Python's map(): Processing Iterables Without a Loop

by Leodanis Pozo Ramos Sep 30, 2020 19 Comments basics best-practices python

Mark as Completed

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Python's map() Function: Transforming Iterables**

Python's `map()` is a built-in function that allows you to process and transform all the items in an iterable without using an explicit `for` loop, a technique commonly known as mapping. `map()` is useful when you need to apply a **transformation function** to each item in an

Table of Contents

- Coding With Functional Style in Python
- Getting Started With Python's map()
- Transforming Iterables of Strings With Python's map()
- Transforming Iterables of Numbers With Python's map()
- Combining map() With Other Functional Tools
- Processing Tuple-Based Iterables With starmap()
- Coding With Pythonic Style: Replacing map()
- Conclusion

Mark as Completed

Tweet Share Email

`map()` is useful when you need to apply a **transformation function** to each item in an iterable and transform them into a new iterable. `map()` is one of the tools that support a functional programming style in Python.

In this tutorial, you'll learn:

- How Python's `map()` works
- How to **transform** different types of Python iterables using `map()`
- How to **combine** `map()` with other functional tools to perform more complex transformations
- What tools you can use to **replace** `map()` and make your code more **Pythonic**

With this knowledge, you'll be able to use `map()` effectively in your programs or, alternatively, to use list comprehensions or generator expressions to make your code more Pythonic and readable.

For a better understanding of `map()`, some previous knowledge of how to work with iterables, `for` loops, functions, and `lambda` functions would be helpful.

Coding With Functional Style in Python

In **functional programming**, computations are done by combining functions that take arguments and return a concrete value (or values) as a result. These functions don't modify their input arguments and don't change the program's state. They just provide the result of a given computation. These kinds of functions are commonly known as pure functions.

In theory, programs that are built using a functional style will be easier to:

- **Develop** because you can code and use every function in isolation
- **Debug and test** because you can test and debug individual functions without looking at the rest of the program
- **Understand** because you don't need to deal with state changes throughout the program

Functional programming typically uses lists, 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** 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** consists of applying a predicate or 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** consists of applying a reduction function to an iterable to produce a single cumulative value.

According to Guido van Rossum, Python is more strongly influenced by imperative programming languages than functional languages:

> I have never considered Python to be heavily influenced by functional languages, no matter what people say or think. I was much more familiar with imperative languages such as C and Algol 68 and although I had made functions first-class objects, I didn't view Python as a functional programming language. (Source)

However, back in 1993, the Python community was demanding some functional programming features. They were asking for:

- Anonymous functions

- A `map()` function
- A `filter()` function
- A `reduce()` function

These functional features were added to the language thanks to the contribution of a community member. Nowadays, `map()`, `filter()`, and `reduce()` are fundamental components of the functional programming style in Python.

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 and generator expressions to get the same functionality of `map()` in a Pythonic and readable way.

Getting Started With Python's `map()`

Sometimes you might face situations in which you need to perform the same operation on all the items of an input iterable to build a new iterable. The quickest and most common approach to this problem is to use a Python `for` loop. However, you can also tackle this problem without an explicit loop by using `map()`.

In the following three sections, you'll learn how `map()` works and how you can use it to process and transform iterables without a loop.

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, `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:

Python

```
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()`.

> **Note:** The first argument to `map()` is a **function object**, which means that you need to pass a function without calling it. That is, without using a pair of parentheses.

This first argument to `map()` is a **transformation function**. In other words, it's the function that transforms each original item into a new (transformed) item. Even though the Python documentation calls this argument `function`, it can be any Python callable. This includes built-in functions, classes, methods, `lambda` functions, and user-defined functions.

The operation that `map()` performs is commonly known as a **mapping** because it maps every item in an input iterable to a new item in a resulting iterable. To do that, `map()` applies a transformation function to all the items in the input iterable.

To better understand `map()`, suppose you need to take a list of numeric values and transform it into a list containing the square value of every number in the original list. In this case, you can use a `for` loop and code something like this:

Python >>>

```
>>> numbers = [1, 2, 3, 4, 5]
>>> squared = []

>>> for num in numbers:
...     squared.append(num ** 2)
```

```
...     squared.append(num ** 2)
...

