Functional programming typically uses [lists](https://realpython.com/python-lists-tuples/), arrays, and other iterables to represent the data along with a set of functions that operate on that data and transform it. When it comes to processing data with a functional style, there are at least three commonly used techniques:

1. [**Mapping**](https://en.wikipedia.org/wiki/Map_(higher-order_function)) consists of applying a transformation function to an iterable to produce a new iterable. Items in the new iterable are produced by calling the transformation function on each item in the original iterable.
2. [**Filtering**](https://en.wikipedia.org/wiki/Filter_(higher-order_function)) consists of applying a [predicate or Boolean-valued function](https://en.wikipedia.org/wiki/Boolean-valued_function) to an iterable to generate a new iterable. Items in the new iterable are produced by filtering out any items in the original iterable that make the predicate function return false.
3. [**Reducing**](https://en.wikipedia.org/wiki/Fold_(higher-order_function)) consists of applying a reduction function to an iterable to produce a single cumulative value.

* [Anonymous functions](https://en.wikipedia.org/wiki/Anonymous_function)
* A map() function
* A filter() function
* A reduce() function

In this tutorial, you’ll cover one of these functional features, the built-in function map(). You’ll also learn how to use [list comprehensions](https://realpython.com/list-comprehension-python/) and [generator expressions](https://realpython.com/introduction-to-python-generators/#building-generators-with-generator-expressions) to get the same functionality of map() in a Pythonic and readable way.

### **Understanding map()**

### map() loops over the items of an input iterable (or iterables) and returns an iterator that results from applying a transformation function to every item in the original input iterable.

According to the [documentation](https://docs.python.org/3/library/functions.html#map), map() takes a function object and an iterable (or multiple iterables) as arguments and returns an iterator that yields transformed items on demand. The function’s signature is defined as follows:

map(function, iterable[, iterable1, iterable2,..., iterableN])

map() applies function to each item in iterable in a loop and returns a new iterator that yields transformed items on demand. function can be any Python function that takes a number of arguments equal to the number of iterables you pass to map()

Even though the Python documentation calls this argument function, it can be any Python callable. This includes [built-in functions](https://docs.python.org/3/library/functions.html#built-in-functions), [classes](https://realpython.com/lessons/classes-python/), [methods](https://realpython.com/lessons/mastering-method-types-oop-pizza-example/), [lambda functions](https://realpython.com/courses/python-lambda-functions/), and [user-defined functions](https://realpython.com/defining-your-own-python-function/).

Since map() is written in [C](https://realpython.com/build-python-c-extension-module/) and is highly optimized, its internal implied loop can be more efficient than a regular Python for loop. This is one advantage of using map().

A second advantage of using map() is related to memory consumption. With a for loop, you need to store the whole list in your system’s memory. With map(), you get items on demand, and only one item is in your system’s memory at a given time.

For another example, say you need to convert all the items in a list from a [string to an integer number](https://realpython.com/courses/convert-python-string-int/). To do that, you can use map() along with int() as follows:

>>> str\_nums = ["4", "8", "6", "5", "3", "2", "8", "9", "2", "5"]

>>> int\_nums = map(int, str\_nums)

>>> int\_nums

<map object at 0x7fb2c7e34c70>

>>> list(int\_nums)

[4, 8, 6, 5, 3, 2, 8, 9, 2, 5]

>>> str\_nums

["4", "8", "6", "5", "3", "2", "8", "9", "2", "5"]

map() applies int() to every value in str\_nums. Since map() returns an iterator (a map object), you’ll need call list() so that you can exhaust the iterator and turn it into a list object. Note that the original sequence doesn’t get modified in the process.

### **Using map() With Different Kinds of Functions**

You can use any kind of Python callable with map(). The only condition would be that the callable takes an argument and returns a concrete and useful value. For example, you can use classes, instances that implement a special method called \_\_call\_\_(), instance methods, class methods, static methods, and functions.

>>> numbers = [-2, -1, 0, 1, 2]

>>> abs\_values = list(map(abs, numbers))

>>> abs\_values

[2, 1, 0, 1, 2]

>>> list(map(float, numbers))

[-2.0, -1.0, 0.0, 1.0, 2.0]

>>> words = ["Welcome", "to", "Real", "Python"]

>>> list(map(len, words))

[7, 2, 4, 6]

You can use any built-in function with map(), provided that the function takes an argument and returns a value.

### **Processing Multiple Input Iterables With map()**

If you supply multiple iterables to map(), then the transformation function must take as many arguments as iterables you pass in. Each iteration of map() will pass one value from each iterable as an argument to function. The iteration stops at the end of the shortest iterable.

Consider the following example that uses [pow()](https://docs.python.org/3/library/functions.html#pow):

>>> first\_it = [1, 2, 3]

>>> second\_it = [4, 5, 6, 7]

>>> list(map(pow, first\_it, second\_it))

[1, 32, 729]

pow() takes two arguments, x and y, and returns x to the power of y.

