What does the "yield" keyword do in Python?

Asked 14 years, 1 month ago Modified yesterday Viewed 3.1m times

12391

Want to improve this post? Provide detailed answers to this question, including citations and an explanation of why your answer is correct. Answers without enough detail may be edited or deleted.

What is the use of the `yield` keyword in Python? What does it do?

For example, I'm trying to understand this code[1]:

```
def _get_child_candidates(self, distance, min_dist, max_dist):
    if self._leftchild and distance - max_dist < self._median:
        yield self._leftchild
    if self._rightchild and distance + max_dist >= self._median:
        yield self._rightchild
```

And this is the caller:

```
result, candidates = [], [self]
while candidates:
    node = candidates.pop()
    distance = node._get_dist(obj)
    if distance <= max_dist and distance >= min_dist:
        result.extend(node._values)
    candidates.extend(node._get_child_candidates(distance, min_dist, max_dist))
return result
```

What happens when the method `_get_child_candidates` is called? Is a list returned? A single element? Is it called again? When will subsequent calls stop?

1. This piece of code was written by Jochen Schulz (jrschulz), who made a great Python library for metric spaces. This is the link to the complete source: Module mspace.

python generator yield

Share Edit Follow

edited yesterday

Martijn Pieters ♦
998k 280 3925 3265

asked Oct 23, 2008 at 22:21

Alex. S.
140k 19 53 61

51 Answers

Sorted by: Highest score (default)

1 2 Next

17415

To understand what `yield` does, you must understand what *generators* are. And before you can understand generators, you must understand *iterables*.

+600

Iterables

When you create a list, you can read its items one by one. Reading its items one by one is called iteration:

```
>>> mylist = [1, 2, 3]
>>> for i in mylist:
...    print(i)
1
2
3
```

`mylist` is an *iterable*. When you use a list comprehension, you create a list, and so an iterable:

```
>>> mylist = [x*x for x in range(3)]
>>> for i in mylist:
...    print(i)
0
1
4
```

Everything you can use "`for... in...`" on is an iterable; `lists`, `strings`, files...

These iterables are handy because you can read them as much as you wish, but you store all the values in memory and this is not always what you want when you have a lot of values.

Generators

Generators are iterators, a kind of iterable **you can only iterate over once**. Generators do not store all the values in memory, **they generate the values on the fly**:

```
>>> mygenerator = (x*x for x in range(3))
>>> for i in mygenerator:
...    print(i)
0
1
4
```

It is just the same except you used `()` instead of `[]`. BUT, you **cannot** perform `for i in mygenerator` a second time since generators can only be used once: they calculate 0, then forget about it and calculate 1, and end calculating 4, one by one.

Yield

`yield` is a keyword that is used like `return`, except the function will return a generator.

```
>>> def create_generator():
...    mylist = range(3)
...    for i in mylist:
...        yield i*i
...
>>> mygenerator = create_generator() # create a generator
>>> print(mygenerator) # mygenerator is an object!
<generator object create_generator at 0xb7555c34>
>>> for i in mygenerator:
...     print(i)
0
```

```
1
4
```

Here it's a useless example, but it's handy when you know your function will return a huge set of values that you will only need to read once.

To master `yield` , you must understand that **when you call the function, the code you have written in the function body does not run.** The function only returns the generator object, this is a bit tricky.

Then, your code will continue from where it left off each time `for` uses the generator.

Now the hard part:

The first time the `for` calls the generator object created from your function, it will run the code in your function from the beginning until it hits `yield` , then it'll return the first value of the loop. Then, each subsequent call will run another iteration of the loop you have written in the function and return the next value. This will continue until the generator is considered empty, which happens when the function runs without hitting `yield` . That can be because the loop has come to an end, or because you no longer satisfy an `"if/else"` .

Your code explained

Generator:

```
# Here you create the method of the node object that will return the generator
def _get_child_candidates(self, distance, min_dist, max_dist):

    # Here is the code that will be called each time you use the generator
object:

    # If there is still a child of the node object on its left
    # AND if the distance is ok, return the next child
    if self._leftchild and distance - max_dist < self._median:
        yield self._leftchild

    # If there is still a child of the node object on its right
    # AND if the distance is ok, return the next child
    if self._rightchild and distance + max_dist >= self._median:
        yield self._rightchild

    # If the function arrives here, the generator will be considered empty
    # there are no more than two values: the left and the right children
```

Caller:

```
# Create an empty list and a list with the current object reference
result, candidates = list(), [self]

# Loop on candidates (they contain only one element at the beginning)
while candidates:

    # Get the last candidate and remove it from the list
    node = candidates.pop()

    # Get the distance between obj and the candidate
    distance = node._get_dist(obj)

    # If the distance is ok, then you can fill in the result
    if distance <= max_dist and distance >= min_dist:
        result.extend(node._values)

    # Add the children of the candidate to the candidate's list
    # so the loop will keep running until it has looked
    # at all the children of the children of the children, etc. of the
```

```
candidate
    candidates.extend(node._get_child_candidates(distance, min_dist, max_dist))

return result
```

This code contains several smart parts:

- The loop iterates on a list, but the list expands while the loop is being iterated. It's a concise way to go through all these nested data even if it's a bit dangerous since you can end up with an infinite loop. In this case, `candidates.extend(node._get_child_candidates(distance, min_dist, max_dist))` exhausts all the values of the generator, but `while` keeps creating new generator objects which will produce different values from the previous ones since it's not applied on the same node.
- The `extend()` method is a list object method that expects an iterable and adds its values to the list.

Usually, we pass a list to it:

```
>>> a = [1, 2]
>>> b = [3, 4]
>>> a.extend(b)
>>> print(a)
[1, 2, 3, 4]
```

But in your code, it gets a generator, which is good because:

1. You don't need to read the values twice.
2. You may have a lot of children and you don't want them all stored in memory.

And it works because Python does not care if the argument of a method is a list or not. Python expects iterables so it will work with strings, lists, tuples, and generators! This is called duck typing and is one of the reasons why Python is so cool. But this is another story, for another question...

You can stop here, or read a little bit to see an advanced use of a generator:

Controlling a generator exhaustion

```
>>> class Bank(): # Let's create a bank, building ATMs
...    crisis = False
...    def create_atm(self):
...        while not self.crisis:
...            yield "$100"
>>> hsbc = Bank() # When everything's ok the ATM gives you as much as you want
>>> corner_street_atm = hsbc.create_atm()
>>> print(corner_street_atm.next())
$100
>>> print(corner_street_atm.next())
$100
>>> print([corner_street_atm.next() for cash in range(5)])
['$100', '$100', '$100', '$100', '$100']
>>> hsbc.crisis = True # Crisis is coming, no more money!
>>> print(corner_street_atm.next())
<type 'exceptions.StopIteration'>
>>> wall_street_atm = hsbc.create_atm() # It's even true for new ATMs
>>> print(wall_street_atm.next())
<type 'exceptions.StopIteration'>
>>> hsbc.crisis = False # The trouble is, even post-crisis the ATM remains
empty
>>> print(corner_street_atm.next())
<type 'exceptions.StopIteration'>
>>> brand_new_atm = hsbc.create_atm() # Build a new one to get back in business
>>> for cash in brand_new_atm:
...    print cash
```

```
$100
$100
$100
$100
$100
$100
$100
$100
$100
...
```

Note: For Python 3, use `print(corner_street_atm.__next__())` or `print(next(corner_street_atm))`

It can be useful for various things like controlling access to a resource.

Itertools, your best friend

The itertools module contains special functions to manipulate iterables. Ever wish to duplicate a generator? Chain two generators? Group values in a nested list with a one-liner? `Map / Zip` without creating another list?

Then just `import itertools`.

An example? Let's see the possible orders of arrival for a four-horse race:

```
>>> horses = [1, 2, 3, 4]
>>> races = itertools.permutations(horses)
>>> print(races)
<itertools.permutations object at 0xb754f1dc>
>>> print(list(itertools.permutations(horses)))
[(1, 2, 3, 4),
 (1, 2, 4, 3),
 (1, 3, 2, 4),
 (1, 3, 4, 2),
 (1, 4, 2, 3),
 (1, 4, 3, 2),
 (2, 1, 3, 4),
 (2, 1, 4, 3),
 (2, 3, 1, 4),
 (2, 3, 4, 1),
 (2, 4, 1, 3),
 (2, 4, 3, 1),
 (3, 1, 2, 4),
 (3, 1, 4, 2),
 (3, 2, 1, 4),
 (3, 2, 4, 1),
 (3, 4, 1, 2),
 (3, 4, 2, 1),
 (4, 1, 2, 3),
 (4, 1, 3, 2),
 (4, 2, 1, 3),
 (4, 2, 3, 1),
 (4, 3, 1, 2),
 (4, 3, 2, 1)]
```

Understanding the inner mechanisms of iteration

Iteration is a process implying iterables (implementing the `__iter__()` method) and iterators (implementing the `__next__()` method). Iterables are any objects you can get an iterator from. Iterators are objects that let you iterate on iterables.

