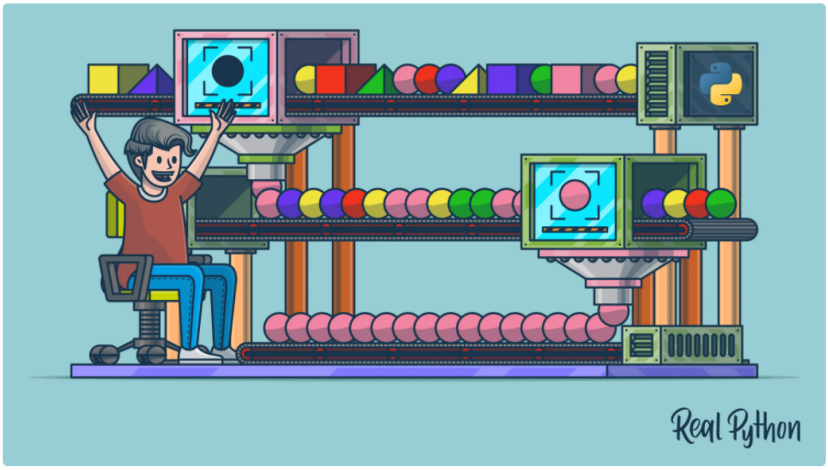


Python's filter(): Extract Values From Iterables

by Leodanis Pozo Ramos Jun 09, 2021 6 Comments best-practices intermediate python

Table of Contents

Python's `filter()` is a built-in function that allows you to process an iterable and extract those items that satisfy a given condition. This process is commonly known as a **filtering** operation. With `filter()`, you can apply a **filtering function** to an iterable and produce a new iterable with the items that satisfy the condition at hand. In Python, `filter()` is one of the tools you can use for functional programming.

In this tutorial, you'll learn how to:

- Use Python's `filter()` in your code

All Tutorial Topics

advanced api basics best-practices community databases data-science devops django docker flask front-end gamedev gui intermediate machine-learning projects python testing tools web-dev web-scraping

Table of Contents

- Coding With Functional Style in Python
- Understanding the Filtering Problem
- Getting Started With Python's filter()
- Filtering Iterables With filter()
- Combining filter() With Other Functional Tools
- Filtering Iterables With filterfalse()
- Coding With Pythonic Style
- Conclusion

- Extract **needed values** from your iterables
- Combine `filter()` with other **functional tools**
- **Replace** `filter()` with more **Pythonic** tools

With this knowledge, you'll be able to use `filter()` effectively in your code. Alternatively, you have the choice of using list comprehensions or generator expressions to write more Pythonic and readable code.

To better understand `filter()`, it would be helpful for you to have some previous knowledge on iterables, `for` loops, functions, and `lambda` functions.

Free Bonus: 5 Thoughts On Python Mastery, a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

Coding With Functional Style in Python

Functional programming is a paradigm that promotes using functions to perform almost every task in a program. A pure functional style relies on functions that don't modify their input arguments and don't change the program's state. They just take a specific set of arguments and return the same result every time. These kinds of functions are known as pure functions.

In functional programming, functions often operate on arrays of data, transform them, and produce new arrays with added features. There are three fundamental operations in functional programming:

1. Mapping applies a transformation function to an iterable and produces a new iterable of transformed items.
2. Filtering applies a predicate, or Boolean-valued, function to an iterable and generates a new iterable containing the items that satisfy the Boolean condition.
3. Reducing applies a reduction function to an iterable and returns a single cumulative value.

Python isn't heavily influenced by functional languages but by imperative ones. However, it provides several features that allow you to use a functional style:

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

Functions in Python are first-class objects, which means that you can pass them around as you'd do with any other object. You can also use them as arguments and return values of other functions. Functions that accept other functions as arguments or that return functions (or both) are known as higher-order functions, which are also a desirable feature in functional programming.