## Transforming Iterables of Strings With Python’s map()

Using the Methods of str

>>> string\_it = ["processing", "strings", "with", "map"]

>>> list(map(str.capitalize, string\_it))

['Processing', 'Strings', 'With', 'Map']

>>> list(map(str.upper, string\_it))

['PROCESSING', 'STRINGS', 'WITH', 'MAP']

>>> list(map(str.lower, string\_it))

['processing', 'strings', 'with', 'map']

There are a few transformations that you can perform on every item in string\_it using map() and string methods. Most of the time, you’d use methods that don’t take additional arguments, like str.capitalize(), str.lower(), str.swapcase(), str.title(), and str.upper().

When you use str.strip() like this, you rely on the default value of char. In this case, you use map() to remove all the whitespace in the items of with\_spaces.

chr() is the inverse of ord()

It takes an integer representing the Unicode code point of a Unicode character and returns the character at that position. For example, chr(97) will return 'a', and chr(98) will return 'b':

>>> chr(97)

'a'

>>> chr(98)

'b'

def rotate\_chr(c):

rot\_by = 3

c = c.lower()

alphabet = "abcdefghijklmnopqrstuvwxyz"

# Keep punctuation and whitespace

if c not in alphabet:

return c

rotated\_pos = ord(c) + rot\_by

# If the rotation is inside the alphabet

if rotated\_pos <= ord(alphabet[-1]):

return chr(rotated\_pos)

# If the rotation goes beyond the alphabet

return chr(rotated\_pos - len(alphabet))

Here’s an example that uses [str.join()](https://realpython.com/lessons/concatenating-joining-strings-python/) to concatenate the string:

>>> "".join(map(rotate\_chr, "My secret message goes here."))

'pb vhfuhw phvvdjh jrhv khuh.'

Strings are also iterables in Python. So, the call to map() applies rotate\_chr() to every character in the original input string. In this case, "M" becomes "p", "y" becomes "b", and so on. Finally, the call to str.join() concatenates every rotated character in a final encrypted message.

### Converting Strings to Numbers

If you’re sure that your data is clean and doesn’t contain wrong values, then you can use [float()](https://docs.python.org/3/library/functions.html#float) or int() directly according to your needs. Here are some examples:

>>> # Convert to floating-point

>>> list(map(float, ["12.3", "3.3", "-15.2"]))

[12.3, 3.3, -15.2]

>>> # Convert to integer

>>> list(map(int, ["12", "3", "-15"]))

[12, 3, -15]

Note that if one of the values is not a valid number, then you’ll get a [ValueError](https://docs.python.org/3/library/exceptions.html#ValueError).

>>> def to\_float(number):

... try:

... return float(number.replace(",", "."))

... except ValueError:

... return float("nan")

>>> list(map(to\_float, ["12.3", "3,3", "-15.2", "One"]))

[12.3, 3.3, -15.2, nan]

Inside to\_float(), you use a [try statement](https://realpython.com/python-exceptions/#the-try-and-except-block-handling-exceptions) that catches a ValueError if float() fails when converting number. If no error occurs, then your function returns number converted to a valid floating-point number. Otherwise, you get a [nan (Not a Number) value](https://docs.python.org/3/library/functions.html#float), which is a special float value that you can use to represent values that aren’t valid numbers, just like "One" in the above example.

## Combining map() With Other Functional Tools

if you use map() along with other functional tools like [filter()](https://docs.python.org/3/library/functions.html#filter) and [reduce()](https://realpython.com/python-reduce-function/), then you can perform more complex transformations on your iterables.

### map() and filter()

Sometimes you need to process an input iterable and return another iterable that results from filtering out unwanted values in the input iterable. In that case, Python’s [filter()](https://realpython.com/python-filter-function/) can be a good option for you. filter() is a built-in function that takes two positional arguments:

1. **function** will be a [predicate or Boolean-valued function](https://en.wikipedia.org/wiki/Boolean-valued_function), a function that returns True or False according to the input data.
2. **iterable** will be any Python iterable.

If you pass None to function, then filter() uses the identity function. This means that filter() will check the truth value of each item in iterable and filter out all of the items that are [falsy](https://realpython.com/python-operators-expressions/#evaluation-of-non-boolean-values-in-boolean-context).

To illustrate how you can use map() along with filter(), say you need to calculate the [square root](https://realpython.com/python-square-root-function/) of all the values in a list. Since your list can contain negative values, you’ll get an error because the square root isn’t defined for negative numbers:

import math

>>> def is\_positive(num):

... return num >= 0

>>> def sanitized\_sqrt(numbers):

... cleaned\_iter = map(math.sqrt, filter(is\_positive, numbers))

... return list(cleaned\_iter)

>>> sanitized\_sqrt([25, 9, 81, -16, 0])

[5.0, 3.0, 9.0, 0.0]

### map() and reduce()

Python’s [reduce()](https://realpython.com/python-reduce-function/) is a function that lives in a module called [functools](https://realpython.com/lessons/functools-module/) in the Python standard library. reduce() is another core functional tool in Python that is useful when you need to apply a function to an iterable and reduce it to a single cumulative value. This kind of operation is commonly known as [**reduction or folding**](https://en.wikipedia.org/wiki/Fold_(higher-order_function)). reduce() takes two required arguments:

1. **function** can be any Python callable that accepts two arguments and returns a value.
2. **iterable** can be any Python iterable.

reduce() will apply function to all the items in iterable and cumulatively compute a final value.

