

# Lists vs Tuples in Python

by Leodanis Pozo Ramos  Jan 26, 2025  28m  32 Comments

 [basics](#) [python](#)

Mark as Completed

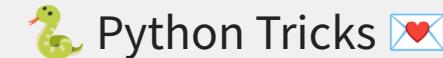


Share 

## Table of Contents

- [Getting Started With Python Lists and Tuples](#)
  - [Creating Lists in Python](#)
  - [Creating Tuples in Python](#)
- [Exploring Core Features of Lists and Tuples](#)

— FREE Email Series —



```
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.

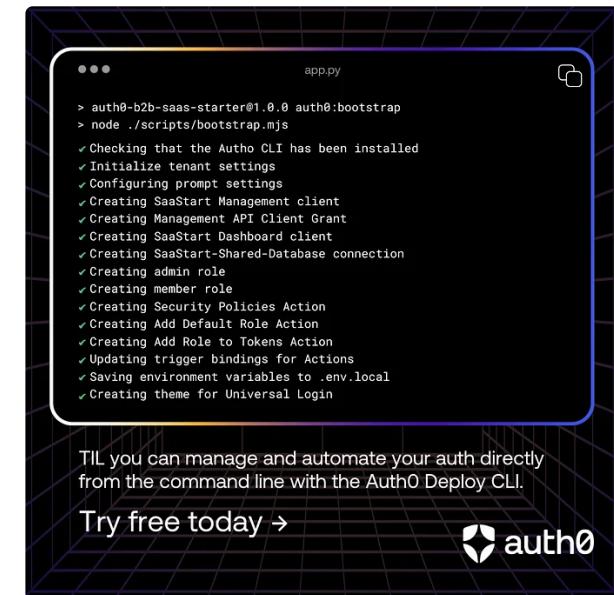
 [Browse Topics](#)  [Guided Learning Paths](#)

 [Basics](#)  [Intermediate](#)  [Advanced](#)

[ai](#) [algorithms](#) [api](#) [best-practices](#) [career](#)  
[community](#) [databases](#) [data-science](#)  
[data-structures](#) [data-viz](#) [devops](#) [django](#)  
[docker](#) [editors](#) [flask](#) [front-end](#) [gamedev](#)  
[gui](#) [machine-learning](#) [news](#) [numpy](#)

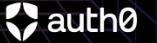
- Lists and Tuples Are Ordered Sequences
- Lists and Tuples Can Contain Arbitrary Objects
- Lists and Tuples Can Be Indexed and Sliced
- Lists and Tuples Can Be Nested
- Lists Are Mutable, Tuples Are Immutable
- Lists Have Mutator Methods, Tuples Don't
- Using Operators and Built-in Functions With Lists and Tuples
- Packing and Unpacking Lists and Tuples
- Using Lists vs Tuples in Python
- Conclusion
- Frequently Asked Questions

projects python stdlib testing tools  
web-dev web-scraping



```
... app.py
> auth0-b2b-saas-starter@1.0.0 auth0:bootstrap
> node ./scripts/bootstrap.mjs
✓ Checking that the Auth0 CLI has been installed
✓ Initialize tenant settings
✓ Configuring prompt settings
✓ Creating SaaStart Management client
✓ Creating Management API Client Grant
✓ Creating SaaStart Dashboard client
✓ Creating SaaStart-Shared-Database connection
✓ Creating admin role
✓ Creating member role
✓ Creating Security Policies Action
✓ Creating Add Default Role Action
✓ Creating Add Role to Tokens Action
✓ Updating trigger bindings for Actions
✓ Saving environment variables to .env.local
✓ Creating theme for Universal Login
```

TIL you can manage and automate your auth directly from the command line with the Auth0 Deploy CLI.

Try free today → 



Secure access for AI agents.  
But not too much.

Try free today

 Remove ads

 **Watch Now** This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Lists and Tuples in Python](#)

Python lists and tuples are sequence data types that store ordered collections of items. While lists are mutable and ideal for dynamic, homogeneous data, tuples are immutable, making them suitable for fixed, heterogeneous data. Read on to compare tuples vs. lists.

**By the end of this tutorial, you'll understand that:**

- **Lists are mutable**, allowing you to modify their content, while **tuples are immutable**, meaning you can't change them after creation.

## Table of Contents

- Getting Started With Python Lists and Tuples
- Exploring Core Features of Lists and Tuples
- Using Operators and Built-in Functions With Lists and Tuples
- Packing and Unpacking Lists and Tuples
- Using Lists vs Tuples in Python
- Conclusion
- Frequently Asked Questions

Mark as Completed



- You should prefer **tuples** when you need an **immutable sequence**, such as function return values or constant data.
- You can **create a list from a tuple** using the `list()` constructor, which converts the tuple into a mutable list.
- **Tuples are immutable**, and this characteristic supports their use in scenarios where data should remain unchanged.

In this tutorial, you'll learn to define, manipulate, and choose between these two data structures. To get the most out of this tutorial, you should know the basics of Python programming, including how to define variables.

**Get Your Code:** Click here to download the free sample code that shows you how to work with lists and tuples in Python.



Share

#### ► Recommended Video Course

Lists and Tuples in Python



Join Real Python and Unlock Learning Paths, Courses, Live Q&As, and More:

[Become a Python Expert »](#)

**Take the Quiz:** Test your knowledge with our interactive “Lists vs Tuples in Python” quiz. You'll receive a score upon completion to help you track your learning progress:

#### Interactive Quiz

#### Lists vs Tuples in Python



Challenge yourself with this quiz to evaluate and deepen your understanding of Python lists and tuples. You'll explore key concepts, such as how to create, access, and manipulate these data types, while also learning best practices for using them efficiently in your code.

## Getting Started With Python Lists and Tuples

In Python, a list is a collection of arbitrary objects, somewhat akin to an [array](#) in many other programming languages but more flexible. To define a list, you typically enclose a comma-separated sequence of objects in square brackets `([])`, as shown below:

## Python

