter")
>>> harry_potter.title
'Harry Potter'

>>> harry_potter.author = "J. K. Rowling"
Traceback (most recent call last):
...
AttributeError: 'Book' object has no attribute 'author'
```

If you add a `__slots__` attribute to a custom class, then you won't be able to add new attributes to the instances of that class dynamically. That's why you can't add the `.author` attribute to your `harry_potter` instance in the above example. If you try to do it, then you get an [AttributeError](#).

**Note:** Although `__slots__` can hold a list object, you should use a tuple object. Even if changing the list in `__slots__` after the class body is processed has no effect, it'd be misleading to use a mutable sequence there.

Unfortunately, the `__slots__` attribute doesn't prevent you from adding new class



attributes and methods dynamically:

Python



```
>>> Book.publisher = "Example"
>>> dir(Book)
[
    '__class__',
    '__delattr__',
    '__dir__',
    '...',
    '__subclasshook__',
    'publisher',
    'title'
]
```

Even though `Book` has a `__slots__` attribute holding a tuple of allowed instance attributes, you can still add new class attributes and methods dynamically to your `Book` class.

Even if you define a `__slots__` attribute, you won't be able to prevent your users from removing allowed attributes from your class:

Python



```
>>> del harry_potter.title

>>> harry_potter.title
Traceback (most recent call last):
  ...
AttributeError: 'Book' object has no attribute 'title'
```

In this example, the `del` statement removes the `.title` attribute from your `harry_potter` object. If you want to prevent this behavior, then you'll need to use another technique that consists of writing a `__delattr__()` method. Keep reading to learn about it.

## Providing Custom `__setattr__()` and `__delattr__()` Methods

Python automatically calls the `__setattr__()` method when you use an attribute in an assignment statement. Similarly, Python calls the `__delattr__()` method when you run a `del` statement to remove a given attribute.

You can provide your own implementations of these methods to prevent these mutations from happening in your classes. Here's an example of how this technique works in practice:

Python

```
# immutable.py

class Immutable:
    def __init__(self, value):
        super().__setattr__("value", value)

    def __setattr__(self, name, attr_value):
        raise AttributeError(f"can't set attribute '{name}'")

    def __delattr__(self, name):
        raise AttributeError(f"can't delete attribute '{name}'")
```

This class provides custom implementations of `__setattr__()` and `__delattr__()`. The former raises an `AttributeError` when you try to change the value of an existing attribute or when you try to add a new one.

Similarly, the `.__delattr__()` method raises an `AttributeError` when you try to delete the `.value` attribute using the `del` statement.

Here's how your `Immutable` class works in practice:

Python



```
>>> from immutable import Immutable

>>> gravity = Immutable(9.78)
>>> gravity.value
9.78

>>> gravity.value = 9.832
Traceback (most recent call last):
  ...
AttributeError: can't set attribute 'value'

>>> gravity.unit = "m/s²"
Traceback (most recent call last):
  ...
AttributeError: can't set attribute 'unit'

>>> del gravity.value
Traceback (most recent call last):
  ...
AttributeError: can't delete attribute 'value'
```

Wow! This technique is amazing! You have a class that doesn't accept changes to its `.value` attribute.

In addition, the class prevents you from adding new attributes dynamically. That's why you can add a `.unit` attribute in the example above. The class even prevents you from removing existing attributes. Isn't that cool?

It's important to point out that if you use mutable objects as the value of an immutable class like the above, then you won't be able to prevent mutations on that value:

Python



```
>>> fruits = Immutable(["apple", "orange", "banana"])
>>> fruits.value
['apple', 'orange', 'banana']

>>> fruits.value = 42
Traceback (most recent call last):
  ...
AttributeError: can't set attribute 'value'

>>> fruits.value.append("lemon")
>>> fruits.value
['apple', 'orange', 'banana', 'lemon']

>>> del fruits.value[0]
>>> fruits.value
['orange', 'banana', 'lemon']
```

In this example, you can't reassign `.value` to hold the number 42 because your class is immutable in that sense. However, you can change the current value of this attribute because the contained object is mutable.

In this regard, your `Immutable` class behaves similarly to Python tuples, which are immutable by definition but can store mutable objects that you can mutate at will.

## Using Read-Only Properties and Descriptors

You can turn the instance attributes of a class into read-only properties or read-only descriptors to prevent your users from changing the value of those attributes. For example, say that you need to write a `Point` class with two attributes representing the [Cartesian](#) coordinates:

Python

