Jeff Knupp

PYTHON PROGRAMMER

Everything I know about Python...

Learn to Write Pythonic Code!

Check out the book *Writing Idiomatic Python*!

Looking for Python Tutoring? Remote and local (NYC) slots still available! Email me at jeff@jeffknupp.com for more info.

Improve Your Python: 'yield' and Generators Explained

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Prior to beginning tutoring sessions, I ask new students to fill out a brief self-assessment where they rate their understanding of various Python concepts. Some topics ("control flow with if/else" or "defining and using functions") are understood by a majority of students before ever beginning tutoring. There are a handful of topics, however, that almost all students report having no knowledge or *very* limited understanding of. Of these, "`generators` and the `yield` keyword" is one of the biggest culprits. I'm guessing this is the case for *most* novice Python programmers.

Many report having difficulty understanding `generators` and the `yield` keyword even after making a concerted effort to teach themselves the topic. I want to change that. In this post, I'll explain *what* the `yield` keyword does, *why* it's useful, and *how* to use it.

What is a Python Generator (Textbook Definition)

A Python *generator* is a function which returns a *generator iterator* (just an object we can iterate over) by calling `yield`. `yield` may be called with a value, in which case that value is treated as the "generated" value. The *next time* `next()` is called on the *generator iterator* (i.e. in the next step in a `for` loop, for example),Rthe generator resumes execution *from where it called* `yield`, not from the beginning of the function. All of the state, like the values of local variables, is recovered and the generator contiues to execute until the next call to `yield`.

If this doesn't make *any* sense to you, don't worry. I wanted to get the textbook definition out of the way so I can explain to you what all that nonsense actually means.

Note: In recent years, generators have grown more powerful as features have been added through PEPs. In my next post, I'll explore the true power of `yield` with respect to coroutines, cooperative multitasking and asynchronous I/O (especially their use in the "tulip" prototype implementation GvR has been working on). Before we get there, however, we need a solid understanding of how the `yield` keyword and `generators` work.

Coroutines and Subroutines

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

When we call a normal Python function, execution starts at function's first line and continues until a `return` statement, `exception`, or the end of the function (which is seen as an implicit `return None`) is encountered. Once a function returns control to its caller, that's it. Any work done by the function and stored in local variables is lost. A new call to the function creates everything from scratch.

This is all very standard when discussing functions (more generally referred to as subroutines) in computer programming. There are times, though, when it's beneficial to have the ability to create a "function" which, instead of simply returning a single value, is able to yield a series of values. To do so, such a function would need to be able to "save its work," so to speak.

I said, "yield a series of values" because our hypothetical function doesn't "return" in the normal sense. `return` implies that the function is *returning control of execution* to the point where the function was called. "Yield," however, implies that *the transfer of control is temporary and voluntary*, and our function expects to regain it in the future.

In Python, "functions" with these capabilities are called `generators`, and they're incredibly useful. `generators` (and the `yield` statement) were initially introduced to give programmers a more straightforward way to write code responsible for producing a series of values. Previously, creating something like a random number generator required a class or module that both generated values and kept track of state between calls. With the introduction of `generators`, this became much simpler.

To better understand the problem `generators` solve, let's take a look at an example. Throughout the example, keep in mind the core problem being solved: **generating a series of values.**

Note: Outside of Python, all but the simplest `generators` would be referred to as `coroutines`. I'll use the latter term later in the post. The important thing to remember is, in Python, everything described here as a `coroutine` is still a `generator`. Python formally defines the term `generator`; `coroutine` is used in discussion but has no formal definition in the language.

Example: Fun With Prime Numbers

Suppose our boss asks us to write a function that takes a `list` of `int`s and returns some Iterable containing the elements which are prime[1] numbers.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Remember, an Iterable is just an object capable of returning its members one at a time.