In this tutorial, you'll learn about `filter()`. This built-in function is one of the more popular functional tools of Python.

Understanding the Filtering Problem

Say you need to process a list of numbers and return a new list containing only those numbers greater than `0`. A quick way to approach this problem is to use a `for` loop like this:

Python >>>

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

```
>>> def extract_positive(numbers):
...     positive_numbers = []
...     for number in numbers:
...         if number > 0:  # Filtering condition
...             positive_numbers.append(number)
...     return positive_numbers
...

>>> extract_positive(numbers)
[1, 2]
```

The loop in `extract_positive()` iterates through `numbers` and stores every number greater than `0` in `positive_numbers`. The conditional statement *filters out* the negative numbers and `0`. This kind of functionality is known as a **filtering**.

Filtering operations consist of testing each value in an iterable with a predicate function and retaining only those values for which the function produces a true result. Filtering operations are fairly common in programming, so most programming languages provide tools to approach them. In the next section, you'll learn about Python's way to filter iterables.

Getting Started With Python's `filter()`

Python provides a convenient built-in function, `filter()`, that abstracts out the logic behind filtering operations. Here's its signature:

Python

```
filter(function, iterable)
```

The first argument, `function`, must be a single-argument function. Typically, you provide a predicate (Boolean-valued) function to this argument. In other words, you provide a function that returns either `True` or `False` according to a specific condition.

This `function` plays the role of a **decision function**, also known as a **filtering function**, because it provides the criteria to filter out unwanted values from the input iterable and to keep those values that you want in the resulting iterable. Note that the term **unwanted values** refers to those values that evaluate to false when `filter()` processes them using `function`.

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

The second argument, `iterable`, can hold any Python iterable, such as a list, tuple, or set. It can also hold generator and iterator objects. An important point regarding `filter()` is that it accepts only one `iterable`.

To perform the filtering process, `filter()` applies `function` to every item of `iterable` in a loop. The result is an iterator that yields the values of `iterable` for which `function` returns a true value. The process doesn't modify the original input iterable.

Since `filter()` is written in C and is highly optimized, its internal implicit loop can be more efficient than a regular `for` loop regarding execution time. This efficiency is arguably the most important advantage of using the function in Python.

A second advantage of using `filter()` over a loop is that it returns a `filter` object, which is an iterator that yields values on demand, promoting a lazy evaluation strategy. Returning an iterator makes `filter()` more memory efficient than an equivalent `for` loop.

> **Note:** In Python 2.x, `filter()` returns `list` objects. This behavior changed in Python 3.x. Now the function returns a `filter` object, which is an iterator that yields items on demand. Python iterators are well known to be memory efficient.

In your example about positive numbers, you can use `filter()` along with a convenient predicate function to extract the desired numbers. To code the predicate, you can use either

a `lambda` or a user-defined function:

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

>>> # Using a lambda function
>>> positive_numbers = filter(lambda n: n > 0, numbers)
>>> positive_numbers
<filter object at 0x7f3632683610>
>>> list(positive_numbers)
[1, 2]

>>> # Using a user-defined function
>>> def is_positive(n):
...     return n > 0
...
>>> list(filter(is_positive, numbers))
[1, 2]
```

In the first example, you use a `lambda` function that provides the filtering functionality. The call to `filter()` applies that `lambda` function to every value in `numbers` and filters out the negative numbers and `0`. Since `filter()` returns an iterator, you need to call `list()` to consume the iterator and create the final list.

> **Note:** Since `filter()` is a built-in function, you don't have to import anything to be able to use it in your code.

In the second example, you write `is_positive()` to take a number as an argument and return `True` if the number is greater than `0`. Otherwise, it returns `False`. The call to `filter()` applies `is_positive()` to every value in `numbers`, filtering out the negative numbers. This solution is way more readable than its `lambda` equivalent.

In practice, `filter()` isn't limited to Boolean functions such as those in the examples above. You can use other types of functions, and `filter()` will evaluate their return value for truthiness:

```
Python >>>
>>> def identity(x):
...     return x
...