There is more about it in this article about [how `for` loops work].

Share Edit Follow

edited Oct 1 at 18:08
Saket Thakur
15 3

answered Oct 23, 2008 at 22:48
e-satis
565k 110 294 328

675 `yield` is not as magical this answer suggests. When you call a function that contains a `yield` statement anywhere, you get a generator object, but no code runs. Then each time you extract an object from the generator, Python executes code in the function until it comes to a `yield` statement, then pauses and delivers the object. When you extract another object, Python resumes just after the `yield` and continues until it reaches another `yield` (often the same one, but one iteration later). This continues until the function runs off the end, at which point the generator is deemed exhausted. – Matthias Fripp May 23, 2017 at 21:41

74 "These iterables are handy... but you store all the values in memory and this is not always what you want", is either wrong or confusing. An iterable returns an iterator upon calling the iter() on the iterable, and an iterator doesn't always have to store its values in memory, depending on the implementation of the **iter** method, it can also generate values in the sequence on demand. – picmate 涅 Feb 15, 2018 at 19:21

25 It would be nice to add to this **great** answer why *It is just the same except you used* `()` *instead of* `[]`, specifically what `()` is (there may be confusion with a tuple). – WoJ May 7, 2020 at 10:12

40 @MatthiasFripp "This continues until the function runs off the end" -- or it encounters a `return` statement. (`return` is permitted in a function containing `yield`, provided that it does not specify a return value.) – alani Jun 6, 2020 at 6:03

8 The yield statement suspends function's execution and sends a value back to the caller, but retains enough state to enable function to resume where it is left off. When resumed, the function continues execution immediately after the last yield run. This allows its code to produce a series of values over time, rather than computing them at once and sending them back like a list. – Jacob Ward Dec 3, 2020 at 1:23

2428

Shortcut to understanding `yield`

When you see a function with `yield` statements, apply this easy trick to understand what will happen:

1. Insert a line `result = []` at the start of the function.
2. Replace each `yield expr` with `result.append(expr)`.
3. Insert a line `return result` at the bottom of the function.
4. Yay - no more `yield` statements! Read and figure out the code.
5. Compare function to the original definition.

This trick may give you an idea of the logic behind the function, but what actually happens with `yield` is significantly different than what happens in the list-based approach. In many cases, the yield approach will be a lot more memory efficient and faster too. In other cases, this trick will get you stuck in an infinite loop, even though the original function works just fine. Read on to learn more...

Don't confuse your Iterables, Iterators, and Generators

First, the **iterator protocol** - when you write

```
for x in mylist:
    ...loop body...
```

Python performs the following two steps:

1. Gets an iterator for `mylist`:

   Call `iter(mylist)` -> this returns an object with a `next()` method (or `__next__()` in Python 3).

   [This is the step most people forget to tell you about]

2. Uses the iterator to loop over items:

Keep calling the `next()` method on the iterator returned from step 1. The return value from `next()` is assigned to `x` and the loop body is executed. If an exception `StopIteration` is raised from within `next()`, it means there are no more values in the iterator and the loop is exited.

The truth is Python performs the above two steps anytime it wants to *loop over* the contents of an object - so it could be a for loop, but it could also be code like `otherlist.extend(mylist)` (where `otherlist` is a Python list).

Here `mylist` is an *iterable* because it implements the iterator protocol. In a user-defined class, you can implement the `__iter__()` method to make instances of your class iterable. This method should return an *iterator*. An iterator is an object with a `next()` method. It is possible to implement both `__iter__()` and `next()` on the same class, and have `__iter__()` return `self`. This will work for simple cases, but not when you want two iterators looping over the same object at the same time.

So that's the iterator protocol, many objects implement this protocol:

1. Built-in lists, dictionaries, tuples, sets, and files.
2. User-defined classes that implement `__iter__()`.
3. Generators.

Note that a `for` loop doesn't know what kind of object it's dealing with - it just follows the iterator protocol, and is happy to get item after item as it calls `next()`. Built-in lists return their items one by one, dictionaries return the *keys* one by one, files return the *lines* one by one, etc. And generators return... well that's where `yield` comes in:

```
def f123():
    yield 1
    yield 2
    yield 3

for item in f123():
    print item
```

Instead of `yield` statements, if you had three `return` statements in `f123()` only the first would get executed, and the function would exit. But `f123()` is no ordinary function. When `f123()` is called, it *does not* return any of the values in the yield statements! It returns a generator object. Also, the function does not really exit - it goes into a suspended state. When the `for` loop tries to loop over the generator object, the function resumes from its suspended state at the very next line after the `yield` it previously returned from, executes the next line of code, in this case, a `yield` statement, and returns that as the next item. This happens until the function exits, at which point the generator raises `StopIteration`, and the loop exits.

So the generator object is sort of like an adapter - at one end it exhibits the iterator protocol, by exposing `__iter__()` and `next()` methods to keep the `for` loop happy. At the other end, however, it runs the function just enough to get the next value out of it, and puts it back in suspended mode.

Why Use Generators?

Usually, you can write code that doesn't use generators but implements the same logic. One option is to use the temporary list 'trick' I mentioned before. That will not work in all cases, for e.g. if you have infinite loops, or it may make inefficient use of memory when you have a really long list. The other approach is to implement a new iterable class SomethingIter that keeps the state in instance members and performs the next logical step in its `next()` (or `__next__()` in Python 3) method. Depending on the logic, the code inside the `next()` method may end up looking very complex and prone to bugs. Here generators provide a clean and easy solution.

39 *"When you see a function with yield statements, apply this easy trick to understand what will happen"* Doesn't this completely ignore the fact that you can `send` into a generator, which is a huge part of the point of generators? – DanielSank Jun 17, 2017 at 22:41

16 "it could be a for loop, but it could also be code like `otherlist.extend(mylist)` " -> This is incorrect. `extend()` modifies the list in-place and does not return an iterable. Trying to loop over `otherlist.extend(mylist)` will fail with a `TypeError` because `extend()` implicitly returns `None` , and you can't loop over `None` . – Pedro Sep 14, 2017 at 14:48

13 @pedro You have misunderstood that sentence. It means that python performs the two mentioned steps on `mylist` (not on `otherlist`) when executing `otherlist.extend(mylist)` . – today Dec 26, 2017 at 18:53

720

Think of it this way:

An iterator is just a fancy sounding term for an object that has a `next()` method. So a yield-ed function ends up being something like this:

Original version:

```
def some_function():
    for i in xrange(4):
        yield i

for i in some_function():
    print i
```

This is basically what the Python interpreter does with the above code:

```
class it:
    def __init__(self):
        # Start at -1 so that we get 0 when we add 1 below.
        self.count = -1

    # The __iter__ method will be called once by the 'for' loop.
    # The rest of the magic happens on the object returned by this method.
    # In this case it is the object itself.
    def __iter__(self):
        return self

    # The next method will be called repeatedly by the 'for' loop
    # until it raises StopIteration.
    def next(self):
        self.count += 1
        if self.count < 4:
            return self.count
        else:
            # A StopIteration exception is raised
            # to signal that the iterator is done.
            # This is caught implicitly by the 'for' loop.
            raise StopIteration

def some_func():
    return it()

for i in some_func():
    print i
```

For more insight as to what's happening behind the scenes, the `for` loop can be rewritten to this:

```
iterator = some_func()
try:
    while 1:
        print iterator.next()
except StopIteration:
    pass
```

Does that make more sense or just confuse you more? :)

I should note that this *is* an oversimplification for illustrative purposes. :)

Share Edit Follow

edited May 7, 2019 at 13:28
Georgy
11.2k 7 62 70

answered Oct 23, 2008 at 22:28
Jason Baker
187k 134 370 510

1 `__getitem__` could be defined instead of `__iter__`. For example: `class it: pass; it.__getitem__ = lambda self, i: i*10 if i < 10 else [][0]; for i in it(): print(i)`, It will print: 0, 10, 20, ..., 90 – jfs Oct 25, 2008 at 2:03

33 I tried this example in Python 3.6 and if I create `iterator = some_function()`, the variable `iterator` does not have a function called `next()` anymore, but only a `__next__()` function. Thought I'd mention it. – Peter May 6, 2017 at 14:37

611

The `yield` keyword is reduced to two simple facts:

1. If the compiler detects the `yield` keyword *anywhere* inside a function, that function no longer returns via the `return` statement. ***Instead***, it **immediately** returns a **lazy "pending list" object** called a generator
2. A generator is iterable. What is an *iterable*? It's anything like a `list` or `set` or `range` or dict-view, with a *built-in protocol for visiting each element in a certain order*.