>>> squared
[1, 4, 9, 16, 25]
```

When you run this loop on `numbers`, you get a list of square values. The `for` loop iterates over `numbers` and applies a power operation on each value. Finally, it stores the resulting values in `squared`.

You can achieve the same result without using an explicit loop by using `map()`. Take a look at the following reimplementation of the above example:

Python >>>

```
>>> def square(number):
...     return number ** 2
...

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

>>> squared = map(square, numbers)

>>> list(squared)
[1, 4, 9, 16, 25]
```

`square()` is a transformation function that maps a number to its square value. The call to `map()` applies `square()` to all of the values in `numbers` and returns an iterator that yields square values. Then you call `list()` on `map()` to create a list object containing the square values.

Since `map()` is written in C 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.

> **Note:** In Python 2.x, `map()` returns a list. This behavior changed in Python 3.x. Now, `map()` returns a map object, which is an iterator that yields items on demand. That's why you need to call `list()` to create the desired list object.

For another example, say you need to convert all the items in a list from a string to an integer number. To do that, you can use `map()` along with `int()` as follows:

Python >>>

```
>>> 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.

There are some built-in functions that you can use with `map()`. Consider the following examples:

```
Python >>>
>>> 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.

A common pattern that you'll see when it comes to using `map()` is to use a `lambda` function as the first argument. `lambda` functions are handy when you need to pass an expression-based function to `map()`. For example, you can reimplement the example of square values using a `lambda` function as follows:

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

>>> squared = map(lambda num: num ** 2, numbers)

>>> list(squared)
[1, 4, 9, 16, 25]
```

`lambda` functions are quite useful when it comes to using `map()`. They can play the role of the first argument to `map()`. You can use `lambda` functions along with `map()` to quickly process and transform your iterables.

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()`:

```
Python >>>
>>> 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`. In the first iteration, `x` will be `1`, `y` will be `4`, and the result will be `1`. In the second iteration, `x` will be `2`, `y` will be `5`, and the result will be `32`, and so on. The final iterable is only as long as the shortest iterable, which is `first_it` in this case.

This technique allows you to merge two or more iterables of numeric values using different kinds of math operations. Here are some examples that use `lambda` functions to perform different math operations on several input iterables:

```
Python >>>
>>> list(map(lambda x, y: x - y, [2, 4, 6], [1, 3, 5]))
[1, 1, 1]

>>> list(map(lambda x, y, z: x + y + z, [2, 4], [1, 3], [7, 8]))
[10, 15]
```

In the first example, you use a subtraction operation to merge two iterables of three items each. In the second example, you add together the values of three iterables.

Transforming Iterables of Strings With Python's `map()`

When you're working with iterables of string objects, you might be interested in transforming all the objects using some kind of transformation function. Python's `map()` can be your ally in these situations. The following sections will walk you through some examples of how to use `map()` to transform iterables of string objects.

Using the Methods of `str`

A quite common approach to string manipulation is to use some of the methods of the class `str` to transform a given string into a new string. If you're dealing with iterables of strings and need to apply the same transformation to each string, then you can use `map()` along with various string methods:

```
Python >>>
>>> 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()`.

You can also use some methods that take additional arguments with default values, such as `str.strip()`, which takes an optional argument called `char` that defaults to removing whitespace:

```
Python >>>
>>> with_spaces = ["processing ", "  strings", "with   ", " map   "]

>>> list(map(str.strip, with_spaces))
['processing', 'strings', 'with', 'map']
```

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`.

> **Note:** If you need to supply arguments rather than rely on the default value, then you can use a `lambda` function.
>
> Here's an example that uses `str.strip()` to remove dots rather than the default whitespace:
>
> ```
> Python >>>
> >>> with_dots = ["processing..", "...strings", "with....", "..map.."]
>
> >>> list(map(lambda s: s.strip("."), with_dots))
> ['processing', 'strings', 'with', 'map']
> ```

The `lambda` function calls `.strip()` on the string object `s` and removes all the leading and trailing dots.