```
# point.py

class Point:
    def __init__(self, x, y):
        self._x = x
        self._y = y

    @property
    def x(self):
        return self._x

    @property
    def y(self):
        return self._y

    def __repr__(self):
        return f'{type(self).__name__}(x={self.x}, y={self.y})'
```

In this example, your `Point` class has two properties, `.x` and `.y`. These properties only have [getter methods](#). They don't have setter methods. Because of this, your `.x` and `.y` coordinates are read-only. In other words, you can't change the value originally assigned to them:

Python



```
>>> from point import Point

>>> point = Point(21, 42)
>>> point.x
21
>>> point.y
42

>>> point.x = 7
Traceback (most recent call last):
  ...
AttributeError: property 'x' of 'Point' object has no setter

>>> point.y = 14
Traceback (most recent call last):
  ...
AttributeError: property 'y' of 'Point' object has no setter
```

This way, you've converted `.x` and `.y` into read-only attributes, which contributes to your goal of trying to control how your users can mutate your class.

**Note:** In the previous example, you can still access and change the underlying non-public attributes, `._x` and `._y`:

Python



```
>>> point._x = 555
>>> point
Point(x=555, y=42)
```

Python doesn't have private attributes, which means that you can change any non-public attributes. However, keep in mind that these attributes aren't intended to be changed from outside the class.

You can also use a [descriptor](#) to get a similar result as in the above example. Your descriptor should implement a `.__set__()` method that prevents your user from changing the value of a coordinate.

Here's an alternative version of `Point` using a descriptor to implement its coordinates:

Python

```
# point.py

class Coordinate:
    def __set_name__(self, owner, name):
        self._name = name

    def __get__(self, instance, owner):
        return instance.__dict__[f"_{self._name}"]

    def __set__(self, instance, value):
        raise AttributeError(f"can't set attribute '{self._name}'")

class Point:
    x = Coordinate()
    y = Coordinate()

    def __init__(self, x, y):
        self._x = x
        self._y = y

    def __repr__(self):
        return f"{type(self).__name__}(x={self.x}, y={self.y})"
```

`Coordinate` is a descriptor class. It implements a `.__get__()` method that returns the value of the coordinate at hand. To do that, the method takes advantage of the instance `.__dict__` dictionary.

The class also provides a `.__set__()` method that raises an `AttributeError` exception if someone tries to change the value of the underlying coordinate.

The final step is to define two class attributes in `Point` to manage the coordinates using the descriptor class. Here's how `Point` works now:

Python



```
>>> from point import Point

>>> point = Point(21, 42)
>>> point.x
21
>>> point.y
42

>>> point.x = 7
Traceback (most recent call last):
  ...
AttributeError: can't set attribute 'x'

>>> point.y = 14
Traceback (most recent call last):
  ...
AttributeError: can't set attribute 'y'
```

Cool! Now your `.x` and `.y` coordinates are read-only again. You can access the coordinates at any time using dot notation. However, you can't change the value of any of your current coordinates. If you try to do it, then you get an `AttributeError`.

## Building Immutable Tuple-Like Objects


Sometimes you need a quick way to create immutable tuple-like objects to store certain data, and you need to be able to access the data using readable field names instead of integer indices. If that's your case, then you can use a named tuple.

You'll have at least two different ways to create a named tuple:

1. Using the `namedtuple()` factory function from the `collections` module
2. Subclassing the `NamedTuple` class from the `typing` module

The `namedtuple()` function allows you to create tuple subclasses with named fields that you can use to access data items using descriptive names rather than indices.

Here's an example of how to write a more compact version of your `Point` class:

```
Python 

>>> from collections import namedtuple

>>> Point = namedtuple("Point", "x y")
>>> point = Point(21, 42)
>>> point
Point(x=21, y=42)