In a nutshell: Most commonly, **a generator is a lazy, incrementally-pending list**, and **`yield` statements allow you to use function notation to program the list values** the generator should incrementally spit out. **Furthermore, advanced usage lets you use generators as coroutines (see below).**

```
generator = myYieldingFunction(...)  # basically a list (but lazy)
x = list(generator)  # evaluate every element into a list

   generator
       v
[x[0], ..., ???]

         generator
             v
[x[0], x[1], ..., ???]

               generator
                   v
[x[0], x[1], x[2], ..., ???]

                       StopIteration exception
[x[0], x[1], x[2]]     done
```

Basically, whenever the `yield` statement is encountered, the function pauses and saves its state, then emits "the next return value in the 'list'" according to the python iterator protocol (to some syntactic construct like a for-loop that repeatedly calls `next()` and catches a `StopIteration` exception, etc.). You might have encountered generators with generator expressions; generator functions are more powerful because you can pass arguments back into the paused generator function, using them to implement coroutines. More on that later.

Basic Example ('list')

Let's define a function `makeRange` that's just like Python's `range`. Calling `makeRange(n)` RETURNS A GENERATOR:

```
def makeRange(n):
    # return 0,1,2,...,n-1
    i = 0
```

```
    while i < n:
        yield i
        i += 1

>>> makeRange(5)
<generator object makeRange at 0x19e4aa0>
```

To force the generator to immediately return its pending values, you can pass it into `list()` (just like you could any iterable):

```
>>> list(makeRange(5))
[0, 1, 2, 3, 4]
```

Comparing example to "just returning a list"

The above example can be thought of as merely creating a list which you append to and return:

```
# return a list                  #  # return a generator
def makeRange(n):                #  def makeRange(n):
    """return [0,1,2,...,n-1]""" #      """return 0,1,2,...,n-1"""
    TO_RETURN = []               #
    i = 0                        #      i = 0
    while i < n:                 #      while i < n:
        TO_RETURN += [i]         #          yield i
        i += 1                   #          i += 1
    return TO_RETURN             #

>>> makeRange(5)
[0, 1, 2, 3, 4]
```

There is one major difference, though; see the last section.

How you might use generators

An iterable is the last part of a list comprehension, and all generators are iterable, so they're often used like so:

```
#                   < ITERABLE >
>>> [x+10 for x in makeRange(5)]
[10, 11, 12, 13, 14]
```

To get a better feel for generators, you can play around with the `itertools` module (be sure to use `chain.from_iterable` rather than `chain` when warranted). For example, you might even use generators to implement infinitely-long lazy lists like `itertools.count()`. You could implement your own `def enumerate(iterable): zip(count(), iterable)`, or alternatively do so with the `yield` keyword in a while-loop.

Please note: generators can actually be used for many more things, such as implementing coroutines or non-deterministic programming or other elegant things. However, the "lazy lists" viewpoint I present here is the most common use you will find.

Behind the scenes

This is how the "Python iteration protocol" works. That is, what is going on when you do `list(makeRange(5))`. This is what I describe earlier as a "lazy, incremental list".

```
>>> x=iter(range(5))
>>> next(x)  # calls x.__next__(); x.next() is deprecated
0
```

```
>>> next(x)
1
>>> next(x)
2
>>> next(x)
3
>>> next(x)
4
>>> next(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

The built-in function `next()` just calls the objects `.__next__()` function, which is a part of the "iteration protocol" and is found on all iterators. You can manually use the `next()` function (and other parts of the iteration protocol) to implement fancy things, usually at the expense of readability, so try to avoid doing that...

Coroutines

Coroutine example:

```
def interactiveProcedure():
    userResponse = yield makeQuestionWebpage()
    print('user response:', userResponse)
    yield 'success'

coroutine = interactiveProcedure()
webFormData = next(coroutine)  # same as .send(None)
userResponse = serveWebForm(webFormData)

# ...at some point later on web form submit...

successStatus = coroutine.send(userResponse)
```

A coroutine (generators which generally accept input via the `yield` keyword e.g. `nextInput = yield nextOutput`, as a form of two-way communication) is basically a computation which is allowed to pause itself and request input (e.g. to what it should do next). When the coroutine pauses itself (when the running coroutine eventually hits a `yield` keyword), the computation is paused and control is inverted (yielded) back to the 'calling' function (the frame which requested the `next` value of the computation). The paused generator/coroutine remains paused until another invoking function (possibly a different function/context) requests the next value to unpause it (usually passing input data to direct the paused logic interior to the coroutine's code).

You can think of python coroutines as lazy incrementally-pending lists, where the next element doesn't just depend on the previous computation, but also on input you may opt to inject during the generation process.

Minutiae

Normally, most people would not care about the following distinctions and probably want to stop reading here.

In Python-speak, an *iterable* is any object which "understands the concept of a for-loop" like a list `[1,2,3]`, and an *iterator* is a specific instance of the requested for-loop like `[1,2,3].__iter__()`. A *generator* is exactly the same as any iterator, except for the way it was written (with function syntax).

When you request an iterator from a list, it creates a new iterator. However, when you request an iterator from an iterator (which you would rarely do), it just gives you a copy of itself.

Thus, in the unlikely event that you are failing to do something like this...

```
> x = myRange(5)
> list(x)
[0, 1, 2, 3, 4]
> list(x)
[]
```

... then remember that a generator is an *iterator*; that is, it is one-time-use. If you want to reuse it, you should call `myRange(...)` again. If you need to use the result twice, convert the result to a list and store it in a variable `x = list(myRange(5))`. Those who absolutely need to clone a generator (for example, who are doing terrifyingly hackish metaprogramming) can use `itertools.tee` (still works in Python 3) if absolutely necessary, since the copyable iterator Python PEP standards proposal has been deferred.

Share Edit Follow

edited Oct 1 at 12:07

answered Jun 19, 2011 at 6:33

ninjagecko
85.9k 24 134 143

528

What does the `yield` keyword do in Python?

Answer Outline/Summary

- A function with `yield`, when called, **returns a Generator.**
- Generators are iterators because they implement the **iterator protocol**, so you can iterate over them.
- A generator can also be **sent information**, making it conceptually a **coroutine**.
- In Python 3, you can **delegate** from one generator to another in both directions with **`yield from`**.
- (Appendix critiques a couple of answers, including the top one, and discusses the use of `return` in a generator.)

Generators:

`yield` is only legal inside of a function definition, and **the inclusion of `yield` in a function definition makes it return a generator.**

The idea for generators comes from other languages (see footnote 1) with varying implementations. In Python's Generators, the execution of the code is frozen at the point of the yield. When the generator is called (methods are discussed below) execution resumes and then freezes at the next yield.

`yield` provides an easy way of implementing the iterator protocol, defined by the following two methods: `__iter__` and `__next__`. Both of those methods make an object an iterator that you could type-check with the `Iterator` Abstract Base Class from the `collections` module.

```
def func():
    yield 'I am'
    yield 'a generator!'
```

Let's do some introspection:

```
>>> type(func)                 # A function with yield is still a function
<type 'function'>
>>> gen = func()
>>> type(gen)                  # but it returns a generator
<type 'generator'>
>>> hasattr(gen, '__iter__')   # that's an iterable
```

```
True
>>> hasattr(gen, '__next__')   # and with .__next__
True                           # implements the iterator protocol.
```

The generator type is a sub-type of iterator:

```
from types import GeneratorType
from collections.abc import Iterator

>>> issubclass(GeneratorType, Iterator)
True
```

And if necessary, we can type-check like this:

```
>>> isinstance(gen, GeneratorType)
True
>>> isinstance(gen, Iterator)
True
```

A feature of an `Iterator` is that once exhausted, you can't reuse or reset it:

```
>>> list(gen)
['I am', 'a generator!']
>>> list(gen)
[]
```

You'll have to make another if you want to use its functionality again (see footnote 2):

```
>>> list(func())
['I am', 'a generator!']
```

One can yield data programmatically, for example:

```
def func(an_iterable):
    for item in an_iterable:
        yield item
```

The above simple generator is also equivalent to the below - as of Python 3.3 you can use `yield from`:

```
def func(an_iterable):
    yield from an_iterable
```

However, `yield from` also allows for delegation to subgenerators, which will be explained in the following section on cooperative delegation with sub-coroutines.

Coroutines:

`yield` forms an expression that allows data to be sent into the generator (see footnote 3)

Here is an example, take note of the `received` variable, which will point to the data that is sent to the generator:

```python
def bank_account(deposited, interest_rate):
    while True:
        calculated_interest = interest_rate * deposited
        received = yield calculated_interest
        if received:
            deposited += received
```

```python
>>> my_account = bank_account(1000, .05)
```

First, we must queue up the generator with the builtin function, `next`. It will call the appropriate `next` or `__next__` method, depending on the version of Python you are using:

```python
>>> first_year_interest = next(my_account)
>>> first_year_interest
50.0
```

And now we can send data into the generator. (Sending `None` is the same as calling `next`.) :

```python
>>> next_year_interest = my_account.send(first_year_interest + 1000)
>>> next_year_interest
102.5
```

Cooperative Delegation to Sub-Coroutine with `yield from`

Now, recall that `yield from` is available in Python 3. This allows us to delegate coroutines to a subcoroutine:

```python
def money_manager(expected_rate):
    # must receive deposited value from .send():
    under_management = yield                   # yield None to start.
    while True:
        try:
            additional_investment = yield expected_rate * under_management
            if additional_investment:
                under_management += additional_investment
        except GeneratorExit:
            '''TODO: write function to send unclaimed funds to state'''
            raise
        finally:
            '''TODO: write function to mail tax info to client'''


def investment_account(deposited, manager):
    '''very simple model of an investment account that delegates to a
manager'''
    # must queue up manager:
    next(manager)      # <- same as manager.send(None)
    # This is where we send the initial deposit to the manager:
    manager.send(deposited)
    try:
        yield from manager
    except GeneratorExit:
        return manager.close()  # delegate?
```

And now we can delegate functionality to a sub-generator and it can be used by a generator just as above:

```python
my_manager = money_manager(.06)
my_account = investment_account(1000, my_manager)
```

```
first_year_return = next(my_account) # -> 60.0
```

Now simulate adding another 1,000 to the account plus the return on the account (60.0):

```
next_year_return = my_account.send(first_year_return + 1000)
next_year_return # 123.6
```

You can read more about the precise semantics of `yield from` in [PEP 380.]

Other Methods: close and throw

The `close` method raises `GeneratorExit` at the point the function execution was frozen. This will also be called by `__del__` so you can put any cleanup code where you handle the `GeneratorExit` :

```
my_account.close()
```

You can also throw an exception which can be handled in the generator or propagated back to the user:

```
import sys
try:
    raise ValueError
except:
    my_manager.throw(*sys.exc_info())
```

Raises:

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
  File "<stdin>", line 6, in money_manager
  File "<stdin>", line 2, in <module>
ValueError
```

Conclusion

I believe I have covered all aspects of the following question:

> **What does the `yield` keyword do in Python?**

It turns out that `yield` does a lot. I'm sure I could add even more thorough examples to this. If you want more or have some constructive criticism, let me know by commenting below.

Appendix:

Critique of the Top/Accepted Answer**

- It is confused on what makes an **iterable**, just using a list as an example. See my references above, but in summary: an **iterable** has an `__iter__` method returning an **iterator**. An **iterator** additionally provides a `.__next__` method, which is implicitly called by `for` loops until it raises `StopIteration` , and once it does raise `StopIteration` , it will continue to do so.

- It then uses a generator expression to describe what a generator is. Since a generator expression is simply a convenient way to create an **iterator**, it only confuses the matter, and we still have not yet gotten to the `yield` part.
- In **Controlling a generator exhaustion** he calls the `.next` method (which only works in Python 2), when instead he should use the builtin function, `next`. Calling `next(obj)` would be an appropriate layer of indirection, because his code does not work in Python 3.
- Itertools? This was not relevant to what `yield` does at all.
- No discussion of the methods that `yield` provides along with the new functionality `yield from` in Python 3.

The top/accepted answer is a very incomplete answer.

Critique of answer suggesting `yield` in a generator expression or comprehension.

The grammar currently allows any expression in a list comprehension.

```
expr_stmt: testlist_star_expr (annassign | augassign (yield_expr|testlist) |
                     ('=' (yield_expr|testlist_star_expr))*)
...
yield_expr: 'yield' [yield_arg]
yield_arg: 'from' test | testlist
```

Since yield is an expression, it has been touted by some as interesting to use it in comprehensions or generator expression - in spite of citing no particularly good use-case.

The CPython core developers are [discussing deprecating its allowance](). Here's a relevant post from the mailing list:

> On 30 January 2017 at 19:05, Brett Cannon wrote:
>
>> On Sun, 29 Jan 2017 at 16:39 Craig Rodrigues wrote:
>>
>>> I'm OK with either approach. Leaving things the way they are in Python 3 is no good, IMHO.
>>
>> My vote is it be a SyntaxError since you're not getting what you expect from the syntax.
>
> I'd agree that's a sensible place for us to end up, as any code relying on the current behaviour is really too clever to be maintainable.
>
> In terms of getting there, we'll likely want:
>
> - SyntaxWarning or DeprecationWarning in 3.7
> - Py3k warning in 2.7.x
> - SyntaxError in 3.8
>
> Cheers, Nick.
>
> -- Nick Coghlan | ncoghlan at gmail.com | Brisbane, Australia

Further, there is an [outstanding issue (10544)]() which seems to be pointing in the direction of this *never* being a good idea (PyPy, a Python implementation written in Python, is already raising syntax warnings.)

Bottom line, until the developers of CPython tell us otherwise: **Don't put `yield` in a generator expression or comprehension.**

The `return` statement in a generator

In [Python 3](https://docs.python.org/3/reference/simple_stmts.html#the-return-statement):

> In a generator function, the `return` statement indicates that the generator is done and will cause `StopIteration` to be raised. The returned value (if any) is used as an argument to construct `StopIteration` and becomes the `StopIteration.value` attribute.

Historical note, in [Python 2](https://docs.python.org/2/reference/simple_stmts.html#the-return-statement): "In a generator function, the `return` statement is not allowed to include an `expression_list`. In that context, a bare `return` indicates that the generator is done and will cause `StopIteration` to be raised." An `expression_list` is basically any number of expressions separated by commas - essentially, in Python 2, you can stop the generator with `return`, but you can't return a value.

Footnotes

1. The languages CLU, Sather, and Icon were referenced in the proposal to introduce the concept of generators to Python. The general idea is that a function can maintain internal state and yield intermediate data points on demand by the user. This promised to be [superior in performance to other approaches, including Python threading](https://www.python.org/dev/peps/pep-0255/), which isn't even available on some systems.
2. This means, for example, that `range` objects aren't `Iterator` s, even though they are iterable, because they can be reused. Like lists, their `__iter__` methods return iterator objects.
3. `yield` was originally introduced as a statement, meaning that it could only appear at the beginning of a line in a code block. Now `yield` creates a yield expression. https://docs.python.org/2/reference/simple_stmts.html#grammar-token-yield_stmt This change was [proposed](https://www.python.org/dev/peps/pep-0342/) to allow a user to send data into the generator just as one might receive it. To send data, one must be able to assign it to something, and for that, a statement just won't work.

Share Edit Follow

edited Oct 26 at 8:26

answered Jun 25, 2015 at 6:11

Russia Must Remove Putin ♦

356k 85 395 329

434

`yield` is just like `return` - it returns whatever you tell it to (as a generator). The difference is that the next time you call the generator, execution starts from the last call to the `yield` statement. Unlike return, **the stack frame is not cleaned up when a yield occurs, however control is transferred back to the caller, so its state will resume the next time the function is called.**

In the case of your code, the function `get_child_candidates` is acting like an iterator so that when you extend your list, it adds one element at a time to the new list.

`list.extend` calls an iterator until it's exhausted. In the case of the code sample you posted, it would be much clearer to just return a tuple and append that to the list.

Share Edit Follow

edited Jan 24, 2019 at 9:39

Fang

2,091 4 22 40

answered Oct 23, 2008 at 22:24

Douglas Mayle

20.5k 8 42 57

124 This is close, but not correct. Every time you call a function with a yield statement in it, it returns a brand new generator object. It's only when you call that generator's .next() method that execution resumes after the last yield. – kurosch Oct 24, 2008 at 18:11

306

There's one extra thing to mention: a function that yields doesn't actually have to terminate. I've written code like this:

```
def fib():
    last, cur = 0, 1
```

```
    while True:
        yield cur
        last, cur = cur, last + cur
```

Then I can use it in other code like this:

```
for f in fib():
    if some_condition: break
    coolfuncs(f);
```

It really helps simplify some problems, and makes some things easier to work with.