This technique can be handy when, for example, you're processing text files in which lines can have trailing spaces (or other characters) and you need to remove them. If this is the case, then you need to consider that using `str.strip()` without a custom `char` will remove the newline character as well.

Removing Punctuation

When it comes to processing text, you sometimes need to remove the punctuation marks that remain after you split the text into words. To deal with this problem, you can create a custom function that removes the punctuation marks from a single word using a regular expression that matches the most common punctuation marks.

Here's a possible implementation of this function using `sub()`, which is a regular expression function that lives in the `re` module in Python's standard library:

```
Python                                                    >>>
>>> import re

>>> def remove_punctuation(word):
...     return re.sub(r'[!?.:;,"()-]', "", word)

>>> remove_punctuation("...Python!")
'Python'
```

Inside `remove_punctuation()`, you use a regular expression pattern that matches the most common punctuation marks that you'll find in any text written in English. The call to `re.sub()` replaces the matched punctuation marks using an empty string (`""`) and returns a cleaned `word`.

With your transformation function in place, you can use `map()` to run the transformation on every word in your text. Here's how it works:

```
Python                                                    >>>
>>> text = """Some people, when confronted with a problem, think
... "I know, I'll use regular expressions."
... Now they have two problems. Jamie Zawinski"""

>>> words = text.split()
>>> words
['Some', 'people,', 'when', 'confronted', 'with', 'a', 'problem,', 'think'
, '"I', 'know,', "I'll", 'use', 'regular', 'expressions."', 'Now', 'they',
 'have', 'two', 'problems.', 'Jamie', 'Zawinski']

>>> list(map(remove_punctuation, words))
['Some', 'people', 'when', 'confronted', 'with', 'a', 'problem', 'think',
'I', 'know', "I'll", 'use', 'regular', 'expressions', 'Now', 'they', 'have
', 'two', 'problems', 'Jamie', 'Zawinski']
```

In this piece of text, some words include punctuation marks. For example, you have `'people,'` instead of `'people'`, `'problem,'` instead of `'problem'`, and so on. The call to `map()` applies `remove_punctuation()` to every word and removes any punctuation mark. So, in the second `list`, you have cleaned words.

Note that the apostrophe (`'`) isn't in your regular expression because you want to keep contractions like `I'll` as they are.

Implementing a Caesar Cipher Algorithm

Julius Caesar, the Roman statesman, used to protect the messages he sent to his generals by encrypting them using a cipher. A Caesar cipher shifts each letter by a number of letters. For example, if you shift the letter `a` by three, then you get the letter `d`, and so on.

If the shift goes beyond the end of the alphabet, then you just need to rotate back to the beginning of the alphabet. In the case of a rotation by three, `x` would become `a`. Here's how the alphabet would look after the rotation:

- **Original alphabet:** `abcdefghijklmnopqrstuvwxyz`
- **Alphabet rotated by three:** `defghijklmnopqrstuvwxyzabc`

The following code implements `rotate_chr()`, a function that takes a character and rotates it by three. `rotate_chr()` will return the rotated character. Here's the code:

Python

```
1  def rotate_chr(c):
2      rot_by = 3
3      c = c.lower()
4      alphabet = "abcdefghijklmnopqrstuvwxyz"
5      # Keep punctuation and whitespace
6      if c not in alphabet:
7          return c
8      rotated_pos = ord(c) + rot_by
9      # If the rotation is inside the alphabet
10     if rotated_pos <= ord(alphabet[-1]):
11         return chr(rotated_pos)
12     # If the rotation goes beyond the alphabet
13     return chr(rotated_pos - len(alphabet))
```

Inside `rotate_chr()`, you first check if the character is in the alphabet. If not, then you return the same character. This has the purpose of keeping punctuation marks and other unusual characters. In line 8, you calculate the new rotated position of the character in the alphabet. To do this, you use the built-in function `ord()`.

`ord()` takes a Unicode character and returns an integer that represents the **Unicode code point** of the input character. For example, `ord("a")` returns `97`, and `ord("b")` returns `98`:

Python >>>

```
>>> ord("a")
97
>>> ord("b")
98
```

`ord()` takes a character as an argument and returns the Unicode code point of the input character.