"Simple," we say, and we write the following:

```python
def get_primes(input_list):
    result_list = list()
    for element in input_list:
        if is_prime(element):
            result_list.append()

    return result_list

# or better yet...

def get_primes(input_list):
    return (element for element in input_list if is_prime(element))

# not germane to the example, but here's a possible implementation of
# is_prime...

def is_prime(number):
    if number > 1:
        if number == 2:
            return True
        if number % 2 == 0:
            return False
        for current in range(3, int(math.sqrt(number) + 1), 2):
            if number % current == 0:
                return False
        return True
    return False
```

Either `get_primes` implementation above fulfills the requirements, so we tell our boss we're done. She reports our function works and is exactly what she wanted.

Dealing With Infinite Sequences

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Well, not quite *exactly*. A few days later, our boss comes back and tells us she's run into a small problem: she wants to use our `get_primes` function on a very large list of numbers. In fact, the list is so large that merely creating it would consume all of the system's memory. To work around this, she wants to be able to call `get_primes` with a `start` value and get all the primes larger than `start` (perhaps she's solving Project Euler problem 10).

Once we think about this new requirement, it becomes clear that it requires more than a simple change to `get_primes`. Clearly, we can't return a list of all the prime numbers from `start` to infinity *(operating on infinite sequences, though, has a wide range of useful applications)*. The chances of solving this problem using a normal function seem bleak.

Before we give up, let's determine the core obstacle preventing us from writing a function that satisfies our boss's new requirements. Thinking about it, we arrive at the following: **functions only get one chance to return results, and thus must return all results at once.** It seems pointless to make such an obvious statement; "functions just work that way," we think. The real value lies in asking, "but what if they didn't?"

Imagine what we could do if `get_primes` could simply return the *next* value instead of all the values at once. It wouldn't need to create a list at all. No list, no memory issues. Since our boss told us she's just iterating over the results, she wouldn't know the difference.

Unfortunately, this doesn't seem possible. Even if we had a magical function that allowed us to iterate from `n` to `infinity`, we'd get stuck after returning the first value:

```
def get_primes(start):
    for element in magical_infinite_range(start):
        if is_prime(element):
            return element
```

Imagine `get_primes` is called like so:

```
def solve_number_10():
    # She *is* working on Project Euler #10, I knew it!
    total = 2
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
    for next_prime in get_primes(3):
        if next_prime < 2000000:
            total += next_prime
        else:
            print(total)
            return
```

Clearly, in `get_primes`, we would immediately hit the case where `number = 3` and return at line 4. Instead of `return`, we need a way to generate a value and, when asked for the next one, pick up where we left off.

Functions, though, can't do this. When they `return`, they're done for good. Even if we could guarantee a function would be called again, we have no way of saying, "OK, now, instead of starting at the first line like we normally do, start up where we left off at line 4." Functions have a single `entry point`: the first line.

Enter the Generator

This sort of problem is so common that a new construct was added to Python to solve it: the `generator`. A `generator` "generates" values. Creating `generators` was made as straightforward as possible through the concept of `generator functions`, introduced simultaneously.

A `generator function` is defined like a normal function, but whenever it needs to generate a value, it does so with the `yield` keyword rather than `return`. If the body of a `def` contains `yield`, the function automatically becomes a `generator function` (even if it also contains a `return` statement). There's nothing else we need to do to create one.

`generator functions` create `generator iterators`. That's the last time you'll see the term `generator iterator`, though, since they're almost always referred to as "`generators`". Just remember that a `generator` is a special type of `iterator`. To be considered an `iterator`, `generators` must define a few methods, one of which is `__next__()`. To get the next value from a `generator`, we use the same built-in function as for `iterators`: `next()`.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

This point bears repeating: **to get the next value from a `generator`, we use the same built-in function as for `iterators`: `next()`**.

(`next()` takes care of calling the generator's `__next__()` method). Since a `generator` is a type of `iterator`, it can be used in a `for` loop.