```
>>> colors = ["red", "green", "blue", "yellow"]

>>> colors
['red', 'green', 'blue', 'yellow']
```

In this code snippet, you define a list of colors using [string](#) objects separated by commas and enclose them in square brackets.

Similarly, tuples are also collections of arbitrary objects. To define a tuple, you'll enclose a comma-separated sequence of objects in parentheses (()), as shown below:

## Python

```
>>> person = ("Jane Doe", 25, "Python Developer", "Canada")

>>> person
('Jane Doe', 25, 'Python Developer', 'Canada')
```

In this example, you define a tuple with data for a given person, including their name, age, job, and base country.

Up to this point, it may seem that lists and tuples are mostly the same. However, there's an important difference:

Feature	List	Tuple
Is an ordered sequence	✓	✓
Can contain arbitrary objects	✓	✓
Can be indexed and sliced	✓	✓

Can be nested	✓	✓
Is mutable	✓	✗

Both lists and tuples are [sequence](#) data types, which means they can contain objects arranged in order. You can access those objects using an integer index that represents their position in the sequence.

Even though both data types can contain arbitrary and heterogeneous objects, you'll commonly use lists to store homogeneous objects and tuples to store heterogeneous objects.

**Note:** In this tutorial, you'll see the terms homogeneous and heterogeneous used to express the following ideas:

- **Homogeneous:** Objects of the same data type or the same semantic meaning, like a series of animals, fruits, colors, and so on.
- **Heterogeneous:** Objects of different data types or different semantic meanings, like the attributes of a car: model, color, make, year, fuel type, and so on.

You can perform indexing and slicing operations on both lists and tuples. You can also have nested lists and nested tuples or a combination of them, like a list of tuples.

The most notable difference between lists and tuples is that lists are mutable, while tuples are immutable. This feature distinguishes them and drives their specific use cases.

Essentially, a list doesn't have a fixed length since it's mutable. Therefore, it's natural to use homogeneous elements to have some structure in the list. A tuple, on the other hand, has a fixed length so the position of elements can have meaning, supporting heterogeneous data.



Secure access for developers.  
But not attackers.

Try free today

Remove ads

## Creating Lists in Python

In many situations, you'll define a `list` object using a [literal](#). A list literal is a comma-separated sequence of objects enclosed in square brackets:

Python

```
>>> countries = ["United States", "Canada", "Poland", "Germany", "Austria"]  
  
>>> countries  
['United States', 'Canada', 'Poland', 'Germany', 'Austria']
```

In this example, you create a list of countries represented by string objects. Because lists are ordered sequences, the values retain the insertion order.

**Note:** To learn more about the `list` data type, check out the [Python's list Data Type: A Deep Dive With Examples](#) tutorial.

Alternatively, you can create new lists using the `list()` [constructor](#):

Python

```
>>> digits = list(range(10))  
  
>>> digits  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In this example, you use the `list()` constructor to define a list of digits using a `range` object.

In general, `list()` can convert any iterable to a list.