>>> identity(42)
42

>>> objects = [0, 1, [], 4, 5, "", None, 8]
>>> list(filter(identity, objects))
[1, 4, 5, 8]
```

In this example, the filtering function, `identity()`, doesn't return `True` or `False` explicitly but the same argument it takes. Since `0`, `[]`, `""`, and `None` are falsy, `filter()` uses their **truth value** to filter them out. The final list contains only those values that are truthy in Python.

> **Note:** Python follows a set of rules to determine an object's truth value.
>
> For example, the following objects are falsy:
>
> - Constants like `None` and `False`
> - Numeric types with a zero value, like `0`, `0.0`, `0j`, `Decimal(0)`, and `Fraction(0, 1)`
> - Empty sequences and collections, like `""`, `()`, `[]`, `{}`, `set()`, and `range(0)`
> - Objects that implement `__bool__()` with a return value of `False` or `__len__()` with a return value of `0`
>
> Any other object will be considered truthy.

Finally, if you pass `None` to `function`, then `filter()` uses the **identity function** and yields all the elements of `iterable` that evaluate to `True`:

```
Python >>>
```

```
>>> objects = [0, 1, [], 4, 5, "", None, 8]

>>> list(filter(None, objects))
[1, 4, 5, 8]
```

In this case, `filter()` tests every item in the input iterable using the Python rules you saw before. Then it yields those items that evaluate to `True`.

So far, you've learned the basics of `filter()` and how it works. In the following sections, you'll learn how to use `filter()` to process iterables and throw away unwanted values without a loop.

Filtering Iterables With `filter()`

The job of `filter()` is to apply a decision function to each value in an input iterable and return a new iterable with those items that pass the test. The following sections provide some practical examples so you can get up and running with `filter()`.

Extracting Even Numbers

As a first example, say you need to process a list of integer numbers and build a new list containing the even numbers. Your first approach to this problem might be to use a `for` loop like this:

Python >>>

```
>>> numbers = [1, 3, 10, 45, 6, 50]

>>> def extract_even(numbers):
...     even_numbers = []
...     for number in numbers:
...         if number % 2 == 0:  # Filtering condition
...             even_numbers.append(number)
...     return even_numbers
...

>>> extract_even(numbers)
[10, 6, 50]
```

Here, `extract_even()` takes an iterable of integer numbers and returns a list containing only those that are even. The conditional statement plays the role of a filter that tests every number to find out if it's even or not.

When you run into code like this, you can extract the filtering logic into a small predicate function and use it with `filter()`. This way, you can perform the same computation without using an explicit loop:

Python >>>

```
>>> numbers = [1, 3, 10, 45, 6, 50]

>>> def is_even(number):
...     return number % 2 == 0  # Filtering condition
...

>>> list(filter(is_even, numbers))
[10, 6, 50]
```

Here, `is_even()` takes an integer and returns `True` if it's even and `False` otherwise. The call to `filter()` does the hard work and filters out the odd numbers. As a result, you get a list of the even numbers. This code is shorter and more efficient than its equivalent `for` loop.

Finding Prime Numbers

Finding Prime Numbers

Another interesting example might be to extract all the prime numbers in a given interval. To do that, you can start by coding a predicate function that takes an integer as an argument and returns `True` if the number is prime and `False` otherwise. Here's how you can do that:

Python >>>

```
>>> import math

>>> def is_prime(n):
...     if n <= 1:
...         return False
...     for i in range(2, int(math.sqrt(n)) + 1):
...         if n % i == 0:
...             return False
...     return True
...

