Generator Expressions in Python: An Introduction

dbader.org/blog/python-generator-expressions

By Dan Bader — Get free updates of new posts here.

In one of my previous tutorials you saw how Python's generator functions and the yield keyword provide syntactic sugar for writing class-based iterators more easily.

The *generator expressions* we'll cover in this tutorial add another layer of syntactic sugar on top—they give you an even more effective shortcut for writing iterators:

With a simple and concise syntax that looks like a list comprehension, you'll be able to define iterators in a single line of code.

Here's an example:

```
iterator = ('Hello' for i in range(3))
```

Python Generator Expressions 101 – The Basics

When iterated over, the above generator expression yields the same sequence of values as the `bounded_repeater` generator function we implemented in my generators tutorial. Here it is again to refresh your memory:

```
def bounded_repeater(value, max_repeats):
    for i in range(max_repeats):
        yield value

iterator = bounded_repeater('Hello', 3)
```

Isn't it amazing how a single-line generator expression now does a job that previously required a four-line generator function or a much longer class-based iterator?

But I'm getting ahead of myself. Let's make sure our iterator defined with a generator expression actually works as expected:

```
>>> iterator = ('Hello' for i in range(3))
>>> for x in iterator:
...     print(x)
'Hello'
'Hello'
'Hello'
```

That looks pretty good to me! We seem to get the same results from our one-line generator expression that we got from the `bounded_repeater` generator function.

There's one small caveat though:

Once a generator expression has been consumed, it can't be restarted or reused. So in some cases there is an advantage to using generator functions or class-based iterators.

Generator Expressions vs List Comprehensions

As you can tell, generator expressions are somewhat similar to list comprehensions:

```
>>> listcomp = ['Hello' for i in range(3)]
>>> genexpr = ('Hello' for i in range(3))
```

Unlike list comprehensions, however, generator expressions don't construct list objects. Instead, they generate values "just in time" like a class-based iterator or generator function would.

All you get by assigning a generator expression to a variable is an iterable "generator object":

```
>>> listcomp
['Hello', 'Hello', 'Hello']

>>> genexpr
<generator object <genexpr> at 0x1036c3200>
```

To access the values produced by the generator expression, you need to call `next()` on it, just like you would with any other iterator:

```
>>> next(genexpr)
'Hello'
>>> next(genexpr)
'Hello'
>>> next(genexpr)
'Hello'
>>> next(genexpr)
StopIteration
```

Alternatively, you can also call the `list()` function on a generator expression to construct a list object holding all generated values:

```
>>> genexpr = ('Hello' for i in range(3))
>>> list(genexpr)
['Hello', 'Hello', 'Hello']
```

Of course, this was just a toy example to show how you can "convert" a generator expression (or any other iterator for that matter) into a list. If you need a list object right away, you'd normally just write a list comprehension from the get-go.

Let's take a closer look at the syntactic structure of this simple generator expression. The pattern you should begin to see looks like this:

```
genexpr = (expression for item in collection)
```

The above generator expression "template" corresponds to the following generator function:

```
def generator():
    for item in collection:
        yield expression
```