That means that you can also use it to convert a tuple into a list:

Python

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

You can also create new list objects using [list comprehensions](#). For example, the following comprehension builds a list of even digits:

Python

```
>>> even_digits = [number for number in range(1, 10) if number % 2 == 0]  
  
>>> even_digits  
[2, 4, 6, 8]
```

List comprehensions are powerful tools for creating lists in Python. You'll often find them in Python code that runs transformations over sequences of data.

Finally, to create an empty list, you can use either an empty pair of square brackets or the `list()` constructor without arguments:

Python

```
>>> []  
[]  
  
>>> list()  
[]
```

The first approach is arguably the most efficient and most commonly used. However, the second approach can be more explicit and readable in some situations.

# Creating Tuples in Python

Similar to lists, you'll often create new tuples using literals. Here's a short example showing a tuple definition:

```
Python >

>>> connection = ("localhost", "8080", 3, "database.db")

>>> connection
('localhost', '8080', 3, 'database.db')
```

In this example, you create a tuple containing the parameters for a database connection. The data includes the server name, port, timeout, and database name.

**Note:** To dive deeper into the tuple data type, check out the [Python's tuple Data Type: A Deep Dive With Examples](#) tutorial.

Strictly speaking, to define a tuple, you don't need the parentheses. The comma-separated sequence will be enough:

```
Python >

>>> contact = "John Doe", "john@example.com", "55-555-5555"

>>> contact
('John Doe', 'john@example.com', '55-555-5555')
```

In practice, you can define tuples without using a pair of parentheses. However, using the parentheses is a common practice because it improves the readability of your code.

Because the parentheses are optional, to define a single-item tuple, you need to use a comma:

```
Python >
```

```
>>> t = (2, )
>>> type(t)
<class 'tuple'>

>>> t = (2)
>>> type(t)
<class 'int'>
```

In the first example, you create a tuple containing a single value by appending a comma after the value. In the second example, you use the parentheses without the comma. In this case, you create an integer value instead of a tuple.

You can also create new tuples using the `tuple()` constructor:

Python

```
>>> tuple(range(10))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

In this example, you create a list of digits using `tuple()`. This way of creating tuples can be helpful when you're working with [iterators](#) and need to convert them into tuples.

For example, you can convert a list into a tuple using the `tuple()` constructor:

Python

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

Finally, to create empty tuples, you can use a pair of parentheses or call `tuple()` without arguments:

Python

```
>>> ()
()
```

```
>>> tuple()  
( )
```

The first approach is a common way to create empty tuples. However, using `tuple()` can be more explicit and readable.



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 »](#)

[i Remove ads](#)

## Exploring Core Features of Lists and Tuples

Now that you know the basics of creating lists and tuples in Python, you're ready to explore their most relevant features and characteristics. In the following section, you'll dive into these features and learn how they can impact the use cases of lists and tuples in your Python code.

### Lists and Tuples Are Ordered Sequences

List and tuples are ordered sequences of objects. The order in which you insert the objects when you create a list or tuple is an innate characteristic. This order remains the same for that list or tuple's lifetime:

```
Python  
>>> ["mango", "orange", "apple"]  
['mango', 'orange', 'apple']  
  
>>> ("Jane", 25, "Norway")  
( 'Jane', 25, 'Norway' )
```

In these examples, you can confirm that the order of items in lists and tuples is the same order you define when creating the list or tuple.

## Lists and Tuples Can Contain Arbitrary Objects

Lists and tuples can contain any Python objects. The elements of a list or tuple can all be the same type:

```
Python
>>> [1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]

>>> (1, 2, 3, 4, 5)
(1, 2, 3, 4, 5)
```

In these examples, you create a list of integer numbers and then a tuple of similar objects. In both cases, the contained objects have the same data type. So, they're homogeneous.

The elements of a list or tuple can also be of heterogeneous data types:

```
Python
>>> ["Pythonista", 7, False, 3.14159]
['Pythonista', 7, False, 3.14159]

>>> ("Pythonista", 7, False, 3.14159)
('Pythonista', 7, False, 3.14159)
```

Here, your list and tuple contain objects of different types, including strings, integers, Booleans, and floats. So, your list and tuple are heterogeneous.

**Note:** Even though lists and tuples can contain heterogeneous or homogeneous objects, the common practice is to use lists for homogeneous objects and tuples for

heterogeneous objects.

Lists and tuples can even contain objects like functions, classes, and modules:

Python

```
>>> int
<class 'int'>

>>> len
<built-in function len>

>>> def func():
...     pass
...
>>> func
<function func at 0x1053abec0>

>>> import math
>>> math
<module 'math' from '.../math.cpython-312-darwin.so'>

>>> [int, len, func, math]
[
    <class 'int'>,
    <built-in function len>,
    <function func at 0x1053abec0>,
    <module 'math' from '.../math.cpython-312-darwin.so'>
]