Here’s an example that combines map() and reduce() to calculate the total size of all the files that live in your home directory cumulatively:

>>> import functools

>>> import operator

>>> import os

>>> import os.path

>>> files = os.listdir(os.path.expanduser("~"))

>>> functools.reduce(operator.add, map(os.path.getsize, files))

4377381

Even though you can use reduce() to solve the problem covered in this section, Python offers other tools that can lead to a more Pythonic and efficient solution. For example, you can use the built-in function sum() to compute the total size of the files in your home directory:

>>> import os

>>> import os.path

>>> files = os.listdir(os.path.expanduser("~"))

>>> sum(map(os.path.getsize, files))

4377381

## Processing Tuple-Based Iterables With starmap()

Python’s **itertools.starmap()** makes an iterator that applies a function to the arguments obtained from an iterable of tuples and yields the results. It’s useful when you’re processing iterables that are already grouped in tuples.

The main difference between map() and starmap() is that the latter calls its transformation function using the [unpacking operator (\*)](https://www.python.org/dev/peps/pep-0448) to unpack each tuple of arguments into several positional arguments. So, the transformation function is called as function(\*args) instead of function(arg1, arg2,... argN).

The [official documentation for starmap()](https://docs.python.org/3/library/itertools.html#itertools.starmap) says that the function is roughly equivalent to the following Python function:

def starmap(function, iterable):

for args in iterable:

yield function(\*args)

>>> from itertools import starmap

>>> list(starmap(pow, [(2, 7), (4, 3)]))

[128, 64]

If you use map() instead of starmap(), then you’ll get a different result because map() takes one item from each tuple:

>>>

>>> list(map(pow, (2, 7), (4, 3)))

[16, 343]

## Coding With Pythonic Style: Replacing map()

Functional programming tools like map(), filter(), and reduce() have been around for a long time. However, [list comprehensions](https://realpython.com/courses/using-list-comprehensions-effectively/) and [generator expressions](https://realpython.com/introduction-to-python-generators/#building-generators-with-generator-expressions) have become a natural replacement for them almost in every use case.

### Using List Comprehensions

There’s a general pattern that you can use to replace a call to map() with a list comprehension. Here’s how:

# Generating a list with map

list(map(function, iterable))

# Generating a list with a list comprehension

[function(x) for x in iterable]

>>> numbers = [1, 2, 3, 4, 5, 6]

>>> # Using map()

>>> list(map(square, numbers))

[1, 4, 9, 16, 25, 36]

>>> # Using a list comprehension

>>> [square(x) for x in numbers]

[1, 4, 9, 16, 25, 36]

### Using Generator Expressions

map() returns a **map object**, which is an iterator that yields items on demand. So, the natural replacement for map() is a [generator expression](https://realpython.com/introduction-to-python-generators/#building-generators-with-generator-expressions) because generator expressions return generator objects, which are also iterators that yield items on demand.

There’s a tiny syntactical difference between a list comprehension and a generator expression. The first uses a pair of square brackets ([]) to delimit the expression. The second uses a pair of parentheses (()).

>>> # Transformation function

>>> def square(number):

... return number \*\* 2

>>> numbers = [1, 2, 3, 4, 5, 6]

>>> # Using map()

>>> map\_obj = map(square, numbers)

>>> map\_obj

<map object at 0x7f254d180a60>

>>> list(map\_obj)

[1, 4, 9, 16, 25, 36]

>>> # Using a generator expression

>>> gen\_exp = (square(x) for x in numbers)

>>> gen\_exp

<generator object <genexpr> at 0x7f254e056890>

>>> list(gen\_exp)

[1, 4, 9, 16, 25, 36]

## Conclusion

Python’s [**map()**](https://docs.python.org/3/library/functions.html#map) allows you to perform [**mapping**](https://en.wikipedia.org/wiki/Map_(higher-order_function)) operations on iterables. A mapping operation consists of applying a **transformation function** to the items in an iterable to generate a transformed iterable. In general, map() will allow you to process and transform iterables without using an explicit loop. In this tutorial, you’ve learned how map() works and how to use it to process iterables. You also learned about some [Pythonic](https://realpython.com/learning-paths/writing-pythonic-code/) tools that you can use to replace map() in your code.

**You now know how to:**

* Work with Python’s **map()**
* Use map() to **process** and **transform** iterables without using an explicit loop
* Combine map() with functions like **filter()** and **reduce()** to perform complex transformations
* Replace map() with tools like **list comprehensions** and **generator expressions**

With this new knowledge, you’ll be able to use map() in your code and approach your code with a [functional programming style](https://realpython.com/courses/functional-programming-python/). You can also switch to a more Pythonic and modern style by replacing map() with a [list comprehension](https://realpython.com/list-comprehension-python/) or a [generator expression](https://realpython.com/introduction-to-python-generators/#building-generators-with-generator-expressions).