Share Edit Follow

edited Apr 21, 2013 at 15:42

answered Oct 24, 2008 at 8:44
Claudiu
220k 161 471 676

296

For those who prefer a minimal working example, meditate on this interactive Python session:

```
>>> def f():
...   yield 1
...   yield 2
...   yield 3
...
>>> g = f()
>>> for i in g:
...   print(i)
...
1
2
3
>>> for i in g:
...   print(i)
...
>>> # Note that this time nothing was printed
```

Share Edit Follow

edited Feb 2, 2020 at 22:09
Oren
4,097 4 33 61

answered Jan 18, 2013 at 17:25
Daniel
3,206 1 12 3

275

TL;DR

Instead of this:

```
def square_list(n):
    the_list = []                         # Replace
    for x in range(n):
        y = x * x
        the_list.append(y)                # these
    return the_list                       # lines
```

do this:

```
def square_yield(n):
    for x in range(n):
        y = x * x
        yield y                           # with this one.
```

Whenever you find yourself building a list from scratch, `yield` each piece instead.

This was my first "aha" moment with yield.

`yield` is a sugary way to say

> build a series of stuff

Same behavior:

```
>>> for square in square_list(4):
...     print(square)
...
0
1
4
9
>>> for square in square_yield(4):
...     print(square)
...
0
1
4
9
```

Different behavior:

Yield is **single-pass**: you can only iterate through once. When a function has a yield in it we call it a generator function. And an iterator is what it returns. Those terms are revealing. We lose the convenience of a container, but gain the power of a series that's computed as needed, and arbitrarily long.

Yield is **lazy**, it puts off computation. A function with a yield in it *doesn't actually execute at all when you call it.* It returns an iterator object that remembers where it left off. Each time you call `next()` on the iterator (this happens in a for-loop) execution inches forward to the next yield. `return` raises StopIteration and ends the series (this is the natural end of a for-loop).

Yield is **versatile**. Data doesn't have to be stored all together, it can be made available one at a time. It can be infinite.

```
>>> def squares_all_of_them():
...     x = 0
...     while True:
...         yield x * x
...         x += 1
...
>>> squares = squares_all_of_them()
>>> for _ in range(4):
...     print(next(squares))
...
0
1
4
9
```

If you need **multiple passes** and the series isn't too long, just call `list()` on it:

```
>>> list(square_yield(4))
[0, 1, 4, 9]
```

Brilliant choice of the word `yield` because both meanings apply:

> **yield** — produce or provide (as in agriculture)

...provide the next data in the series.

> **yield** — give way or relinquish (as in political power)

...relinquish CPU execution until the iterator advances.

Share Edit Follow

edited Jan 4, 2019 at 15:30

answered Mar 25, 2016 at 13:21

Bob Stein
15.2k 9 80 98

236

Yield gives you a generator.

```
def get_odd_numbers(i):
    return range(1, i, 2)
def yield_odd_numbers(i):
    for x in range(1, i, 2):
       yield x
foo = get_odd_numbers(10)
bar = yield_odd_numbers(10)
foo
[1, 3, 5, 7, 9]
bar
<generator object yield_odd_numbers at 0x1029c6f50>
bar.next()
1
bar.next()
3
bar.next()
5
```

As you can see, in the first case `foo` holds the entire list in memory at once. It's not a big deal for a list with 5 elements, but what if you want a list of 5 million? Not only is this a huge memory eater, it also costs a lot of time to build at the time that the function is called.

In the second case, `bar` just gives you a generator. A generator is an iterable--which means you can use it in a `for` loop, etc, but each value can only be accessed once. All the values are also not stored in memory at the same time; the generator object "remembers" where it was in the looping the last time you called it--this way, if you're using an iterable to (say) count to 50 billion, you don't have to count to 50 billion all at once and store the 50 billion numbers to count through.

Again, this is a pretty contrived example, you probably would use itertools if you really wanted to count to 50 billion. :)

This is the most simple use case of generators. As you said, it can be used to write efficient permutations, using yield to push things up through the call stack instead of using some sort of stack variable. Generators can also be used for specialized tree traversal, and all manner of other things.

Share Edit Follow

edited Mar 13, 2019 at 6:04

Andreas
2,395 10 20 23

answered Jan 16, 2013 at 6:42

RBansal
2,369 1 12 3

3 Just a note - in Python 3, `range` also returns a generator instead of a list, so you'd also see a similar idea, except that `__repr__` / `__str__` are overridden to show a nicer result, in this case `range(1, 10, 2)`. – It'sNotALie. Mar 21, 2019 at 18:33

235

It's returning a generator. I'm not particularly familiar with Python, but I believe it's the same kind of thing as [C#'s iterator blocks] if you're familiar with those.

The key idea is that the compiler/interpreter/whatever does some trickery so that as far as the caller is concerned, they can keep calling next() and it will keep returning values - *as if the generator method was paused*. Now obviously you can't really "pause" a method, so the compiler builds a state machine for you to remember where you currently are and what the local variables etc look like. This is much easier than writing an iterator yourself.

Share Edit Follow

edited Oct 31, 2018 at 8:42

answered Oct 23, 2008 at 22:26

Jon Skeet
1.4m 842 9009
9115

207

There is one type of answer that I don't feel has been given yet, among the many great answers that describe how to use generators. Here is the programming language theory answer:

The `yield` statement in Python returns a generator. A generator in Python is a function that returns *continuations* (and specifically a type of coroutine, but continuations represent the more general mechanism to understand what is going on).

Continuations in programming languages theory are a much more fundamental kind of computation, but they are not often used, because they are extremely hard to reason about and also very difficult to implement. But the idea of what a continuation is, is straightforward: it is the state of a computation that has not yet finished. In this state, the current values of variables, the operations that have yet to be performed, and so on, are saved. Then at some point later in the program the continuation can be invoked, such that the program's variables are reset to that state and the operations that were saved are carried out.

Continuations, in this more general form, can be implemented in two ways. In the `call/cc` way, the program's stack is literally saved and then when the continuation is invoked, the stack is restored.

In continuation passing style (CPS), continuations are just normal functions (only in languages where functions are first class) which the programmer explicitly manages and passes around to subroutines. In this style, program state is represented by closures (and the variables that happen to be encoded in them) rather than variables that reside somewhere on the stack. Functions that manage control flow accept continuation as arguments (in some variations of CPS, functions may accept multiple continuations) and manipulate control flow by invoking them by simply calling them and returning afterwards. A very simple example of continuation passing style is as follows:

```
def save_file(filename):
  def write_file_continuation():
    write_stuff_to_file(filename)

  check_if_file_exists_and_user_wants_to_overwrite(write_file_continuation)
```

In this (very simplistic) example, the programmer saves the operation of actually writing the file into a continuation (which can potentially be a very complex operation with many details to write out), and then passes that continuation (i.e, as a first-class closure) to another operator which does some more processing, and then calls it if necessary. (I use this design pattern a lot in actual GUI programming, either because it saves me lines of code or, more importantly, to manage control flow after GUI events trigger.)

The rest of this post will, without loss of generality, conceptualize continuations as CPS, because it is a hell of a lot easier to understand and read.

Now let's talk about generators in Python. Generators are a specific subtype of continuation. Whereas **continuations are able in general to save the state of a *computation*** (i.e., the program's call stack), **generators are only able to save the state of iteration over an *iterator***. Although, this definition is slightly misleading for certain use cases of generators. For instance:

```
def f():
  while True:
    yield 4
```

This is clearly a reasonable iterable whose behavior is well defined -- each time the generator iterates over it, it returns 4 (and does so forever). But it isn't probably the prototypical type of iterable that comes to mind when thinking of iterators (i.e., `for x in collection: do_something(x)`). This example illustrates the power of generators: if anything is an iterator, a generator can save the state of its iteration.

To reiterate: Continuations can save the state of a program's stack and generators can save the state of iteration. This means that continuations are more a lot powerful than generators, but also that generators are a lot, lot easier. They are easier for the language designer to implement, and they are easier for the programmer to use (if you have some time to burn, try to read and understand [this page about continuations and call/cc](#)).

But you could easily implement (and conceptualize) generators as a simple, specific case of continuation passing style:

Whenever `yield` is called, it tells the function to return a continuation. When the function is called again, it starts from wherever it left off. So, in pseudo-pseudocode (i.e., not pseudocode, but not code) the generator's `next` method is basically as follows:

```
class Generator():
  def __init__(self,iterable,generatorfun):
    self.next_continuation = lambda:generatorfun(iterable)

  def next(self):
    value, next_continuation = self.next_continuation()
    self.next_continuation = next_continuation
    return value
```

where the `yield` keyword is actually syntactic sugar for the real generator function, basically something like:

```
def generatorfun(iterable):
  if len(iterable) == 0:
    raise StopIteration
  else:
    return (iterable[0], lambda:generatorfun(iterable[1:]))
```

Remember that this is just pseudocode and the actual implementation of generators in Python is more complex. But as an exercise to understand what is going on, try to use continuation passing style to implement generator objects without use of the `yield` keyword.