So whenever `next()` is called on a `generator`, the `generator` is responsible for passing back a value to whomever called `next()`. It does so by calling `yield` along with the value to be passed back (e.g. `yield 7`). The easiest way to remember what `yield` does is to think of it as `return` (plus a little magic) for `generator functions`.**

Again, this bears repeating: **`yield` is just `return` (plus a little magic) for `generator functions`.**

Here's a simple `generator function`:

```
>>> def simple_generator_function():
>>>    yield 1
>>>    yield 2
>>>    yield 3
```

And here are two simple ways to use it:

```
>>> for value in simple_generator_function():
>>>     print(value)
1
2
3
>>> our_generator = simple_generator_function()
>>> next(our_generator)
1
>>> next(our_generator)
2
>>> next(our_generator)
3
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Magic?

What's the magic part? Glad you asked! When a `generator function` calls `yield`, the "state" of the `generator function` is frozen; the values of all variables are saved and the next line of code to be executed is recorded until `next()` is called again. Once it is, the `generator function` simply resumes where it left off. If `next()` is never called again, the state recorded during the `yield` call is (eventually) discarded.

Let's rewrite `get_primes` as a `generator function`. Notice that we no longer need the `magical_infinite_range` function. Using a simple `while` loop, we can create our own infinite sequence:

```
def get_primes(number):
    while True:
        if is_prime(number):
            yield number
        number += 1
```

If a `generator function` calls `return` or reaches the end its definition, a `StopIteration` exception is raised. This signals to whoever was calling `next()` that the `generator` is exhausted (this is normal `iterator` behavior). It is also the reason the `while True:` loop is present in `get_primes`. If it weren't, the first time `next()` was called we would check if the number is prime and possibly yield it. If `next()` were called again, we would uselessly add `1` to `number` and hit the end of the `generator function` (causing `StopIteration` to be raised). Once a generator has been exhausted, calling `next()` on it will result in an error, so you can only consume all the values of a `generator` once. The following will not work:

```
>>> our_generator = simple_generator_function()
>>> for value in our_generator:
>>>     print(value)

>>> # our_generator has been exhausted...
>>> print(next(our_generator))
Traceback (most recent call last):
  File "<ipython-input-13-7e48a609051a>", line 1, in <module>
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
    next(our_generator)
StopIteration

>>> # however, we can always create a new generator
>>> # by calling the generator function again...

>>> new_generator = simple_generator_function()
>>> print(next(new_generator)) # perfectly valid
1
```

Thus, the `while` loop is there to make sure we *never* reach the end of `get_primes`. It allows us to generate a value for as long as `next()` is called on the generator. This is a common idiom when dealing with infinite series (and `generators` in general).

Visualizing the flow

Let's go back to the code that was calling `get_primes`: `solve_number_10`.

```
def solve_number_10():
    # She *is* working on Project Euler #10, I knew it!
    total = 2
    for next_prime in get_primes(3):
        if next_prime < 2000000:
            total += next_prime
        else:
            print(total)
            return
```

It's helpful to visualize how the first few elements are created when we call `get_primes` in `solve_number_10`'s `for` loop. When the `for` loop requests the first value from `get_primes`, we enter `get_primes` as we would in a normal function.

1. We enter the `while` loop on line 3
2. The `if` condition holds (`3` is prime)

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

3. We yield the value `3` and control to `solve_number_10`.

Then, back in `solve_number_10`:

1. The value `3` is passed back to the `for` loop
2. The `for` loop assigns `next_prime` to this value
3. `next_prime` is added to `total`
4. The `for` loop requests the next element from `get_primes`

This time, though, instead of entering `get_primes` back at the top, we resume at line `5`, where we left off.