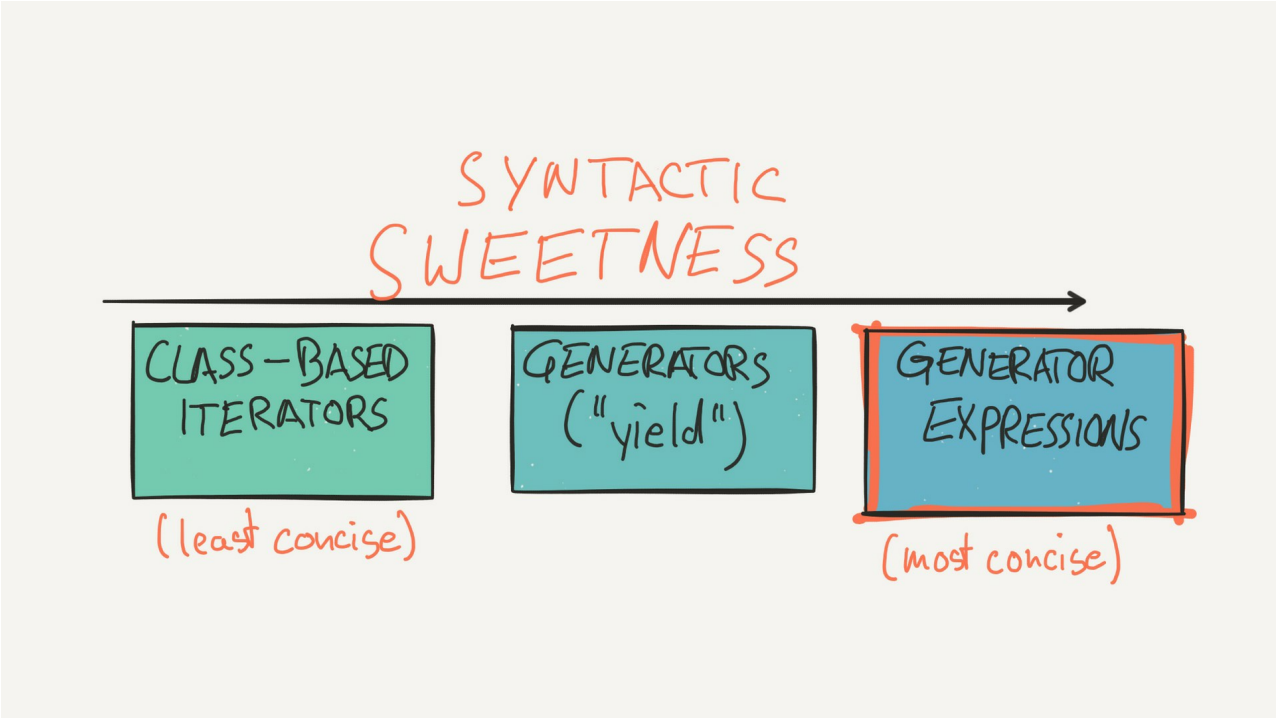
Just like with list comprehensions, this gives you a "cookie-cutter pattern" you can apply to many generator functions in order to transform them into concise *generator expressions*.

⏱ Sidebar: Pythonic Syntactic Sugar

As I learned more about Python's iterator protocol and the different ways to implement it in my own code, I realized that "syntactic sugar" was a recurring theme.

You see, class-based iterators and generator functions are two expressions of the same underlying design pattern.

Generator functions give you a shortcut for supporting the iterator protocol in your own code, and they avoid much of the verbosity of class-based iterators. With a little bit of specialized syntax, or *syntactic sugar*, they save you time and make your life as a developer easier:



This is a recurring theme in Python and in other programming languages. As more developers use a design pattern in their programs, there's a growing incentive for the language creators to provide abstractions and implementation shortcuts for it.

That's how programming languages evolve over time—and as developers, we reap the benefits. We get to work with more and more powerful building blocks, which reduces busywork and lets us achieve more in less time.

Filtering Values

There's one more useful addition we can make to this template, and that's element filtering with conditions. Here's an example:

```
>>> even_squares = (x * x for x in range(10)
                    if x % 2 == 0)
```

This generator yields the square numbers of all even integers from zero to nine. The filtering condition using the `%` (modulo) operator will reject any value not divisible by two:

```
>>> for x in even_squares:
...     print(x)
0
4
16
36
64
```

Let's update our generator expression template. After adding element filtering via `if`-conditions, the template now looks like this:

```
genexpr = (expression for item in collection
          if condition)
```

And once again, this pattern corresponds to a relatively straightforward, but longer, generator function. Syntactic sugar at its best:

```
def generator():
    for item in collection:
        if condition:
            yield expression
```