>>> point.x
21
>>> point.y
42
```

The call to `namedtuple()` builds and returns a new class object. The class's name is `"Point"`, as the first argument to `namedtuple()` defines. This class will have two attributes, `.x` and `.y`, which you provide in the second argument to `namedtuple()` using a string with space-separated attribute names.

Finally, you assign the class object to the `Point` variable, which you typically name after the class's name. Because this variable is an alias of a class object, you can use it as a class constructor. That's what you do in the statement `point = Point(21, 42)`. Now `point` is an instance of your tuple-like class.

Note that you can access the attributes `.x` and `.y` using dot notation as usual. Now, what would happen if you tried to change an attribute's value or add a new attribute

dynamically?

Here's the answer to that question:

Python

```
>>> point.x = 7
Traceback (most recent call last):
  ...
AttributeError: can't set attribute

>>> point.y = 14
Traceback (most recent call last):
  ...
AttributeError: can't set attribute

>>> point.z = 100
Traceback (most recent call last):
  ...
AttributeError: 'Point' object has no attribute 'z'
```

Named tuples are subclasses of `tuple` with named fields. Because of this, they're also immutable. You can neither modify the value of their attributes nor add new attributes dynamically.

**Note:** Like their parent, `tuple`, named tuples can store mutable objects like lists and dictionaries. Keep this in mind when using them.

You can also add methods to your named tuple classes. However, to do that, you need to use inheritance. For example, say that you want to add a method to your `Point` class to compute the [Euclidean distance](#) between the current point and any other point. In that case, you can do something like this:

Python

```
# point.py

import math
from collections import namedtuple

class Point(namedtuple("Point", "x y")):
    __slots__ = ()

    def distance(self, other: "Point") -> float:
        return math.dist((self.x, self.y), (other.x, other.y))
```

`Point` now inherits from your original named tuple class. In the subclass definition, you set `__slots__` to an empty tuple, which prevents the automatic creation of an instance `__dict__` and, therefore, the addition of new attributes.

Here's how `Point` works now:

Python

```
>>> from point import Point

>>> origin = Point(0, 0)
>>> point = Point(4, 3)
>>> point.distance(origin)
5.0

>>> point.z = 100
Traceback (most recent call last):
  ...
AttributeError: 'Point' object has no attribute 'z'
```



Now you can use the `.distance()` method to compute the distance between two instances of `Point`. You can add as many methods as you need to your class. However, you must ensure those methods don't attempt to change the value of `.x` or `.y`. If you try to change them, then you'll get an `AttributeError` because your class is immutable in that sense too.

Named tuples that you create using the `namedtuple()` function look great, and they are. However, if you're into [type hints](#), then you should use `typing.NamedTuple`, which will allow you to provide type hints for your attributes:

Python

```
# point.py

import math
from typing import NamedTuple

class Point(NamedTuple):
    x: float
    y: float

    def distance(self, other: "Point") -> float:
        return math.dist((self.x, self.y), (other.x, other.y))
```

In this alternative version of `Point`, you've declared that the `.x` and `.y` attributes are both floating-point numbers, which is valuable information for anyone reading your code. Note that you haven't added a `__slots__` attribute to your class. `NamedTuple` takes care of preventing the addition of new attributes for you.

Your version of `Point` works just like the old one:

Python

```
>>> from point import Point

>>> origin = Point(0, 0)
>>> point = Point(4, 3)
>>> point.distance(origin)
5.0

>>> point.z = 100
Traceback (most recent call last):
  ...
AttributeError: 'Point' object has no attribute 'z'
```

Using `typing.NamedTuple` instead of `collections.namedtuple()` can make your code more readable by providing type hints. It also guarantees that your class is immutable by preventing you from adding new attributes and from changing the value of existing ones.

## Creating Immutable Data Classes

Python's data classes provide a straightforward, quick, and efficient way to write custom classes. Data classes are essentially [code generators](#) that write a lot of boilerplate code for you. This auto-generated code covers your back in many ways.

You'll have accurate, functional implementations of many special methods, such as `__init__()`, `__repr__()`, `__eq__()`, and `__hash__()`. These methods will ensure your class supports important functionalities, such as string representation, comparison, hashability, and more.

Data classes also support type hints, default attribute values, and methods, which makes them a great tool. To create a data class, you need to use the `@dataclass` [decorator](#) from the `dataclasses` module:

Python



```
>>> from dataclasses import dataclass

>>> @dataclass
... class Color:
...     red: int
...     green: int
...     blue: int
...