```
def get_primes(number):
    while True:
        if is_prime(number):
            yield number
        number += 1 # <<<<<<<<<<
```

Most importantly, `number` *still has the same value it did when we called* `yield` (i.e. `3`). Remember, `yield` both passes a value to whoever called `next()`, *and* saves the "state" of the `generator function`. Clearly, then, `number` is incremented to `4`, we hit the top of the `while` loop, and keep incrementing `number` until we hit the next prime number (`5`). Again we `yield` the value of `number` to the `for` loop in `solve_number_10`. This cycle continues until the `for` loop stops (at the first prime greater than `2,000,000`).

Moar Power

In PEP 342, support was added for passing values *into* generators. PEP 342 gave `generator`s the power to yield a value (as before), *receive* a value, or both yield a value *and* receive a (possibly different) value in a single statement.

To illustrate how values are sent to a `generator`, let's return to our prime number example. This time, instead of simply printing every prime number greater than `number`, we'll find the smallest prime number greater than successive powers of a

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

number (i.e. for 10, we want the smallest prime greater than 10, then 100, then 1000, etc.). We start in the same way as `get_primes`:

```
def print_successive_primes(iterations, base=10):
    # like normal functions, a generator function
    # can be assigned to a variable

    prime_generator = get_primes(base)
    # missing code...
    for power in range(iterations):
        # missing code...

def get_primes(number):
    while True:
        if is_prime(number):
        # ... what goes here?
```

The next line of `get_primes` takes a bit of explanation. While `yield number` would yield the value of `number`, a statement of the form `other = yield foo` means, "yield `foo` and, when a value is sent to me, set `other` to that value." You can "send" values to a generator using the generator's `send` method.

```
def get_primes(number):
    while True:
        if is_prime(number):
            number = yield number
        number += 1
```

In this way, we can set `number` to a different value each time the generator `yield`s. We can now fill in the missing code in `print_successive_primes`:

```
def print_successive_primes(iterations, base=10):
    prime_generator = get_primes(base)
    prime_generator.send(None)
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
    for power in range(iterations):
        print(prime_generator.send(base ** power))
```

Two things to note here: First, we're printing the result of `generator.send`, which is possible because `send` both sends a value to the generator *and* returns the value yielded by the generator (mirroring how `yield` works from within the `generator function`).

Second, notice the `prime_generator.send(None)` line. When you're using send to "start" a generator (that is, execute the code from the first line of the generator function up to the first `yield` statement), you must send `None`. This makes sense, since by definition the generator hasn't gotten to the first `yield` statement yet, so if we sent a real value there would be nothing to "receive" it. Once the generator is started, we can send values as we do above.

Round-up

In the second half of this series, we'll discuss the various ways in which `generators` have been enhanced and the power they gained as a result. `yield` has become one of the most powerful keywords in Python. Now that we've built a solid understanding of how `yield` works, we have the knowledge necessary to understand some of the more "mind-bending" things that `yield` can be used for.

Believe it or not, we've barely scratched the surface of the power of `yield`. For example, while `send` *does* work as described above, it's almost never used when generating simple sequences like our example. Below, I've pasted a small demonstration of one common way `send` *is* used. I'll not say any more about it as figuring out how and why it works will be a good warm-up for part two.

```
import random

def get_data():
    """Return 3 random integers between 0 and 9"""
    return random.sample(range(10), 3)

```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
def consume():
    """Displays a running average across lists of integers sent to it"""
    running_sum = 0
    data_items_seen = 0

    while True:
        data = yield
        data_items_seen += len(data)
        running_sum += sum(data)
        print('The running average is {}'.format(running_sum / float(data_items_seen)))

def produce(consumer):
    """Produces a set of values and forwards them to the pre-defined consumer
    function"""
    while True:
        data = get_data()
        print('Produced {}'.format(data))
        consumer.send(data)
        yield

if __name__ == '__main__':
    consumer = consume()
    consumer.send(None)
    producer = produce(consumer)

    for _ in range(10):
        print('Producing...')
        next(producer)
```

Remember...