Share Edit Follow

edited May 20, 2018 at 10:25
Peter Mortensen
30.6k 21 104 125

answered Apr 4, 2013 at 14:56
aestrivex
5,010 2 25 43

207

Here is an example in plain language. I will provide a correspondence between high-level human concepts to low-level Python concepts.

I want to operate on a sequence of numbers, but I don't want to bother my self with the creation of that sequence, I want only to focus on the operation I want to do. So, I do the following:

- I call you and tell you that I want a sequence of numbers which are calculated in a specific way, and I let you know what the algorithm is.
  This step corresponds to `def`ining the generator function, i.e. the function containing a `yield`.
- Sometime later, I tell you, "OK, get ready to tell me the sequence of numbers".
  This step corresponds to calling the generator function which returns a generator object. Note that you don't tell me any numbers yet; you just grab your paper and pencil.
- I ask you, "tell me the next number", and you tell me the first number; after that, you wait for me to ask you for the next number. It's your job to remember where you were, what numbers you have already said, and what is the next number. I don't care about the details.
  This step corresponds to calling `next(generator)` on the generator object.
  (In Python 2, `.next` was a method of the generator object; in Python 3, it is named `.__next__`, but the proper way to call it is using the builtin `next()` function just like `len()` and `.__len__`)
- … repeat previous step, until…
- eventually, you might come to an end. You don't tell me a number; you just shout, "hold your horses! I'm done! No more numbers!"
  This step corresponds to the generator object ending its job, and raising a `StopIteration` exception.
  The generator function does not need to raise the exception. It's raised automatically when the function ends or issues a `return`.

This is what a generator does (a function that contains a `yield`); it starts executing on the first `next()`, pauses whenever it does a `yield`, and when asked for the `next()` value it continues from the point it was last. It fits perfectly by design with the iterator protocol of Python, which describes how to sequentially request values.

The most famous user of the iterator protocol is the `for` command in Python. So, whenever you do a:

```
for item in sequence:
```

it doesn't matter if `sequence` is a list, a string, a dictionary or a generator *object* like described above; the result is the same: you read items off a sequence one by one.

Note that `def`ining a function which contains a `yield` keyword is not the only way to create a generator; it's just the easiest way to create one.

For more accurate information, read about [iterator types](), the [yield statement]() and [generators]() in the Python documentation.

Share Edit Follow

edited Jan 17 at 10:21

answered Oct 24, 2008 at 0:36

tzot
89.8k 29 138 202

162

While a lot of answers show why you'd use a `yield` to create a generator, there are more uses for `yield`. It's quite easy to make a coroutine, which enables the passing of information between two blocks of code. I won't repeat any of the fine examples that have already been given about using `yield` to create a generator.

To help understand what a `yield` does in the following code, you can use your finger to trace the cycle through any code that has a `yield`. Every time your finger hits the `yield`, you have to wait for a `next` or a `send` to be entered. When a `next` is called, you trace through the code until you hit the `yield`… the code on the right of the `yield` is evaluated and returned to the caller… then you wait. When `next` is called again, you perform another loop through the code. However, you'll note that in a coroutine, `yield` can also be used with a `send`… which will send a value from the caller *into* the yielding function. If a `send` is given, then `yield` receives the value sent, and spits it out the left hand side… then the trace through the code progresses until you hit the `yield` again (returning the value at the end, as if `next` was called).

For example:

```
>>> def coroutine():
...     i = -1
```

```
...     while True:
...         i += 1
...         val = (yield i)
...         print("Received %s" % val)
...
>>> sequence = coroutine()
>>> sequence.next()
0
>>> sequence.next()
Received None
1
>>> sequence.send('hello')
Received hello
2
>>> sequence.close()
```

Share Edit Follow

answered Feb 4, 2014 at 2:27

Mike McKerns
32.2k 8 114 138

2 Cute! A trampoline (in the Lisp sense). Not often one sees those! – 00prometheus Dec 4, 2015 at 18:31

158

There is another `yield` use and meaning (since Python 3.3):

```
yield from <expr>
```

From *PEP 380 -- Syntax for Delegating to a Subgenerator*:

> A syntax is proposed for a generator to delegate part of its operations to another generator. This allows a section of code containing 'yield' to be factored out and placed in another generator. Additionally, the subgenerator is allowed to return with a value, and the value is made available to the delegating generator.
>
> The new syntax also opens up some opportunities for optimisation when one generator re-yields values produced by another.

Moreover this will introduce (since Python 3.5):

```
async def new_coroutine(data):
   ...
   await blocking_action()
```

to avoid coroutines being confused with a regular generator (today `yield` is used in both).

Share Edit Follow

edited Jun 20, 2020 at 9:12

Community Bot
1 1

answered Jul 24, 2014 at 21:15

Sławomir Lenart
6,732 3 40 58

146

All great answers, however a bit difficult for newbies.

I assume you have learned the `return` statement.

As an analogy, `return` and `yield` are twins. `return` means 'return and stop' whereas 'yield` means 'return, but continue'

1. Try to get a num_list with `return`.

```
def num_list(n):
    for i in range(n):
        return i
```

Run it:

```
In [5]: num_list(3)
Out[5]: 0
```

See, you get only a single number rather than a list of them. `return` never allows you prevail happily, just implements once and quit.

2. There comes `yield`

Replace `return` with `yield`:

```
In [10]: def num_list(n):
    ...:     for i in range(n):
    ...:         yield i
    ...:

In [11]: num_list(3)
Out[11]: <generator object num_list at 0x10327c990>

In [12]: list(num_list(3))
Out[12]: [0, 1, 2]
```

Now, you win to get all the numbers.

Comparing to `return` which runs once and stops, `yield` runs times you planed. You can interpret `return` as `return one of them`, and `yield` as `return all of them`. This is called `iterable`.

3. One more step we can rewrite `yield` statement with `return`

```
In [15]: def num_list(n):
    ...:     result = []
    ...:     for i in range(n):
    ...:         result.append(i)
    ...:     return result

In [16]: num_list(3)
Out[16]: [0, 1, 2]
```

It's the core about `yield`.

The difference between a list `return` outputs and the object `yield` output is:

You will always get [0, 1, 2] from a list object but only could retrieve them from 'the object `yield` output' once. So, it has a new name `generator` object as displayed in `Out[11]: <generator object num_list at 0x10327c990>`.

In conclusion, as a metaphor to grok it:

- `return` and `yield` are twins
- `list` and `generator` are twins

Share Edit Follow

edited May 28, 2018 at 9:06

answered Nov 14, 2017 at 12:02

AbstProcDo
18.9k 15 74 124

3 This is understandable, but one major difference is that you can have multiple yields in a function/method. The analogy totally breaks down at that point. Yield remembers its place in a function, so the next time you call next(), your function continues on to the next `yield` . This is important, I think, and should be expressed. – Mike S Aug 23, 2018 at 13:27

133

From a programming viewpoint, the iterators are implemented as thunks.

To implement iterators, generators, and thread pools for concurrent execution, etc. as thunks, one uses messages sent to a closure object, which has a dispatcher, and the dispatcher answers to "messages".

"next" is a message sent to a closure, created by the "*iter*" call.

There are lots of ways to implement this computation. I used mutation, but it is possible to do this kind of computation without mutation, by returning the current value and the next yielder (making it referential transparent). Racket uses a sequence of transformations of the initial program in some intermediary languages, one of such rewriting making the yield operator to be transformed in some language with simpler operators.