If you add this integer to the target number of `rot_by`, then you'll get the rotated position of the new letter in the alphabet. In this example, `rot_by` is `3`. So, the letter `"a"` rotated by three will become the letter at position `100`, which is the letter `"d"`. The letter `"b"` rotated by three will become the letter at position `101`, which is the letter `"e"`, and so on.

If the new position of the letter doesn't go beyond the position of the last letter (`alphabet[-1]`), then you return the letter at this new position. To do that, you use the built-in function `chr()`.

`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'`:

Python >>>

```
>>> chr(97)
'a'
>>> chr(98)
'b'
```

`chr()` takes an integer that represents the Unicode code point of a character and returns corresponding character.

Finally, if the new rotated position is beyond the position of the last letter (`alphabet[-1]`), then you need to rotate back to the beginning of the alphabet. To do that, you need to subtract the length of the alphabet from the rotated position (`rotated_pos - len(alphabet)`) and then return the letter at that new position using `chr()`.

With `rotate_chr()` as your transformation function, you can use `map()` to encrypt any text using the Caesar cipher algorithm. Here's an example that uses `str.join()` to concatenate the string:

```
Python >>>
>>> "".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.

Transforming Iterables of Numbers With Python's `map()`

`map()` also has great potential when it comes to processing and transforming iterables of **numeric values**. You can perform a wide variety of math and arithmetic operations, convert string values to floating-point numbers or integer numbers, and so on.

In the following sections, you'll cover some examples of how to use `map()` to process and transform iterables of numbers.

Using Math Operations

A common example of using math operations to transform an iterable of numeric values is to use the power operator (`**`). In the following example, you code a transformation function that takes a number and returns the number squared and cubed:

```
Python >>>
>>> def powers(x):
...     return x ** 2, x ** 3
...

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

>>> list(map(powers, numbers))
[(1, 1), (4, 8), (9, 27), (16, 64)]
```

`powers()` takes a number `x` and returns its square and cube. Since Python handles multiple return values as tuples, each call to `powers()` returns a tuple with two values. When you call `map()` with `powers()` as an argument, you get a list of tuples containing the square and the cube of every number in the input iterable.

There are a lot of math-related transformations that you can perform with `map()`. You can add constants to and subtract them from each value. You can also use some functions from the `math` module like `sqrt()`, `factorial()`, `sin()`, `cos()`, and so on. Here's an example using `factorial()`:

```
Python >>>
>>> import math

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

>>> list(map(math.factorial, numbers))
[1, 2, 6, 24, 120, 720, 5040]
```

In this case, you transform `numbers` into a new list containing the factorial of each number in the original list.

You can perform a wide spectrum of math transformations on an iterable of numbers using `map()`. How far you get into this topic will depend on your needs and your imagination. Give it some thought and code your own examples!

Converting Temperatures

Another use case for `map()` is to convert between **units of measure**. Suppose you have a list of temperatures measured in degrees Celsius or Fahrenheit and you need to convert them into the corresponding temperatures in degrees Fahrenheit or Celsius.

You can code two transformation functions to accomplish this task:

Python

```
def to_fahrenheit(c):
    return 9 / 5 * c + 32

def to_celsius(f):
    return (f - 32) * 5 / 9
```

`to_fahrenheit()` takes a temperature measurement in Celsius and makes the conversion to Fahrenheit. Similarly, `to_celsius()` takes a temperature in Fahrenheit and converts it to Celsius.

These functions will be your transformation functions. You can use them with `map()` to convert an iterable of temperature measurements to Fahrenheit and to Celsius respectively:

Python >>>

```
>>> celsius_temps = [100, 40, 80]
>>> # Convert to Fahrenheit
>>> list(map(to_fahrenheit, celsius_temps))
[212.0, 104.0, 176.0]