>>> (int, len, func, math)
(
    <class 'int'>,
    <built-in function len>,
    <function func at 0x1053abec0>,
    <module 'math' from '.../math.cpython-312-darwin.so'>
)
```

In these examples, the list and the tuple contain a class, [built-in function](#), custom [function](#), and module objects.

Lists and tuples can contain any number of objects, from zero to as many as your computer's memory allows. In the following code, you have a list and tuple built out of a range with a million numbers:

```
Python >

>>> list(range(1_000_000))

>>> tuple(range(1_000_000))
```

These two lines of code will take some time to run and populate your screen with many, many numbers.

Finally, objects in a list or tuple don't need to be unique. A given object can appear multiple times:

```
Python >

>>> ["bark", "meow", "woof", "bark", "cheep", "bark"]
['bark', 'meow', 'woof', 'bark', 'cheep', 'bark']

>>> ("bark", "meow", "woof", "bark", "cheep", "bark")
('bark', 'meow', 'woof', 'bark', 'cheep', 'bark')
```

Lists and tuples can contain duplicated values like "bark" in the above examples.



[Remove ads](#)

# Lists and Tuples Can Be Indexed and Sliced

You can access individual elements in a list or tuple using the item's index in square brackets. This is exactly analogous to accessing individual characters in a string. List indexing is zero-based, as it is with strings.

Consider the following list:

Python

```
>>> words = ["foo", "bar", "baz", "qux", "quux", "corge"]
```

The indices for the elements in `words` are shown below:

Python



List Indices

Here's the Python code to access individual elements of `words`:

Python

```
>>> words[0]  
'foo'  
  
>>> words[2]  
'baz'  
  
>>> words[5]  
'corge'
```

The first element in the list has an index of 0. The second element has an index of 1, and so on. Virtually everything about indexing works the same for tuples.

You can also use a negative index, in which case the count starts from the end of the list:



### Negative List Indexing

Index `-1` corresponds to the last element in the list, while the first element is `-len(words)`, as shown below:

Python

```
>>> words[-1]  
'corge'
```

```
>>> words[-2]  
'quux'
```

```
>>> words[-len(words)]  
'foo'
```

Slicing also works with lists and tuples. For example, the expression `words[m:n]` returns the portion of `words` from index `m` to, but not including, index `n`:

Python

```
>>> words[2:5]  
['baz', 'qux', 'quux']
```

Other features of slicing work for lists as well. For example, you can use both positive and negative indices:

Python

```
>>> words[-5:-2]
['bar', 'baz', 'qux']

>>> words[1:4]
['bar', 'baz', 'qux']

>>> words[-5:-2] == words[1:4]
True
```

Omitting the first index starts the slice at the beginning of the list or tuple. Omitting the second index extends the slice to the end of the list or tuple:

```
Python >_>

>>> words[:4]
['foo', 'bar', 'baz', 'qux']
>>> words[0:4]
['foo', 'bar', 'baz', 'qux']

>>> words[2:]
['baz', 'qux', 'quux', 'corge']
>>> words[2:len(words)]
['baz', 'qux', 'quux', 'corge']

>>> words[:4] + words[4:]
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> words[:4] + words[4:] == words
True
```

You can specify a stride—either positive or negative:

```
Python >_>

>>> words[0:6:2]
['foo', 'baz', 'quux']
>>> words[1:6:2]
```

```
['bar', 'qux', 'corge']
```

```
>>> words[6:0:-2]  
['corge', 'qux', 'bar']
```

The slicing operator (`[ : ]`) works for both lists and tuples. You can check it out by turning words into a tuple and running the same slicing operations on it.



[i Remove ads](#)

## Lists and Tuples Can Be Nested

You've seen that an element in a list or tuple can be of any type. This means that they can contain other lists or tuples. For example, a list can contain sublists, which can contain other sublists, and so on, to arbitrary depth.

Consider the following example:

```
Python
```

```
>>> x = ["a", ["bb", ["ccc", "ddd"], "ee", "ff"], "g", ["hh", "ii"], "j"]
```

The internal structure of this list is represented in the diagram below:



### A Nested List

In this diagram,  $x[0]$ ,  $x[2]$ , and  $x[4]$  are strings, each one character long:

```
Python >
>>> x[0], x[2], x[4]
('a', 'g', 'j')
```

However,  $x[1]$  and  $x[3]$  are sublists or nested lists:

```
Python >
>>> x[1]
['bb', ['ccc', 'ddd'], 'ee', 'ff']