>>> is_prime(5)
True
>>> is_prime(12)
False
```

The filtering logic is now in `is_prime()`. The function iterates through the integers between `2` and the square root of `n`. Inside the loop, the conditional statement checks if the current number is divisible by any other in the interval. If so, then the function returns `False` because the number isn't prime. Otherwise, it returns `True` to signal that the input number is prime.

With `is_prime()` in place and tested, you can use `filter()` to extract prime numbers from an interval like this:

Python >>>

```
>>> list(filter(is_prime, range(1, 51)))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

This call to `filter()` extracts all the prime numbers in the range between `1` and `50`. The algorithm used in `is_prime()` comes from Wikipedia's article about primality tests. You can check out that article if you need more efficient approaches.

Removing Outliers in a Sample

When you're trying to describe and summarize a sample of data, you probably start by finding its mean, or average. The mean is a quite popular central tendency measurement and is often the first approach to analyzing a dataset. It gives you a quick idea of the center, or **location**, of the data.

In some cases, the mean isn't a good enough central tendency measure for a given sample. Outliers are one of the elements that affect how accurate the mean is. Outliers are data points that differ significantly from other observations in a sample or population. Other than that, there is no unique mathematical definition for them in statistics.

However, in normally distributed samples, outliers are often defined as data points that lie more than two standard deviations from the sample mean.

Now suppose you have a normally distributed sample with some outliers that are affecting the mean accuracy. You've studied the outliers, and you know they're incorrect data points. Here's how you can use a couple of functions from the `statistics` module along with `filter()` to clean up your data:

Python >>>

```
>>> import statistics as st
>>> sample = [10, 8, 10, 8, 2, 7, 9, 3, 34, 9, 5, 9, 25]

>>> # The mean before removing outliers
>>> mean = st.mean(sample)
>>> mean
10.692307692307692
```

```
>>> stdev = st.stdev(sample)
>>> low = mean - 2 * stdev
>>> high = mean + 2 * stdev

>>> clean_sample = list(filter(lambda x: low <= x <= high, sample))
>>> clean_sample
[10, 8, 10, 8, 2, 7, 9, 3, 9, 5, 9, 25]

>>> # The mean after removing outliers
>>> st.mean(clean_sample)
8.75
```

In the highlighted line, the `lambda` function returns `True` if a given data point lies between the mean and two standard deviations. Otherwise, it returns `False`. When you filter the `sample` with this function, `34` is excluded. After this cleanup, the mean of the sample has a significantly different value.

Validating Python Identifiers

You can also use `filter()` with iterables containing nonnumeric data. For example, say you need to process a list of strings and extract those that are valid Python identifiers. After doing some research, you find out that Python's `str` provides a method called `.isidentifier()` that can help you out with that validation.

Here's how you can use `filter()` along with `str.isidentifier()` to quickly validate identifiers:

Python >>>

```
>>> words = ["variable", "file#", "header", "_non_public", "123Class"]

>>> list(filter(str.isidentifier, words))
['variable', 'header', '_non_public']
```

In this case, `filter()` runs `.isidentifier()` on every string in `words`. If the string is a valid Python identifier, then it's included in the final result. Otherwise, the word is filtered out. Note that you need to use `str` to access `.isidentifier()` in the call to `filter()`.

> **Note:** Besides `.isidentifier()`, `str` provides a rich set of `.is*()` methods that can be useful for filtering iterables of strings.

Finally, an interesting exercise might be to take the example further and check if the identifier is also a keyword. Go ahead and give it a try! Hint: you can use `.kwlist` from the `keyword` module.

Finding Palindrome Words

An exercise that often arises when you're getting familiar with Python strings is to find palindrome words in a list of strings. A palindrome word reads the same backward as forward. Typical examples are "madam" and "racecar."