In-line Generator Expressions

Because generator expressions are, well...expressions, you can use them in-line with other statements. For example, you can define an iterator and consume it right away with a `for` -loop:

```
for x in ('Bom dia' for i in range(3)):
    print(x)
```

There's another syntactic trick you can use to make your generator expressions more beautiful. The parentheses surrounding a generator expression can be dropped if the generator expression is used as the single argument to a function:

```
>>> sum((x * 2 for x in range(10)))
90

# Versus:

>>> sum(x * 2 for x in range(10))
90
```

This allows you to write concise and performant code. Because generator expressions generate values "just in time" like a class-based iterator or a generator function would, they are very memory efficient.

Too Much of a Good Thing…

Like list comprehensions, generator expressions allow for more complexity than what we've covered so far. Through nested `for` -loops and chained filtering clauses, they can cover a wider range of use cases:

```
(expr for x in xs if cond1
      for y in ys if cond2
      ...
      for z in zs if condN)
```

The above pattern translates to the following generator function logic:

```
for x in xs:
    if cond1:
        for y in ys:
            if cond2:
                ...
                    for z in zs:
                        if condN:
                            yield expr
```

And this is where I'd like to place a big caveat:

Please don't write deeply nested generator expressions like that. They can be very difficult to maintain in the long run.

This is one of those "the dose makes the poison" situations where a beautiful and simple tool can be overused to create hard to read and difficult to debug programs.

Just like with list comprehensions, I personally try to stay away from any generator expression that includes more than two levels of nesting.

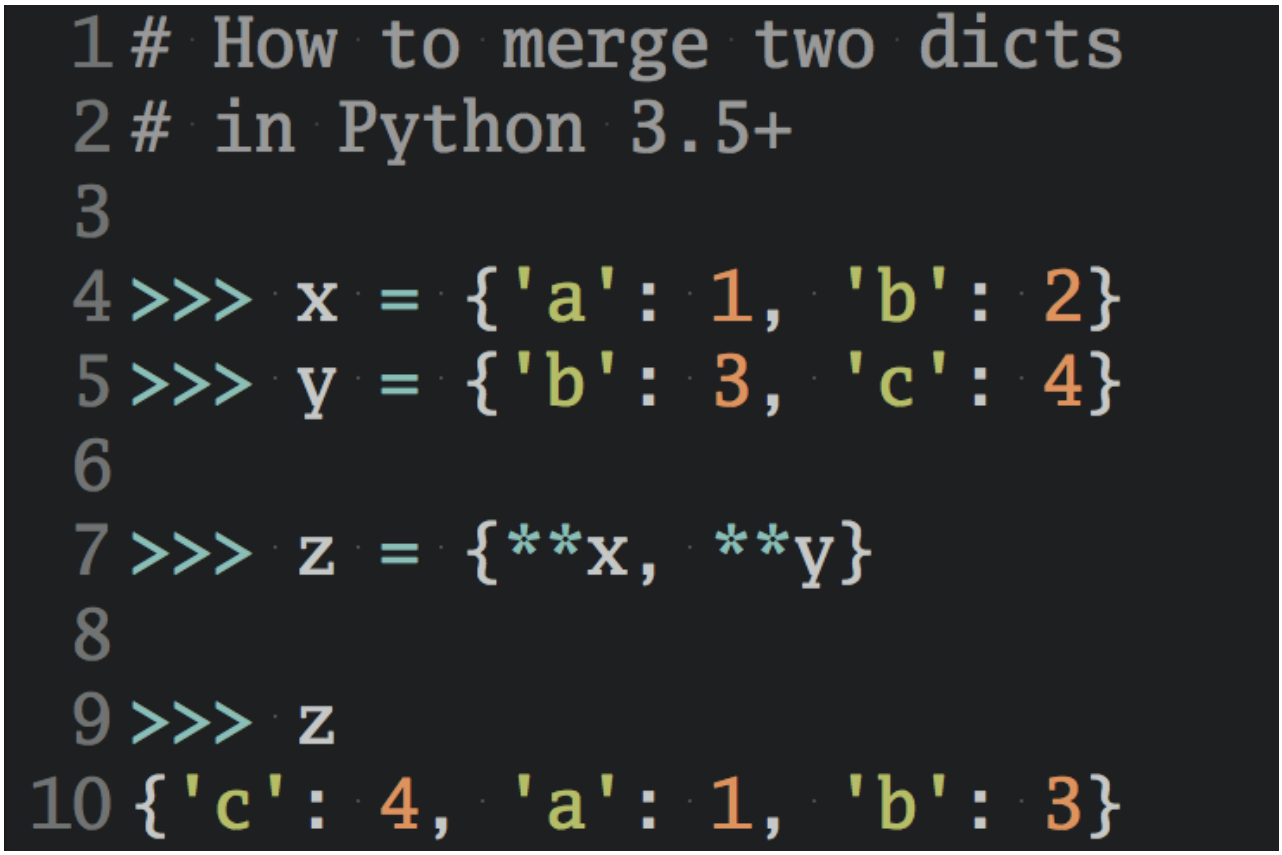
Generator expressions are a helpful and Pythonic tool in your toolbox, but that doesn't mean they should be used for every single problem you're facing. For complex iterators, it's often better to write a generator function or even a class-based iterator.

