Jeff Knupp

PYTHON PROGRAMMER

Everything I know about Python...

Improve Your Python: 'yield' and Generators Explained

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

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.

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

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

```
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

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
```

```
    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()`.

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
```

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>
```

```
    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)

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

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)
```

```
    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)
```

```
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`

- 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

« And Now for Something Completely Different...

Email Address *

Subscribe