Here is a demonstration of how yield could be rewritten, which uses the structure of R6RS, but the semantics is identical to Python's. It's the same model of computation, and only a change in syntax is required to rewrite it using yield of Python.

```
Welcome to Racket v6.5.0.3.

-> (define gen
     (lambda (l)
       (define yield
         (lambda ()
           (if (null? l)
               'END
               (let ((v (car l)))
                 (set! l (cdr l))
                 v))))
       (lambda(m)
         (case m
           ('yield (yield))
           ('init  (lambda (data)
                     (set! l data)
                     'OK))))))
-> (define stream (gen '(1 2 3)))
-> (stream 'yield)
1
-> (stream 'yield)
2
-> (stream 'yield)
3
-> (stream 'yield)
'END
-> ((stream 'init) '(a b))
'OK
-> (stream 'yield)
'a
-> (stream 'yield)
'b
-> (stream 'yield)
'END
```

```
    -> (stream 'yield)
    'END
    ->
```

Share Edit Follow

edited Jul 2, 2020 at 7:36

answered Aug 21, 2013 at 19:01

alinsoar
14.9k 4 55 71

127

Here are some Python examples of how to actually implement generators as if Python did not provide syntactic sugar for them:

As a Python generator:

```
from itertools import islice

def fib_gen():
    a, b = 1, 1
    while True:
        yield a
        a, b = b, a + b

assert [1, 1, 2, 3, 5] == list(islice(fib_gen(), 5))
```

Using lexical closures instead of generators

```
def ftake(fnext, last):
    return [fnext() for _ in xrange(last)]

def fib_gen2():
    #funky scope due to python2.x workaround
    #for python 3.x use nonlocal
    def _():
        _.a, _.b = _.b, _.a + _.b
        return _.a
    _.a, _.b = 0, 1
    return _

assert [1,1,2,3,5] == ftake(fib_gen2(), 5)
```

Using object closures instead of generators (because ClosuresAndObjectsAreEquivalent)

```
class fib_gen3:
    def __init__(self):
        self.a, self.b = 1, 1

    def __call__(self):
        r = self.a
        self.a, self.b = self.b, self.a + self.b
        return r

assert [1,1,2,3,5] == ftake(fib_gen3(), 5)
```

Share Edit Follow

edited Oct 24, 2017 at 10:46

answered Oct 3, 2012 at 20:38

Dustin Getz
20.9k 14 81 129

I was going to post "read page 19 of Beazley's 'Python: Essential Reference' for a quick description of generators", but so many others have posted

116

good descriptions already.

Also, note that `yield` can be used in coroutines as the dual of their use in generator functions. Although it isn't the same use as your code snippet, `(yield)` can be used as an expression in a function. When a caller sends a value to the method using the `send()` method, then the coroutine will execute until the next `(yield)` statement is encountered.

Generators and coroutines are a cool way to set up data-flow type applications. I thought it would be worthwhile knowing about the other use of the `yield` statement in functions.

Share Edit Follow

answered Jan 28, 2013 at 1:37

johnzachary

2,395 2 18 9

100

Here is a simple example:

```
def isPrimeNumber(n):
    print "isPrimeNumber({}) call".format(n)
    if n==1:
        return False
    for x in range(2,n):
        if n % x == 0:
            return False
    return True

def primes (n=1):
    while(True):
        print "loop step ---------------- {}".format(n)
        if isPrimeNumber(n): yield n
        n += 1

for n in primes():
    if n> 10:break
    print "wiriting result {}".format(n)
```

Output:

```
loop step ---------------- 1
isPrimeNumber(1) call
loop step ---------------- 2
isPrimeNumber(2) call
loop step ---------------- 3
isPrimeNumber(3) call
wiriting result 3
loop step ---------------- 4
isPrimeNumber(4) call
loop step ---------------- 5
isPrimeNumber(5) call
wiriting result 5
loop step ---------------- 6
isPrimeNumber(6) call
loop step ---------------- 7
isPrimeNumber(7) call
wiriting result 7
loop step ---------------- 8
isPrimeNumber(8) call
loop step ---------------- 9
isPrimeNumber(9) call
loop step ---------------- 10
isPrimeNumber(10) call
loop step ---------------- 11
isPrimeNumber(11) call
```

I am not a Python developer, but it looks to me `yield` holds the position of program flow and the next loop start from "yield" position. It seems like it is waiting at that position, and just before that, returning a value outside, and next time continues to work.

It seems to be an interesting and nice ability :D

Share Edit Follow

edited May 20, 2018 at 10:31
Peter Mortensen
30.6k 21 104 125

answered Dec 20, 2013 at 13:07
Engin OZTURK
2,035 2 21 13

86

Here is a mental image of what `yield` does.

I like to think of a thread as having a stack (even when it's not implemented that way).

When a normal function is called, it puts its local variables on the stack, does some computation, then clears the stack and returns. The values of its local variables are never seen again.

With a `yield` function, when its code begins to run (i.e. after the function is called, returning a generator object, whose `next()` method is then invoked), it similarly puts its local variables onto the stack and computes for a while. But then, when it hits the `yield` statement, before clearing its part of the stack and returning, it takes a snapshot of its local variables and stores them in the generator object. It also writes down the place where it's currently up to in its code (i.e. the particular `yield` statement).

So it's a kind of a frozen function that the generator is hanging onto.

When `next()` is called subsequently, it retrieves the function's belongings onto the stack and re-animates it. The function continues to compute from where it left off, oblivious to the fact that it had just spent an eternity in cold storage.

Compare the following examples:

```
def normalFunction():
    return
    if False:
        pass

def yielderFunction():
    return
    if False:
        yield 12
```

When we call the second function, it behaves very differently to the first. The `yield` statement might be unreachable, but if it's present anywhere, it changes the nature of what we're dealing with.

```
>>> yielderFunction()
<generator object yielderFunction at 0x07742D28>
```

Calling `yielderFunction()` doesn't run its code, but makes a generator out of the code. (Maybe it's a good idea to name such things with the `yielder` prefix for readability.)

```
>>> gen = yielderFunction()
>>> dir(gen)
['__class__',
 ...
 '__iter__',    #Returns gen itself, to make it work uniformly with containers
 ...            #when given to a for loop. (Containers return an iterator
instead.)
 'close',
```

```
 'gi_code',
 'gi_frame',
 'gi_running',
 'next',         #The method that runs the function's body.
 'send',
 'throw']
```

The `gi_code` and `gi_frame` fields are where the frozen state is stored. Exploring them with `dir(..)`, we can confirm that our mental model above is credible.

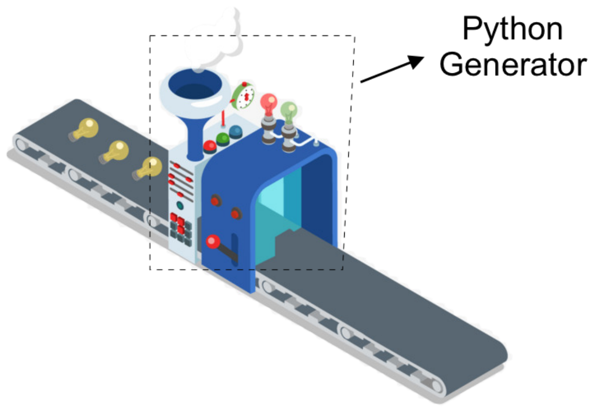
Share Edit Follow

edited Mar 1, 2017 at 13:36

answered Jun 14, 2013 at 16:36

Evgeni Sergeev
21.8k 16 103 123

82



Imagine that you have created a remarkable machine that is capable of generating thousands and thousands of lightbulbs per day. The machine generates these lightbulbs in boxes with a unique serial number. You don't have enough space to store all of these lightbulbs at the same time, so you would like to adjust it to generate lightbulbs on-demand.

Python generators don't differ much from this concept. Imagine that you have a function called `barcode_generator` that generates unique serial numbers for the boxes. Obviously, you can have a huge number of such barcodes returned by the function, subject to the hardware (RAM) limitations. A wiser, and space efficient, option is to generate those serial numbers on-demand.

Machine's code:

```
def barcode_generator():
    serial_number = 10000  # Initial barcode
    while True:
        yield serial_number
        serial_number += 1


barcode = barcode_generator()
while True:
    number_of_lightbulbs_to_generate = int(input("How many lightbulbs to 
generate? "))
    barcodes = [next(barcode) for _ in range(number_of_lightbulbs_to_generate)]
    print(barcodes)

    # function_to_create_the_next_batch_of_lightbulbs(barcodes)

    produce_more = input("Produce more? [Y/n]: ")
    if produce_more == "n":
        break
```

Note the `next(barcode)` bit.

As you can see, we have a self-contained “function” to generate the next unique serial number each time. This function returns a *generator*! As you can see, we are not calling the function each time we need a new serial number, but instead we are using `next()` given the generator to obtain the next serial number.

Lazy Iterators

To be more precise, this generator is a *lazy iterator*! An iterator is an object that helps us traverse a sequence of objects. It's called *lazy* because it does not load all the items of the sequence in memory until they are needed. The use of `next` in the previous example is the *explicit* way to obtain the next item from the iterator. The *implicit* way is using for loops:

```
for barcode in barcode_generator():
    print(barcode)
```

This will print barcodes infinitely, yet you will not run out of memory.

In other words, a generator *looks like* a function but *behaves like* an iterator.

Real-world application?

Finally, real-world applications? They are usually useful when you work with big sequences. Imagine reading a *huge* file from disk with billions of records. Reading the entire file in memory, before you can work with its content, will probably be infeasible (i.e., you will run out of memory).