>>> x[3]
['hh', 'ii']
```

To access the items in a sublist, append an additional index:

Python

```
>>> x[1][0]
'bb'

>>> x[1][1]
['ccc', 'ddd']

>>> x[1][2]
'ee'

>>> x[1][3]
'ff'
```

Here, `x[1][1]` is yet another sublist, so adding one more index accesses its elements:

Python

```
>>> x[1][1][0]
'ccc'

>>> x[1][1][1]
'ddd'
```

There's no limit to the depth you can nest lists this way. However, deeply nested lists or tuples can be hard to decipher in an indexing or slicing context.

## Lists Are Mutable, Tuples Are Immutable

The built-in `list` class provides a [mutable](#) data type. Being mutable means that once you create a `list` object, you can add, delete, shift, and move elements around at will. Python provides many ways to modify lists, as you'll learn in a moment. Unlike lists, tuples are [immutable](#), meaning that you can't change a tuple once it has been created.

**Note:** To learn more about mutability and immutability in Python, check out the [Python's Mutable vs Immutable Types: What's the Difference?](#) tutorial.

You can replace or update a value in a list by indexing it on the left side of an [assignment](#) statement:

Python

```
>>> letters = ["A", "B", "c", "d"] # A list

>>> letters[2] = "C"
>>> letters
['A', 'B', 'C', 'd']

>>> letters[-1] = "D"
>>> letters
['A', 'B', 'C', 'D']
```

In this example, you create a list of letters where some letters are in uppercase while others are in lowercase. You use an assignment to turn the lowercase letters into uppercase letters.

Now, because tuples are immutable, you can't do with a tuple what you did in the above example with a list:

Python

```
>>> letters = ("A", "B", "c", "d") # A tuple

>>> letters[2] = "C"
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

If you try to update the value of a tuple element, you get a [TypeError](#) exception because tuples are immutable, and this type of operation isn't allowed for them.

You can also use the `del` statement to delete individual items from a list. However, that operation won't work on tuples:

```
Python >

>>> fruits = ["apple", "orange", "mango", "grape"]
>>> del fruits[0] # Remove apple
>>> fruits
['orange', 'mango', 'grape']

>>> person = ("John Doe", 35, "Web Dev")
>>> del person[1] # Try to remove the age value
Traceback (most recent call last):
...
TypeError: 'tuple' object doesn't support item deletion
```

You can remove individual elements from lists using the `del` statement because lists are mutable, but this won't work with tuples because they're immutable.

**Note:** To learn more about the `del` statement, check out the [Python's `del`: Remove References From Scopes and Containers](#) tutorial.

What if you want to change several elements in a list at once? Python allows this operation with a [slice assignment](#), which has the following syntax:

```
Python Syntax >

a_list[m:n] = <iterable>
```

Think of an [iterable](#) as a container of multiple values like a list or tuple. This assignment replaces the specified slice of `a_list` with the content of `<iterable>`:

```
Python >
```

```
>>> numbers = [1, 2, 3, 0, 0, 0, 7]  
  
>>> numbers[3:6] = [4, 5, 6]  
>>> numbers  
[1, 2, 3, 4, 5, 6, 7]
```

In this example, you replace the 0 values with the corresponding consecutive numbers using a slice assignment.

It's important to note that the number of elements to insert doesn't need to be equal to the number of elements in the slice. Python grows or shrinks the list as needed. For example, you can insert multiple elements in place of a single element:

```
Python  
  
>>> numbers = [1, 2, 3, 7]  
  
>>> numbers[3:4] = [4, 5, 6, 7]  
>>> numbers  
[1, 2, 3, 4, 5, 6, 7]
```

In this example, you replace the 7 with a list of values from 4 to 7. Note how Python automatically grows the list for you.

You can also insert elements into a list without removing anything. To do this, you can specify a slice of the form [n:n] at the desired index:

```
Python  
  
>>> numbers = [1, 2, 3, 7]  
  
>>> numbers[3:3] = [4, 5, 6]  
>>> numbers  
[1, 2, 3, 4, 5, 6, 7]
```

In this example, you insert the desired values at index 3. Because you're using an empty slice, Python doesn't replace any of the existing values. Instead, it makes space for the new values as needed.