>>> fahr_temps = [212, 104, 176]
>>> # Convert to Celsius
>>> list(map(to_celsius, fahr_temps))
[100.0, 40.0, 80.0]
```

If you call `map()` with `to_fahrenheit()` and `celsius_temps`, then you get a list of temperature measures in Fahrenheit. If you call `map()` with `to_celsius()` and `fahr_temps`, then you get a list of temperature measures in Celsius.

To extend this example and cover any other kind of unit conversion, you just need to code an appropriate transformation function.

Converting Strings to Numbers

When working with numeric data, you'll likely deal with situations in which all your data are string values. To do any further calculation, you'll need to convert the string values into numeric values. `map()` can help with these situations, too.

If you're sure that your data is clean and doesn't contain wrong values, then you can use `float()` or `int()` directly according to your needs. Here are some examples:

Python >>>

```
>>> # 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]
```

In the first example, you use `float()` with `map()` to convert all the values from string values to floating-point values. In the second case, you use `int()` to convert from a string to an integer. Note that if one of the values is not a valid number, then you'll get a `ValueError`.

If you're not sure that your data is clean, then you can use a more elaborate conversion function like the following:

Python >>>

```
>>> 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 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, 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.

You can customize `to_float()` according to your needs. For example, you can replace the statement `return float("nan")` with the statement `return 0.0`, and so on.

Combining `map()` With Other Functional Tools

So far, you've covered how to use `map()` to accomplish different tasks involving iterables. However, if you use `map()` along with other functional tools like `filter()` and `reduce()`, then you can perform more complex transformations on your iterables. That's what you're going to cover in the following two sections.

`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()` 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, a function that returns `True` or `False` according to the input data.
2. **`iterable`** will be any Python iterable.

`filter()` yields the items of the input `iterable` for which `function` returns `True`. 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.

To illustrate how you can use `map()` along with `filter()`, say you need to calculate the square root 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:

Python >>>

```
>>> import math

>>> math.sqrt(-16)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    math.sqrt(-16)
ValueError: math domain error
```

With a negative number as an argument, `math.sqrt()` raises a `ValueError`. To avoid this issue, you can use `filter()` to filter out all the negative values and then find the square root

of the remaining positive values. Check out the following example:

Python >>>

```
>>> 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]
```

`is_positive()` is a predicate function that takes a number as an argument and returns `True` if the number is greater than or equal to zero. You can pass `is_positive()` to `filter()` to remove all the negative numbers from `numbers`. So, the call to `map()` will process only positive numbers and `math.sqrt()` won't give you a `ValueError`.

`map()` and `reduce()`

Python's `reduce()` is a function that lives in a module called `functools` 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**. `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:

Python >>>

```
>>> 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
```

In this example, you call `os.path.expanduser("~")` to get the path to your home directory. Then you call `os.listdir()` on that path to get a list with the paths of all the files that live there.

The call to `map()` uses `os.path.getsize()` to get the size of every file. Finally, you use `reduce()` with `operator.add()` to get the cumulative sum of the size of every single file. The final result is the total size of all the files in your home directory in bytes.

> **Note:** Some years ago, Google developed and started using a programming model that they called MapReduce. It was a new style of data processing designed to manage big data using parallel and distributed computing on a cluster.

This model was inspired by the combination of the **map** and **reduce** operations commonly used in functional programming.

The MapReduce model had a huge impact on Google's ability to handle huge amounts of data in a reasonable time. However, by 2014 Google was no longer using MapReduce as their primary processing model.

Nowadays, you can find some alternative implementations of MapReduce like Apache Hadoop, which is a collection of open source software utilities that use the MapReduce model.

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:

Python >>>

```
>>> import os
>>> import os.path

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

>>> sum(map(os.path.getsize, files))
4377381
```

This example is a lot more readable and efficient than the example that you saw before. If you want to dive deeper into how to use `reduce()` and which alternative tools you can use to replace `reduce()` in a Pythonic way, then check out Python's reduce(): From Functional to Pythonic Style.