Share Edit Follow

edited Oct 21, 2021 at 18:54

answered Mar 23, 2019 at 13:55
Dr Rafael
6,717 5 42 50

78

An easy example to understand what it is: `yield`

```
def f123():
    for _ in range(4):
        yield 1
        yield 2


for i in f123():
    print (i)
```

The output is:

```
1 2 1 2 1 2 1 2
```

Share Edit Follow

edited Feb 2, 2020 at 18:21
ZF007
3,656 8 33 48

answered Jan 2, 2017 at 12:09
Gavriel Cohen
4,099 32 38

8 are you sure about that output? wouldnt that only be printed on a single line if you ran that print statement using `print(i, end=' ')` ? Otherwise, i believe the default behavior would put each number on a new line – user9074332 Feb 5, 2020 at 4:05

3 @user9074332, You're right, but it is written on one line to facilitate understanding – Gavriel Cohen Feb 5, 2020 at 14:58

74

Like every answer suggests, `yield` is used for creating a sequence generator. It's used for generating some sequence dynamically. For example, while reading a file line by line on a network, you can use the `yield` function as follows:

```
def getNextLines():
   while con.isOpen():
       yield con.read()
```

You can use it in your code as follows:

```
for line in getNextLines():
    doSomeThing(line)
```

Execution Control Transfer gotcha

The execution control will be transferred from getNextLines() to the `for` loop when yield is executed. Thus, every time getNextLines() is invoked, execution begins from the point where it was paused last time.

Thus in short, a function with the following code

```
def simpleYield():
    yield "first time"
    yield "second time"
    yield "third time"
    yield "Now some useful value {}".format(12)

for i in simpleYield():
    print i
```

will print

```
"first time"
"second time"
"third time"
"Now some useful value 12"
```

Share Edit Follow

edited May 20, 2018 at 10:42
Peter Mortensen
30.6k 21 104 125

answered Jul 29, 2015 at 6:11
Mangu Singh Rajpurohit
10.2k 4 61 94

72

(My below answer only speaks from the perspective of using Python generator, not the [underlying implementation of generator mechanism](), which involves some tricks of stack and heap manipulation.)

When `yield` is used instead of a `return` in a python function, that function is turned into something special called `generator function`. That function will return an object of `generator` type. **The `yield` keyword is a flag to notify the python compiler to treat such function specially.** Normal functions will terminate once some value is returned from it. But with the help of the compiler, the generator function **can be thought of** as resumable. That is, the execution context will be restored and the execution will continue from last run. Until you explicitly call return, which will raise a `StopIteration` exception (which is also part of the iterator protocol), or reach the end of the function. I found a lot of references about `generator` but this [one]() from the `functional programming perspective` is the most digestable.

(Now I want to talk about the rationale behind `generator`, and the `iterator` based on my own understanding. I hope this can help you grasp the ***essential motivation*** of iterator and generator. Such concept shows up in other languages as well such as C#.)

As I understand, when we want to process a bunch of data, we usually first store the data somewhere and then process it one by one. But this *naive* approach is problematic. If the data volume is huge, it's expensive to store them as a whole beforehand. **So instead of storing the `data` itself directly, why not store some kind of `metadata` indirectly, i.e. `the logic how the data is computed`**.

There are 2 approaches to wrap such metadata.

1. The OO approach, we wrap the metadata `as a class`. This is the so-called `iterator` who implements the iterator protocol (i.e. the `__next__()`, and `__iter__()` methods). This is also the commonly seen iterator design pattern.
2. The functional approach, we wrap the metadata `as a function`. This is the so-called `generator function`. But under the hood, the returned `generator object` still `IS-A` iterator because it also implements the iterator protocol.

Either way, an iterator is created, i.e. some object that can give you the data you want. The OO approach may be a bit complex. Anyway, which one to use is up to you.

Share Edit Follow

edited Nov 23, 2018 at 1:38

answered Mar 25, 2016 at 5:40

smwikipedia
59.5k 90 293 468

71

In summary, the `yield` statement transforms your function into a factory that produces a special object called a `generator` which wraps around the body of your original function. When the `generator` is iterated, it executes your function until it reaches the next `yield` then suspends execution and evaluates to the value passed to `yield`. It repeats this process on each iteration until the path of execution exits the function. For instance,

```
def simple_generator():
    yield 'one'
    yield 'two'
    yield 'three'

for i in simple_generator():
    print i
```

simply outputs

```
one
two
three
```

The power comes from using the generator with a loop that calculates a sequence, the generator executes the loop stopping each time to 'yield' the next result of the calculation, in this way it calculates a list on the fly, the benefit being the memory saved for especially large calculations

Say you wanted to create a your own `range` function that produces an iterable range of numbers, you could do it like so,

```
def myRangeNaive(i):
    n = 0
    range = []
    while n < i:
        range.append(n)
        n = n + 1
    return range
```

and use it like this;

```
for i in myRangeNaive(10):
    print i
```

But this is inefficient because

- You create an array that you only use once (this wastes memory)
- This code actually loops over that array twice! :(

Luckily Guido and his team were generous enough to develop generators so we could just do this;

```
def myRangeSmart(i):
    n = 0
    while n < i:
       yield n
       n = n + 1
    return

for i in myRangeSmart(10):
    print i
```

Now upon each iteration a function on the generator called `next()` executes the function until it either reaches a 'yield' statement in which it stops and 'yields' the value or reaches the end of the function. In this case on the first call, `next()` executes up to the yield statement and yield 'n', on the next call it will execute the increment statement, jump back to the 'while', evaluate it, and if true, it will stop and yield 'n' again, it will continue that way until the while condition returns false and the generator jumps to the end of the function.

Share Edit Follow

edited May 20, 2018 at 11:04
Peter Mortensen
30.6k 21 104 125

answered Oct 13, 2016 at 13:43
redbandit
2,072 15 11

67

Yield is an object

A `return` in a function will return a single value.

If you want **a function to return a huge set of values**, use `yield`.

More importantly, `yield` is a **barrier**.

> like barrier in the CUDA language, it will not transfer control until it gets completed.

That is, it will run the code in your function from the beginning until it hits `yield`. Then, it'll return the first value of the loop.

Then, every other call will run the loop you have written in the function one more time, returning the next value until there isn't any value to return.

Share Edit Follow

edited May 20, 2018 at 10:45
Peter Mortensen
30.6k 21 104 125

answered Sep 1, 2015 at 12:42
Kaleem Ullah
6,501 3 39 47

65

Many people use `return` rather than `yield`, but in some cases `yield` can be more efficient and easier to work with.

Here is an example which `yield` is definitely best for:

return (in function)

```
import random

def return_dates():
    dates = [] # With 'return' you need to create a list then return it
    for i in range(5):
        date = random.choice(["1st", "2nd", "3rd", "4th", "5th", "6th", "7th",
"8th", "9th", "10th"])
        dates.append(date)
    return dates
```

yield (in function)

```
def yield_dates():
    for i in range(5):
        date = random.choice(["1st", "2nd", "3rd", "4th", "5th", "6th", "7th",
"8th", "9th", "10th"])
        yield date # 'yield' makes a generator automatically which works
                   # in a similar way. This is much more efficient.
```

Calling functions

```
dates_list = return_dates()
print(dates_list)
for i in dates_list:
    print(i)

dates_generator = yield_dates()
print(dates_generator)
for i in dates_generator:
    print(i)
```

Both functions do the same thing, but `yield` uses three lines instead of five and has one less variable to worry about.

This is the result from the code:

```
>>> ================================ RESTART ================================
>>>
['6th', '4th', '9th', '4th', '10th']
6th
4th
9th
4th
10th
<generator object yield_dates at 0x031CF2B0>
8th
8th
7th
5th
9th
>>>
```

As you can see both functions do the same thing. The only difference is `return_dates()` gives a list and `yield_dates()` gives a generator.

A real life example would be something like reading a file line by line or if you just want to make a generator.

Share Edit Follow

edited May 20, 2018 at 11:02
Peter Mortensen
30.6k 21 104 125

answered Sep 10, 2016 at 11:37
Tom Fuller
5,081 6 32 42

55

The `yield` keyword simply collects returning results. Think of `yield` like `return +=`

Share Edit Follow

edited Apr 23, 2016 at 3:16
Phillip
2,165 3 25 40

answered Nov 18, 2015 at 19:37
Bahtiyar Özdere
1,734 16 20

52

`yield` is like a return element for a function. The difference is, that the `yield` element turns a function into a generator. A generator behaves just like a function until something is 'yielded'. The generator stops until it is next called, and continues from exactly the same point as it started. You can get a sequence of all the 'yielded' values in one, by calling `list(generator())`.

Share Edit Follow

answered May 20, 2015 at 6:19
Will Dereham
986 10 33

1 2 Next