There are a few key ideas I hope you take away from this discussion:

- `generators` are used to *generate* a series of values
- `yield` is like the `return` of `generator functions`

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

- The only other thing `yield` does is save the "state" of a `generator function`
- A `generator` is just a special type of `iterator`
- Like `iterators`, we can get the next value from a `generator` using `next()`
  - `for` gets values by calling `next()` implicitly

I hope this post was helpful. If you had never heard of `generators`, I hope you now understand what they are, why they're useful, and how to use them. If you were somewhat familiar with `generators`, I hope any confusion is now cleared up.

As always, if any section is unclear (or, more importantly, contains errors), by all means let me know. You can leave a comment below, email me at jeff@jeffknupp.com, or hit me up on Twitter @jeffknupp.

---

1. Quick refresher: a prime number is a positive integer greater than 1 that has no divisors other than 1 and itself. 3 is prime because there are no numbers that evenly divide it other than 1 and 3 itself. ↩

Posted on Apr 07, 2013 by Jeff Knupp

Tweet

Like Share 216 people like this. Sign Up to see what your friends like.

« And Now for Something Completely Different...

Like this article?

Why not sign up for **Python Tutoring**? Sessions can be held remotely using Google+/Skype or in-person if you're in the NYC area. Email jeff@jeffknupp.com if interested.

Sign up for the free jeffknupp.com email newsletter. Sent roughly once a month, it focuses on Python programming, scalable web development, and growing your freelance consultancy. And of course, you'll never be spammed, your privacy is protected, and you can opt out at any time.

* indicates required

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Email Address *

Subscribe

61 Comments jeffknupp.com 1 Login

Recommend 54 Share Sort by Best

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS ?

Name

Carmine • a year ago
Jeff please, where is the second part of this?
45 • Reply • Share ›

Leonardo Chaves Galdino de Mor → Carmine • 2 months ago
You have to call next() on this article
1 • Reply • Share ›

Swing and a miss → Leonardo Chaves Galdino de Mor • 18 days ago
Actually, you'd call next() on Jeff Knupp, as he's the generator of the articles ;)
1 • Reply • Share ›

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Jeff - Mci ➦ Leonardo Chaves Galdino de Mor • 2 months ago

lol

^ | ⌄ • Reply • Share ›

Jenil Shah • 7 months ago

"If next() is never called again, the state recorded during the yield call is (eventually) discarded"

Upto what time will it retain the state? When will the state be discarded if next is not used?

12 ^ | ⌄ • Reply • Share ›

ChadF ➦ Jenil Shah • 7 months ago

A generator function in python is just an object. It, like any object, and any state it holds will be deleted [at some point] after there are no references to it.

If there is always a reference to the generator then its state will never be deleted until the python process exits, even if it is never called again. Normally one would [directly or indirectly] discard the reference to something that won't be used again, leading to its state being "eventually" discarded.

^ | ⌄ • Reply • Share ›

guest8 • 4 years ago

The best article on generators I've seen yet including official documentation! Thanks.

12 ^ | ⌄ • Reply • Share ›

fmars • 2 years ago

Read several articles in the most popular list up to this one. The examples are pretty clear. The only nit maybe the second last example, get the smallest prime number which is greater than some thing. For this case it make no sense to me why do we still need to maintain the state of function, i.g. generator. We can totally use a simple non-generator function to do this because it is totally stateless if my understanding is correct.
For the last example it seems each time when consumer executes, it consume the date passed in the last time not this time. Not sure if it is correct or not. Let's figure it out in next article.
But anyway I totally agree with previous guy's comment, this is the best article, not only on generators, I have seen so far, as a python newbie. I do appreciate

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

4 ^ | ∨ • Reply • Share ›

Johnathan_Irish • 4 years ago

Yield is trying to simulate synchronous multi threaded programming. My view is that if you want to do multi threaded computation, then python is NOT the right language for you. Yield is quite cumbersome and the code can easily become very hard to understand and hard to maintain. Basically, the flow of control is embedded inside the function and the user of the function would need to know the precise internals of the function to use it.