To solve this problem, you'll start by coding a predicate function that takes a string and checks if it reads the same in both directions, backward and forward. Here's a possible implementation:

Python >>>

```
>>> def is_palindrome(word):
...     reversed_word = "".join(reversed(word))
...     return word.lower() == reversed_word.lower()
...

>>> is_palindrome("Racecar")
True
```

```
>>> is_palindrome("Python")
False
```

In `is_palindrome()`, you first reverse the original `word` and store it in `reversed_word`. Then you return the result of comparing both words for equality. In this case, you use `.lower()` to prevent case-related differences. If you call the function with a palindrome word, then you get `True`. Otherwise, you get `False`.

You already have a working predicate function to identify palindrome words. Here's how you can use `filter()` to do the hard work:

Python >>>

```
>>> words = ("filter", "Ana", "hello", "world", "madam", "racecar")

>>> list(filter(is_palindrome, words))
['Ana', 'madam', 'racecar']
```

Cool! Your combination of `filter()` and `is_palindrome()` works properly. It's also concise, readable, and efficient. Good job!

Combining `filter()` With Other Functional Tools

So far, you've learned how to use `filter()` to run different filtering operations on iterables. In practice, you can combine `filter()` with other functional tools to perform many different tasks on iterables without using explicit loops. In the next two sections, you'll learn the basics of using `filter()` along with `map()` and `reduce()`.

The Square of Even Numbers: `filter()` and `map()`

Sometimes you need to take an iterable, process each of its items with a **transformation function**, and produce a new iterable with the resulting items. In that case, you can use `map()`. The function has the following signature:

Python

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

The arguments work like this:

1. **`function`** holds the transformation function. This function should take as many arguments as iterables you pass into `map()`.
2. **`iterable`** holds a Python iterable. Note that you can provide several iterables to `map()`, but that's optional.

`map()` applies `function` to each item in `iterable` to transform it into a different value with additional features. Then `map()` yields each transformed item on demand.

To illustrate how you can use `filter()` along with `map()`, say you need to compute the square value of all the even numbers in a given list. In that case, you can use `filter()` to extract the even numbers and then `map()` to calculate the square values:

Python >>>

```
>>> numbers = [1, 3, 10, 45, 6, 50]

>>> def is_even(number):
...     return number % 2 == 0
...

>>> even_numbers = list(filter(is_even, numbers))
>>> even_numbers
[10, 6, 50]

>>> list(map(lambda n: n ** 2, even_numbers))
[100, 36, 2500]

>>> list(map(lambda n: n ** 2, filter(is_even, numbers)))
[100, 36, 2500]
```

First, you get the even numbers using `filter()` and `is_even()` just like you've done so far. Then you call `map()` with a `lambda` function that takes a number and returns its square value. The call to `map()` applies the `lambda` function to each number in `even_numbers`, so you get a list of square even numbers. The final example shows how to combine `filter()` and `map()` in a single expression.

The Sum of Even Numbers: `filter()` and `reduce()`

Another functional programming tool in Python is `reduce()`. Unlike `filter()` and `map()`, which are still built-in functions, `reduce()` was moved to the `functools` module. This function 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 a reduction or folding.

The signature of `reduce()` is like this:

Python

```
reduce(function, iterable, initial)
```

Here's what the arguments mean:

1. **`function`** holds any Python callable that accepts two arguments and returns a single value.
2. **`iterable`** holds any Python iterable.
3. **`initial`** holds a value that serves as a starting point for the first partial computation or reduction. It's an optional argument.

A call to `reduce()` starts by applying `function` to the first two items in `iterable`. This way, it computes the first cumulative result, called an **accumulator**. Then `reduce()` uses the accumulator and the third item in `iterable` to compute the next cumulative result. The process continues until the function returns with a single value.

If you supply a value to `initial`, then `reduce()` runs the first partial computation using `initial` and the first item of `iterable`.