If you need to use nested generators and complex filtering conditions, it's usually better to factor out sub-generators (so you can name them) and then to chain them together again at the top level.

If you're on the fence, try out different implementations and then select the one that seems the most readable. Trust me, it'll save you time in the long run.

Generator Expressions in Python – Summary

- Generator expressions are similar to list comprehensions. However, they don't construct list objects. Instead, generator expressions generate values "just in time" like a class-based iterator or generator function would.
- Once a generator expression has been consumed, it can't be restarted or reused.
- Generator expressions are best for implementing simple "ad hoc" iterators. For complex iterators, it's better to write a generator function or a class-based iterator.



```
# 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}
```

This article was filed under: python.

Related Articles:

- What Are Python Generators? – Generators are a tricky subject in Python. With this tutorial you'll make the leap from class-based iterators to using generator functions and the "yield" statement in no time.
- A Python Riddle: The Craziest Dict Expression in the West – Let's pry apart this slightly unintuitive Python dictionary expression to find out what's going on in the uncharted depths of the Python interpreter.
- Lambda Functions in Python: What Are They Good For? – An introduction to "lambda" expressions in Python: What they're good for, when you should use them, and when it's best to avoid them.
- Comprehending Python's Comprehensions – One of my favorite features in Python are list comprehensions. They can seem a bit arcane at first but when you break them down they are actually a very simple construct.
- Python's Functions Are First-Class – Python's functions are first-class objects. You can assign them to variables, store them in data structures, pass them as arguments to other functions, and even return them as values from other functions.

Latest Articles:

- Interfacing Python and C: The CFFI Module – How to use Python's built-in CFFI module for interfacing Python with native libraries as an alternative to the "ctypes" approach.
- Write More Pythonic Code by Applying the Things You Already Know – There's a mistake I frequently make when I learn new things about Python... Here's how you can avoid this pitfall and learn something about Python's "enumerate()" function at the same time.
- Working With File I/O in Python – Learn the basics of working with files in Python. How to read from files, how to write data to them, what file seeks are, and why files should be closed.
- How to Reverse a String in Python – An overview of the three main ways to reverse a Python string: "slicing", reverse iteration, and the classic in-place reversal algorithm. Also includes performance benchmarks.
- Mastering Click: Writing Advanced Python Command-Line Apps – How to improve your existing Click Python CLIs with advanced features like sub-commands, user input, parameter types, contexts, and more.
- Working with Random Numbers in Python – An overview for working with randomness in Python, using only functionality built into the standard library and CPython itself.

← Browse All Articles