It seems that the flow of control in python is tightly coupled. In multithreaded languages, one can do this much more cleanly by having two threads and either doing it synchronous control between the two threads through message passing or asynchronous by having a shared array between the two threads.

2 ^ | ∨ • Reply • Share ›

Lars Schotte ➦ Johnathan_Irish • 2 years ago

No, coroutines are a form of context switching, not multithreading.
There are problems which are better solved by multithreading, when you really need some concurrent running processes, and that is also the answer how one could do that with python, just start more processes of python and do stuff in parallel.
For other problems, coroutines are better, because they let you do stuff, then continue doing other stuff and then get back to the old stuff, and that is especially important when one stuff is dependent on other stuff to happen, for example one could check if there is a new connection comming in, before just taking some data and sending it to unknown location.

3 ^ | ∨ • Reply • Share ›

user.template ➦ Lars Schotte • 2 years ago

Hmm, a form of context switching you say ... that sounds familiar ...
it reminds me of -- what's that process called? -- the one that happens when a thread exhausts its time slice or yields its remaining time and the operating system has to push the thread's registers and flags onto its kernel stack, then pop the registers, flags, instruction pointer from the kernel stack of the next thread to run before returning to user mode for the next thread to pick right back up where it last yielded, do it's stuff (e.g. check if a new connection is coming in etc. etc.), before yielding and having the OS repeat the whole process..?

Now I remember! It's called context switching! Silly me.

^ | ∨ • Reply • Share ›

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

suiwenfeng ➦ Lars Schotte • 2 years ago

So the "yield" does stuff by co-routines not multithreading?

Reply • Share ›

Charles Pilgrim ➦ suiwenfeng • 2 years ago

As far as I understand there are never two processes running at once. Generators are put on pause at a yield statement

1 Reply • Share ›

Ole ➦ Johnathan_Irish • 2 years ago

I would say 'asynchronous single-threaded' rather than 'asynchronous multi-threaded'. I don't think this was ever meant to simulate multithreaded code.

Reply • Share ›

Lars Schotte ➦ Johnathan_Irish • 2 years ago

Those languages that I know do not have a thread safe arrays. So a shared array between multiple threads could be a serious problem. Message passing is a much better idea, but then, why would you need to do that between threads, when you just could start more processes and do message passing between whole processes.

Reply • Share ›

geek1 • 4 years ago

In your code example:

Example: Fun With Prime Numbers
# or better yet...

def get_primes(input_list):
return (element for element in input_list if is_prime(element))

Doesn't this function already return a generator object?

I think it should be:

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

I think it should be:
def get_primes(input_list):
return [element for element in input_list if is_prime(element)]
2 ^ | ˅ • Reply • Share ›

Ishan Khare ➦ geek1 • 3 years ago
return [generator expression] -> will return a list object
return (generator expression) -> will return a generator obejct
^ | ˅ • Reply • Share ›

Matt Pr • 3 years ago
In the *Moar Power* section...

```
def print_successive_primes(iterations, base=10):
    prime_generator = get_primes(base)
    prime_generator.send(None)
    for power in range(iterations):
        print(prime_generator.send(base ** power))
```

I find this a bit confusing. In particular it seems that in this use case, having a parameter (*base*) for the generator *get_primes* doesn't seem to serve any purpose. The first send call to the generator (*prime_generator.send(None)*) runs the generator to the first yield and produces prime 11 based on the generator parameter *base=10*. However this isn't used, it is just ignored as part of the generator setup. So extra code and wasted cycles.

The reason I bring it up it because as a training example I find the existence of additional unused code confusing and ended up spending some minutes wondering what I was missing, why these bits were important.