You can't do slice assignment on tuple objects:

```
Python
>>> numbers[3:3] = [4, 5, 6]
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

Because tuples are immutable, they don't support slice assignment. If you try to do it, then you get a `TypeError` exception.



[i Remove ads](#)

## Lists Have Mutator Methods, Tuples Don't

Python lists have several methods that you can use to modify the underlying list. These methods aren't available for tuples because tuples are immutable, so you can't change them in place.

In this section, you'll explore the [mutator methods](#) available in Python list objects. These methods are handy in many situations, so they're great tools for you as a Python developer.

`.append(obj)`

The `.append(obj)` method appends an object to the end of a list as a single item:

Python

```
>>> a = ["a", "b"]

>>> a.append("c")
>>> a
['a', 'b', 'c']
```

In this example, you append the letter "c" at the end of a using the `.append()` method, which modifies the list in place.

**Note:** List mutator methods modify the target list [in place](#). They don't return a new list:

Python

```
>>> a = ["a", "b"]

>>> x = a.append("c")
>>> print(x)
None

>>> a
['a', 'b', 'c']
```

In this code, you grab the return value of `.append()` in `x`. Using the `print()` function, you can uncover that the value is `None` instead of a new list object. While this behavior is [deliberate](#) to make it clear that the method mutates the object in place, it can be a common source of confusion when you're starting to learn Python.

If you use an iterable as an argument to `.append()`, then that iterable is added as a single object:

Python

```
>>> a = ["a", "b"]

>>> a.append(["c", "d", "e"])
>>> a
['a', 'b', ['c', 'd', 'e']]
```

This call to `.append()` adds the input list of letters as it is instead of appending three individual letters at the end of `a`. Therefore, the final list has three elements—the two initial strings and one list object. This may not be what you intended if you wanted to grow the list with the contents of the iterable.

### `.extend(iterable)`

The `.extend()` method also adds items to the end of a list. However, the argument is expected to be an iterable like another list. The items in the input `iterable` are added as individual values:

```
Python

>>> a = ["a", "b"]

>>> a.extend(["c", "d", "e"])
>>> a
['a', 'b', 'c', 'd', 'e']
```

The `.extend()` method behaves like the concatenation operator `(+)`. More precisely, since it modifies the list in place, it behaves like the augmented concatenation operator `(+=)`. Here's an example:

```
Python

>>> a = ["a", "b"]

>>> a += ["c", "d", "e"]
```

```
>>> a
['a', 'b', 'c', 'd', 'e']
```

The augmented concatenation operator produces the same result as `.extend()`, adding individual items at the end of the target list.

### `.insert(index, obj)`

The `.insert()` method inserts the input object into the target list at the position specified by `index`. Following the method call, `a[<index>]` is `<obj>`, and the remaining list elements are pushed to the right:

```
Python >
>>> a = ["a", "c"]
>>> a.insert(1, "b")
>>> a
['a', 'b', 'c']
```

In this example, you insert the letter "b" between "a" and "c" using `.insert()`. Note that just like `.append()`, the `.insert()` method inserts the input object as a single element in the target list.

### `.remove(obj)`

The `.remove()` method removes the input object from a list. If `obj` isn't in the target list, then you get a `ValueError` exception:

```
Python >
>>> a = ["a", "b", "c", "d", "e"]
>>> a.remove("b")
>>> a
```

```
['a', 'c', 'd', 'e']
>>> a.remove("c")
>>> a
['a', 'd', 'e']
```

With `.remove()`, you can delete specific objects from a given list. Note that this method removes only one instance of the input object. If the object is duplicated, then only its first instance will be deleted.

`.pop([index=-1])`

The `.pop()` method also allows you to remove items from a list. It differs from `.remove()` in two aspects:

1. It takes the index of the object to remove rather than the object itself.
2. It returns the value of the removed object.

Calling `.pop()` without arguments removes and returns the last item in the list:

```
Python
>>> a = ["a", "b", "c", "d", "e"]

>>> a.pop()
'e'
>>> a
['a', 'b', 'c', 'd']

>>> a.pop()
'd'
>>> a
['a', 'b', 'c']
```

If you specify the optional `index` argument, then the item at that index is removed and returned. Note that `index` can be negative too:

Python

```
>>> a = ["a", "b", "c", "d", "e"]  
  
>>> a.pop(1)  
'b'  
>>> a  
['a', 'c', 'd', 'e']  
  
>>> a.pop(3)  
'e'  
>>> a  
['a', 'c', 'd']  
  
>>> a.pop(-2)  
'c'  
>>> a  
['a', 'd']  
  
>>> a.pop(-1)  
'd'  
>>> a  
['a']
```

The `index` argument defaults to `-1`, so `a.pop(-1)` is equivalent to `a.pop()`.



[i Remove ads](#)

# Using Operators and Built-in Functions With Lists and Tuples

Several Python [operators](#) and [built-in functions](#) also work with lists and tuples. For example, the `in` and `not in` operators allow you to run membership tests on lists:

Python