Here's an example that combines `filter()` and `reduce()` to cumulatively calculate the total sum of all the even numbers in a list:

Python >>>

```
>>> from functools import reduce
>>> numbers = [1, 3, 10, 45, 6, 50]

>>> def is_even(number):
...     return number % 2 == 0
...

>>> even_numbers = list(filter(is_even, numbers))
>>> reduce(lambda a, b: a + b, even_numbers)
66

>>> reduce(lambda a, b: a + b, filter(is_even, numbers))
66
```

Here, the first call to `reduce()` computes the sum of all the even numbers that `filter()` provides. To do that, `reduce()` uses a `lambda` function that adds two numbers at a time.

The final example shows how to chain `filter()` and `reduce()` to produce the same result you got before.

Filtering Iterables With `filterfalse()`

In `itertools`, you'll find a function called `filterfalse()` that does the inverse of `filter()`. It

takes an iterable as argument and returns a new iterator that yields the items for which the decision function returns a false result. If you use `None` as the first argument to `filterfalse()`, then you get the items that are falsy.

The point of having the `filterfalse()` function is to promote **code reuse**. If you already have a decision function in place, then you can use it with `filterfalse()` to get the rejected items. This saves you from coding an inverse decision function.

In the following sections, you'll code some examples that show how you can take advantage of `filterfalse()` to reuse existing decision functions and continue doing some filtering.

Extracting Odd Numbers

You already coded a predicate function called `is_even()` to check if a number is even or not. With that function and the help of `filterfalse()`, you can build an iterator that yields odd numbers without having to code an `is_odd()` function:

Python >>>

```
>>> from itertools import filterfalse
>>> numbers = [1, 3, 10, 45, 6, 50]

>>> def is_even(number):
...     return number % 2 == 0
...

>>> list(filterfalse(is_even, numbers))
[1, 3, 45]
```

In this example, `filterfalse()` returns an iterator that yields the odd numbers from the input iterator. Note that the call to `filterfalse()` is straightforward and readable.

Filtering Out NaN Values

Sometimes when you're working with floating-point arithmetic, you can face the issue of having NaN (not a number) values. For example, say you're calculating the mean of a sample of data that contains NaN values. If you use Python's `statistics` module for this computation, then you get the following result:

Python >>>

```
>>> import statistics as st

>>> sample = [10.1, 8.3, 10.4, 8.8, float("nan"), 7.2, float("nan")]
>>> st.mean(sample)
nan
```

In this example, the call to `mean()` returns `nan`, which isn't the most informative value you can get. NaN values can have different origins. They can be due to invalid inputs, corrupted data, and so on. You should find the right strategy to deal with them in your applications. One alternative might be to remove them from your data.

The `math` module provides a convenient function called `isnan()` that can help you out with this problem. The function takes a number `x` as an argument and returns `True` if `x` is a NaN and `False` otherwise. You can use this function to provide the filtering criteria in a `filterfalse()` call:

Python >>>

```
>>> import math
>>> import statistics as st
>>> from itertools import filterfalse

>>> sample = [10.1, 8.3, 10.4, 8.8, float("nan"), 7.2, float("nan")]
```

```
>>> st.mean(filterfalse(math.isnan, sample))
8.96
```

Using `math.isnan()` along with `filterfalse()` allows you to exclude all the NaN values from the mean computation. Note that after the filtering, the call to `mean()` returns a value that provides a better description of your sample data.

Coding With Pythonic Style

Even though `map()`, `filter()`, and `reduce()` have been around for a long time in the Python ecosystem, **list comprehensions** and **generator expressions** have become strong and Pythonic competitors in almost every use case.

The functionality these functions provide is almost always more explicitly expressed using a generator expression or a list comprehension. In the following two sections, you'll learn how to replace a call to `filter()` with a list comprehension or a generator expression. This replacement will make your code more Pythonic.