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 (`*`) 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()` says that the function is roughly equivalent to the following Python function:

Python

```
def starmap(function, iterable):
    for args in iterable:
        yield function(*args)
```

The `for` loop in this function iterates over the items in `iterable` and yields transformed items as a result. The call to `function(*args)` uses the unpacking operator to unpack the tuples into several positional arguments. Here are some examples of how `starmap()` works:

Python >>>

```
>>> from itertools import starmap

>>> list(starmap(pow, [(2, 7), (4, 3)]))
[128, 64]

>>> list(starmap(ord, [(2, 7), (4, 3)]))
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    list(starmap(ord, [(2, 7), (4, 3)]))
TypeError: ord() takes exactly one argument (2 given)
```

In the first example, you use `pow()` to calculate the power of the first value raised to the second value in each tuple. The tuples will be in the form `(base, exponent)`.

If every tuple in your iterable has two items, then `function` must take two arguments as well. If the tuples have three items, then `function` must take three arguments, and so on. Otherwise, you'll get a `TypeError`.

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

```
Python >>>
>>> list(map(pow, (2, 7), (4, 3)))
[16, 343]
```

Note that `map()` takes two tuples instead of a list of tuples. `map()` also takes one value from each tuple in every iteration. To make `map()` return the same result as `starmap()`, you'd need to swap values:

```
Python >>>
>>> list(map(pow, (2, 4), (7, 3)))
[128, 64]
```

In this case, you have two tuples instead of a list of tuples. You've also swapped `7` and `4`. Now the first tuple provides the bases and the second tuple provides the exponents.

Coding With Pythonic Style: Replacing `map()`

Functional programming tools like `map()`, `filter()`, and `reduce()` have been around for a long time. However, list comprehensions and generator expressions have become a natural replacement for them almost in every use case.

For example, the functionality provided by `map()` is almost always better expressed using a list comprehension or a generator expression. In the following two sections, you'll learn how to replace a call to `map()` with a list comprehension or a generator expression to make your code more readable and Pythonic.

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:

```
Python
# Generating a list with map
list(map(function, iterable))

# Generating a list with a list comprehension
[function(x) for x in iterable]
```

Note that the list comprehension almost always reads more clearly than the call to `map()`. Since list comprehensions are quite popular among Python developers, it's common to find them everywhere. So, replacing a call to `map()` with a list comprehension will make your code look more familiar to other Python developers.

Here's an example of how to replace `map()` with a list comprehension to build a list of square numbers:

```
Python >>>
>>> # Transformation function
>>> def square(number):
...     return number ** 2

>>> 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]
```

If you compare both solutions, then you might say that the one that uses the list comprehension is more readable because it reads almost like plain English. Additionally, list comprehensions avoid the need to explicitly call `list()` on `map()` to build the final list.

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 because generator expressions return generator objects, which are also iterators that yield items on demand.

Python iterators are known to be quite efficient in terms of memory consumption. This is the reason why `map()` now returns an iterator instead of a `list`.

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 (`()`). So, to turn a list comprehension into a generator expression, you just need to replace the square brackets with parentheses.

You can use generator expressions to write code that reads clearer than code that uses `map()`. Check out the following example:

Python >>>

```
>>> # 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]
```

This code has a main difference from the code in the previous section: you change the square brackets to a pair of parentheses to turn the list comprehension into a generator expression.

Generator expressions are commonly used as arguments in function calls. In this case, you don't need to use parentheses to create the generator expression because the parentheses that you use to call the function also provide the syntax to build the generator. With this idea, you can get the same result as the above example by calling `list()` like this:

Python >>>

```
>>> list(square(x) for x in numbers)
[1, 4, 9, 16, 25, 36]
```

If you use a generator expression as an argument in a function call, then you don't need an extra pair of parentheses. The parentheses that you use to call the function provide the syntax to build the generator.

Generator expressions are as efficient as `map()` in terms of memory consumption because both of them return iterators that yield items on demand. However, generator expressions will almost always improve your code's readability. They also make your code more Pythonic in the eyes of other Python developers.

Conclusion

Python's **`map()`** allows you to perform **mapping** 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 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. You can also switch to a more Pythonic and modern style by replacing `map()` with a list comprehension or a generator expression.

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

 Bryan

 Geir Arne

 Joanna

 Jacob

What Do You Think?

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Related Tutorial Categories: basics best-practices python

Recommended Video Course: Python's map() Function: Transforming Iterables