```
>>> words = ["foo", "bar", "baz", "qux", "quux", "corge"]

>>> "qux" in words
True

>>> "py" in words
False

>>> "thud" not in words
True
```

The `in` operator returns `True` if the target object is in the list and `False` otherwise. The `not in` operator produces the opposite result.

The [concatenation](#) (`+`) and repetition (`*`) operators also work with lists and tuples:

Python

```
>>> words + ["grault", "garply"]
['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'grault', 'garply']

>>> words * 2
['foo', 'bar', 'baz', 'qux', 'quux', 'corge',
 'foo', 'bar', 'baz', 'qux', 'quux', 'corge']
```

You can also use the built-in `len()`, `min()`, `max()`, and `sum()` functions with lists and tuples:

Python

```
>>> numbers = [2, 7, 5, 4, 8]

>>> len(numbers)
5

>>> min(numbers)
2

>>> max(numbers)
8

>>> sum(numbers)
26
```

In this example, the `len()` function returns the number of values in the list. The `min()` and `max()` functions return the minimum and maximum values in the list, respectively. The `sum()` function returns the sum of the values in the input list.

Finally, it's important to note that all these functions work the same with tuples. So, instead of using them with `list` objects, you can also use `tuple` objects.

## Packing and Unpacking Lists and Tuples

A tuple literal can contain several items that you typically assign to a single variable or name:

Python

```
>>> t = ("foo", "bar", "baz", "qux")
```

When this occurs, it's as though the items in the tuple have been *packed* into the object, as shown in the diagram below:



## Tuple Packing

If the *packed* objects are assigned to a tuple of names, then the individual objects are *unpacked* as shown in the diagram below, where you use a tuple of `s*` variables:



## Tuple Unpacking

Here's how this unpacking works in Python code:

Python



```
>>> s1, s2, s3, s4 = t
```

```
>>> s1
'foo'
>>> s2
'bar'
>>> s3
'baz'
>>> s4
'qux'
```

Note how each variable receives a single value from the unpacked tuple. When you're unpacking a tuple, the number of variables on the left must match the number of values in the tuple. Otherwise, you get a `ValueError` exception:

Python

```
>>> s1, s2, s3 = t
Traceback (most recent call last):
...
ValueError: too many values to unpack (expected 3)

>>> s1, s2, s3, s4, s5 = t
Traceback (most recent call last):
...
ValueError: not enough values to unpack (expected 5, got 4)
```

In the first example, the number of variables is less than the items in the tuple, and the error message says that there are too many values to unpack. In the second example, the number of variables exceeds the number of items in the tuple. This time, the error message says that there aren't enough values to unpack.

You can combine packing and unpacking in one statement to run a parallel assignment:

Python

```
>>> s1, s2, s3, s4 = "foo", "bar", "baz", "qux"

>>> s1
'foo'
>>> s2
'bar'
>>> s3
'baz'
>>> s4
'qux'
```

Again, the number of elements in the tuple on the left of the assignment must equal the number on the right. Otherwise, you get an error.

Tuple assignment allows for a curious bit of idiomatic Python. Sometimes, when programming, you have two variables whose values you need to swap. In most programming languages, it's necessary to store one of the values in a temporary variable while the swap occurs.

Consider the following example that compares swapping with a temporary variable and unpacking:

Python

```
>>> a = "foo"
>>> b = "bar"

>>> # Using a temporary variable
>>> temp = a
>>> a = b
>>> b = temp

>>> a, b
('bar', 'foo')

>>> a = "foo"
```

```
>>> b = "bar"