Replacing `filter()` With a List Comprehension

You can use the following pattern to quickly replace a call to `filter()` with an equivalent list comprehension:

Python

```
# Generating a list with filter()
list(filter(function, iterable))

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

In both cases, the final purpose is to create a list object. The list comprehension approach is more explicit than its equivalent `filter()` construct. A quick read through the comprehension reveals the iteration and also the filtering functionality in the `if` clause.

Using list comprehensions instead of `filter()` is probably the path most Python developers take nowadays. However, list comprehensions have some drawbacks compared to `filter()`. The most notable one is the lack of lazy evaluation. Also, when developers start reading code that uses `filter()`, they immediately know that the code is performing filtering operations. However, that's not so evident in code that uses list comprehensions with the same goal.

A detail to notice when turning a `filter()` construct into a list comprehension is that if you pass `None` to the first argument of `filter()`, then the equivalent list comprehension looks like this:

Python

```
# Generating a list with filter() and None
list(filter(None, iterable))

# Equivalent list comprehension
[item for item in iterable if item]
```

In this case, the `if` clause in the list comprehension tests `item` for its truth value. This test follows the standard Python rules about truth values you already saw.

Here's an example of replacing `filter()` with a list comprehension to build a list of even numbers:

Python >>>

```
>>> numbers = [1, 3, 10, 45, 6, 50]

>>> # Filtering function
>>> def is_even(x):
...     return x % 2 == 0
...

>>> # Use filter()
```

```
>>> list(filter(is_even, numbers))
[10, 6, 50]

>>> # Use a list comprehension
>>> [number for number in numbers if is_even(number)]
[10, 6, 50]
```

In this example, you can see that the list comprehension variant is more explicit. It reads almost like plain English. The list comprehension solution also avoids having to call `list()` to build the final list.

Replacing `filter()` With a Generator Expression

The natural replacement for `filter()` is a **generator expression**. That's because `filter()` returns an iterator that yields items on demand just like a generator expression does. Python iterators are known to be memory efficient. That's why `filter()` now returns an iterator instead of a list.

Here's how you can use generator expressions to write the example in the above section:

Python >>>

```
>>> numbers = [1, 3, 10, 45, 6, 50]

>>> # Filtering function
>>> def is_even(x):
...     return x % 2 == 0
...

>>> # Use filter()
>>> even_numbers = filter(is_even, numbers)
>>> even_numbers
<filter object at 0x7f58691de4c0>
>>> list(even_numbers)
[10, 6, 50]

>>> # Use a generator expression
>>> even_numbers = (number for number in numbers if is_even(number))
>>> even_numbers
<generator object <genexpr> at 0x7f586ade04a0>
>>> list(even_numbers)
[10, 6, 50]
```

A generator expression is as efficient as a call to `filter()` in terms of memory consumption. Both tools return iterators that yield items on demand. Using either one might be a question of taste, convenience, or style. So, you're in charge!

Conclusion

Python's `filter()` allows you to perform **filtering** operations on iterables. This kind of operation consists of applying a **Boolean function** to the items in an iterable and keeping only those values for which the function returns a true result. In general, you can use `filter()` to process existing iterables and produce new iterables containing the values that you currently need.

In this tutorial, you learned how to:

- Work with Python's `filter()`
- Use `filter()` to **process iterables** and keep the values you need
- Combine `filter()` with `map()` and `reduce()` to approach different problems
- Replace `filter()` with **list comprehensions** and **generator expressions**

With this new knowledge, you can now use `filter()` in your code to give it a functional style. You can also switch to a more Pythonic style and replace `filter()` with list comprehensions

or generator expressions.

Mark as Completed

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.

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

About **Leodanis Pozo Ramos**

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

» More about Leodanis

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

 Aldren

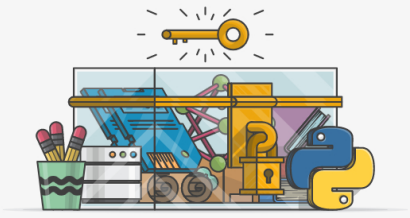
 Bartosz

 David

 Joanna

 Jacob

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



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

Level Up Your Python Skills »

What Do You Think?

Tweet Share Email

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

Help