Not a python guy, but wouldn't the following be more clear?

```
def get_primes():
    number = yield
    while True:
        if is_prime(number):
```

see more

1 ^ | ˅ • Reply • Share ›

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

sosaysi26 ➦ Matt Pr • 2 years ago

I was wondering this too, I was confused at where that initial argument was used with get_primes(base). Turns out you can put any number in place of base. Thank you for pointing out this alternate solution.

^ | ˅ • Reply • Share ›

Matt Pr ➦ Matt Pr • 3 years ago

Small nitpicking aside, thanks for the educational writeup.

^ | ˅ • Reply • Share ›

msimcoe • a month ago

I agree with Carmine, please do Part 2. This was an excellent article. Finally, I know how and why to correctly use 'yield' and generators. Thank you!

^ | ˅ • Reply • Share ›

Kapil Gupta • 3 months ago

Very nice explanation on generators. Thanks.

^ | ˅ • Reply • Share ›

GermanGrandson • 4 months ago

awesome article, much appreciated!

^ | ˅ • Reply • Share ›

Uri Kanter • 4 months ago

Hey Jeff, just wanted to say thanks , while I've yet to encounter the need to write a Generator Function , this helped me to begin to understand them and how to work with them.

^ | ˅ • Reply • Share ›

Ashok Kumar • 5 months ago

Thanks for the elegant explanation Jeff!!

^ | ˅ • Reply • Share ›

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Relte • 6 months ago

Great post.

Reply • Share ›

Eric • 7 months ago

Hey Jeff. This is great! Thank you.

One thing I am hoping you might explore in your next post is more use cases of when it is appropriate to use generators and contrasting it with situations where a person new to generators might be tempted to use them, but its not appropriate.

Reply • Share ›

Eric ➦ Eric • 7 months ago

I would also love to hear your opinion on what are the use cases for generators. Thanks again!

Reply • Share ›

GGMU • 9 months ago

Good article!
P.S. in your round-up example you don't need to pass 'consumer' to the 'produce' generator, actually you don't need to pass any argument, it won't get used anyway.

Reply • Share ›

Rainald Koch • 9 months ago

Great article but slightly outdated.
"in Python, everything described here as a coroutine is still a generator. Python formally defines the term generator; coroutine is used in discussion but has no formal definition in the language."
There are now coroutines not based on generators (yield in a coroutine would be a syntax error): https://www.python.org/dev/...

Reply • Share ›

Arthur Colombini Gusmão • 10 months ago

Great post! Very instructive

Reply • Share ›

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

vijendra kumar • a year ago

In the first two definitions of get_primes, the second definition returns a generator:

def get_primes(input_list):
return (element for element in input_list if is_prime(element))

The expression (element for element in input_list if is_prime(element)) returns a generator. I think author wanted to use [] brackets instead of () brackets.

• Reply • Share ›

L.Pradip • a year ago

This was really helpful than other articles, Thank you.

• Reply • Share ›

Guest1201 • a year ago

Did not understood this : " When you're using send to "start" a generator, you must send None. " - someone please explain why

• Reply • Share ›

Paul Gradie ➔ Guest1201 • a year ago

Because you will get a TypeError.

`TypeError: can't send non-None value to a just-started generator`

So why do you get a TypeError?

Because the program flow hasn't reached the first yield statement for your send command to actually provide a value to be yielded. In other words, there is nothing yet to send to!

Someone please correct me if this isn't accurate - When you create the generator, if your program flow first requires a send command to the generator, if the generator hasn't reached the point at which it can take in a value, the program doesn't know what to do with that value and raises an error. In fact, on the first step in a generator, the generator expects a None value and will raise an error if anything else is provided.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

The point of providing a generator.send(None) statement at all is to just start the generator and get it to the first yeild statement. Alternatively the example could have used:

`next(prime_generator)`

in place of:

see more

^ | ⌄ • Reply • Share ›

tomyworld • a year ago

Nice explanation on generator & yield!