>>> # Using unpacking
>>> a, b = b, a

>>> a, b
('bar', 'foo')
```

Using a temporary variable to swap values can be annoying, so it's great that you can do it with a single unpacking operation in Python. This feature also improves your code's readability, making it more explicit.



[i Remove ads](#)

## Using Lists vs Tuples in Python

Everything you've learned so far about lists and tuples can help you decide when to use a list or a tuple in your code. Here's a summary of when it would be appropriate to use a *list* instead of a tuple:

- **Mutable collections:** When you need to add, remove, or change elements in the collection.
- **Dynamic size:** When the collection's size might change during the code's execution.
- **Homogeneous data:** When you need to store data of a homogeneous type or when the data represents a homogeneous concept.

Similarly, it's appropriate to use a *tuple* rather than a list in the following situations:

- **Immutable collections:** When you have a fixed collection of items that shouldn't change, such as coordinates (x, y, z), RGB color values, or other groupings of related

values.

- **Fixed size:** When the collection's size won't change during the code's execution.
- **Heterogeneous data:** When you need to store data of a heterogeneous type or when the data represents a heterogeneous concept.
- **Function's return values:** When a function returns multiple values, you'll typically use a tuple to pack these values together.

Finally, tuples can be more memory-efficient than lists, especially for large collections where immutability is acceptable or preferred. Similarly, if the integrity of the data is important and should be preserved throughout the program, tuples ensure and communicate that the data must remain unchanged.

## Conclusion

Now you know the basic features of Python **lists** and **tuples** and understand how to manipulate them in your code. You'll use these two data types extensively in your Python programming journey.

**In this tutorial, you've:**

- Learned about the built-in **lists** and **tuples** data types in Python
- Explored the **core features** of lists and tuples
- Discovered how to **define** and **manipulate** lists and tuples
- Learned **when to use** lists or tuples in your code

With this knowledge, you can now decide when it's appropriate to use a list or tuple in your Python code. You also have the essential skills to create and manipulate lists and tuples in Python.

**Get Your Code:** [Click here to download the free sample code](#) that shows you how to

work with lists and tuples in Python.

 **Take the Quiz:** Test your knowledge with our interactive “Lists vs Tuples in Python” quiz. You’ll receive a score upon completion to help you track your learning progress:

#### Interactive Quiz



#### Lists vs Tuples in Python

Challenge yourself with this quiz to evaluate and deepen your understanding of Python lists and tuples. You'll explore key concepts, such as how to create, access, and manipulate these data types, while also learning best practices for using them efficiently in your code.

## Frequently Asked Questions

Now that you have some experience with lists and tuples in Python, you can use the questions and answers below to check your understanding and recap what you've learned.

These FAQs are related to the most important concepts you've covered in this tutorial. Click the *Show/Hide* toggle beside each question to reveal the answer.

### What is the difference between lists and tuples in Python?

Show/Hide

### When would you prefer tuples over lists?

Show/Hide

### How do you create a list from a tuple in Python?

Show/Hide

## What's the point of a tuple?

Show/Hide

## Are tuples immutable?

Show/Hide

Mark as Completed



Share



This tutorial has a related video course created by the Real Python team.

Watch it together with the written tutorial to deepen your understanding: [Lists and Tuples in Python](#)



## Python Tricks



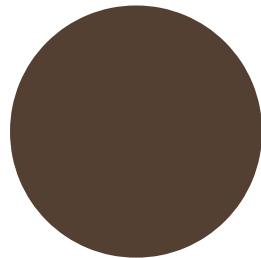
Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
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 a self-taught Python developer, educator, and technical writer with over 10 years of experience.

[» 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



Brenda



Dan



Geir Arne



Joanna



John



Martin

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:**



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 Topics: [basics](#) [python](#)

Recommended Video Course: [Lists and Tuples in Python](#)

Related Tutorials:

- [Dictionaries in Python](#)
- [Sets in Python](#)
- [Python's list Data Type: A Deep Dive With Examples](#)
- [Defining Your Own Python Function](#)
- [Python Virtual Environments: A Primer](#)

Learn Python	Courses & Paths	Community	Membership	Company
<a href="#">Start Here</a>	<a href="#">Learning Paths</a>	<a href="#">Podcast</a>	<a href="#">Plans &amp; Pricing</a>	<a href="#">About Us</a>
<a href="#">Learning Resources</a>	<a href="#">Quizzes &amp; Exercises</a>	<a href="#">Newsletter</a>	<a href="#">Team Plans</a>	<a href="#">Team</a>
<a href="#">Code Mentor</a>	<a href="#">Browse Topics</a>	<a href="#">Community Chat</a>	<a href="#">For Business</a>	<a href="#">Mission &amp; Values</a>
<a href="#">Python Reference</a>	<a href="#">Live Courses</a>	<a href="#">Office Hours</a>	<a href="#">For Schools</a>	<a href="#">Editorial Guidelines</a>
<a href="#">Python Cheat Sheet</a>	<a href="#">Books</a>	<a href="#">Learner Stories</a>	<a href="#">Reviews</a>	<a href="#">Sponsorships</a>
<a href="#">Support Center</a>				<a href="#">Careers</a>
				<a href="#">Press Kit</a>

[Privacy Policy](#) [Terms of Use](#) [Security](#) [Contact](#)

Happy Pythoning!

© 2012–2026 DevCademy Media Inc. DBA Real Python. All rights reserved.

REALPYTHON™ is a trademark of DevCademy Media Inc.