>>> color = Color(255, 0, 0)
>>> color
Color(red=255, green=0, blue=0)

>>> color.green = 128
>>> color
Color(red=255, green=128, blue=0)
```

This `Color` class looks great! It provides a cool string representation and many other features that you can explore for yourself. However, it has a potential issue. You can change the attributes of an existing color, which means turning it into a different one.

To solve this issue, you can make your data class immutable by setting the `frozen` argument of `@dataclass` to `True`:

Python



```
>>> @dataclass(frozen=True)
... class Color:
...     red: int
...     green: int
...     blue: int
...

>>> color = Color(255, 0, 0)
>>> color
Color(red=255, green=0, blue=0)

>>> color.green = 128
Traceback (most recent call last):
...
dataclasses.FrozenInstanceError: cannot assign to field 'green'
```

You can make your data classes mutable or immutable depending on the value that you pass to the `frozen` argument, which defaults to `False`. In this example, you've turned the `Color` class into a more consistent class that prevents users from changing any of its attributes.

When you freeze a data class, that class won't allow the value of its attributes to be changed. However, if one of the values is mutable, then it's possible for it to be changed in place. So, keep an eye on this potential issue.

## Conclusion

Now you have a deep understanding of how mutable and immutable data types internally work in Python. In general, mutable types allow in-place changes to their values, while immutable types don't. Mutability is a fundamental feature that really influences which data types are appropriate for each specific problem.

Learning about mutable and immutable data types is, therefore, a vital skill to have as a Python programmer.

In this tutorial

**Find Your Dream Python Job**

[pythonjobshq.com](https://pythonjobshq.com)



- How **mutability** and **immutability** work under the hood in Python
- Which immutable and mutable **built-in data types** exist in Python
- What common **mutability-related gotchas** can affect your code and how to prevent them
- What **techniques** allow you to control mutability in **custom classes**

© 2012–2023 Real Python · [Newsletter](#) · [Podcast](#) · [YouTube](#) · [Twitter](#) · [Facebook](#) · [Instagram](#) ·

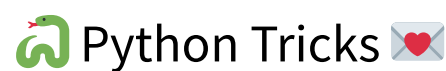
[Python Tutorials](#) · [Search](#) · [Privacy Policy](#) · [Energy Policy](#) · [Advertise](#) · [Contact](#)

♥ Happy Pythoning!

With all this knowledge, you're better prepared to make informed decisions on whether to use a mutable or immutable type to solve a specific coding problem in your day-to-day programming. You're also ready to start writing better classes yourself.

**Free Sample Code:** [Click here to download the free sample code](#) that you'll use to explore mutable vs immutable data types in Python.

Mark as Completed



Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

## About Leodanis Pozo Ramos



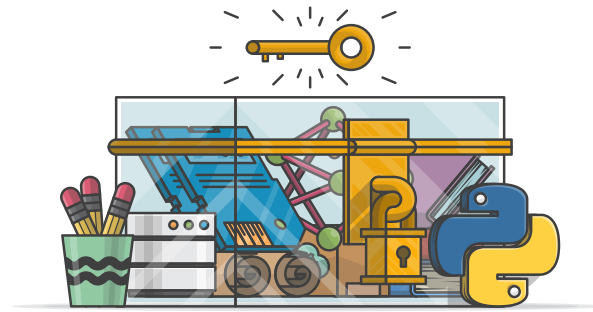
Leodanis is an industrial engineer who loves Python and software development. He's a self-taught Python developer with 6+ years of experience. He's an avid technical writer with a growing number of articles published on Real Python and other sites.

» [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](#)[Geir Arne](#)[Kat](#)

## Master Real-World Python Skills With Unlimited Access to Real Python

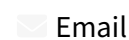


**Join us and get access to thousands of  
tutorials, hands-on video courses, and  
a community of expert Pythonistas:**

[Level Up Your Python Skills »](#)

### What Do You Think?

**Rate this article:**

[LinkedIn](#)[Twitter](#)[Facebook](#)[Email](#)

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.

**Commenting Tips:** The most useful comments are those written with the goal of learning from or helping out other students. [Get tips for asking good questions](#) and [get answers to common questions in our support portal](#).

Looking for a real-time conversation? Visit the [Real Python Community Chat](#) or join the next [“Office Hours” Live Q&A Session](#). Happy Pythoning!

### Keep Learning

Related Tutorial Categories: [data-structures](#) [intermediate](#) [python](#)