^ | ⌄ • Reply • Share ›

trishna_g • a year ago

You are great!

^ | ⌄ • Reply • Share ›

Uduse • 2 years ago

Very helpful indeed, thank you!

^ | ⌄ • Reply • Share ›

guest34 • 2 years ago

def get_primes(input_list):
result_list = list()
for element in input_list:
if is_prime(element):
result_list.append()

return result_list

In the function above, shouldn't result_list.append() take argument of element like result_list.append(element)

^ | ⌄ • Reply • Share ›

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Quyen • 2 years ago

Good article

Reply • Share ›

guest9 • 2 years ago

Sir the problem can be solved using Static variable.???

Reply • Share ›

Deepak Singh Raghuvanshi • 2 years ago

Well explained. _/_

Reply • Share ›

姜潤弦 • 2 years ago

HAHA I actually was doing project euler problem 10 (':

Reply • Share ›

renjithsraj • 2 years ago

i have one doubt, where we can use the generator instead of normal iteration m please tell me in simple war

Reply • Share ›

Anony Mous ➦ renjithsraj • 2 years ago

Normal iteration (they both use the same iterator expression, just generate values differently), assumes you already have the list, dict, collection of values to iterate through. Yielding doesn't require you to have a set of items already built. Which could mean much more flexibility with generators.

For example I use a generator in the context of streaming news items for a websocket app. I yield items from a database should new ones be present, otherwise, I yield None, which tells my application to tell my web app that there are no new values to see. This way, the clients connected to my web app can continuously get an updated feed when the values are present, but the iteration will always continue providing something.

Otherwise, I would have to grab ALL the available items from the database and send them all in one huge chunk which isn't what I

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

need. It's what alot of web apps do when they only render a certain portion of the page and when you scroll down it fetches more data to render.

Reply • Share ›

Robert Tavoularis • 2 years ago

I just finished the generator section in "Learning Python" and was completely lost as to when I would use them, this cleared it up. Thank you!!

Reply • Share ›

Kacper Goc • 2 years ago

Actually, python provides "magical_infinite_range(start)". It's itertools.count(start, step). Just for the reference.

Reply • Share ›

Julio • 2 years ago

Hi, everybody out there, I just came to this good article, but I am still stuck in this: according with the theory a generators, as I understand, are some kind of remember-last-value-function, almost, if not, equally to a turn attend number. When an other person arrived, then the machine will give him my number + 1 turn. That said, all the examples I have seen are with a parameter start number, so the loop will go until this number., but I have searched many web sites, reviewed many code, unfortunately without found something like get_primes() that is with no parameter that put an limit to the function, the limit I will defined it, by putting it, in some sort of outer loop and when needed just call get_primes.next().
I appreciate any collaboration

Reply • Share ›

DiggitBuddy • 2 years ago

This was a very clear and detailed description of generators. Thanks for sharing!

Reply • Share ›

vivek akupatni • 3 years ago

Very insightful post

Reply • Share ›

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Load more comments

ALSO ON **JEFFKNUPP.COM**

How Python Makes Working With Data More Difficult in the Long Run

16 comments • 2 years ago

Jamie Strauss — `classes` are a mutable runtime construct.`namedtuples` is a glorified dict.Don't pretend you have any idea what the fuck you are talking about.

Improve Your Python: Python Classes and Object Oriented Programming

10 comments • a year ago

Pablo Arias — Hello Jeff, very nice article! I've been programming for six years now (mostly Java and C++), but only until recently started learning python.Coming from strongly typed languages, …

A Common Misunderstanding About Python Generators

2 comments • 21 days ago

vikrant — 👍👍

Python, sandman2, and Open Data

1 comment • 2 years ago

Michael Aye — FYI, your sandman2 link points to sandman.

Subscribe Add Disqus to your site Disqus' Privacy Policy DISQUS

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Copyright © 2014 - Jeff Knupp- Powered by Blug

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD