Python 201: An Intro to itertools

April 20, 2016 Python, Python 3 Python, Python 201 Mike

Python provides a great module for creating your own iterators. The module I am referring to is **itertools**. The tools provided by itertools are fast and memory efficient. You will be able to take these building blocks to create your own specialized iterators that can be used for efficient looping. In this chapter, we will be looking at examples of each building block so that by the end you will understand how to use them for your own code bases.

Let's get started by looking at some infinite iterators!

The infinite iterators

The itertools package comes with three iterators that can iterate infinitely. What this means is that when you use them, you need to understand that you will need to break out of these iterators eventually or you'll have an infinite loop.

These can be useful for generating numbers or cycling over iterables of unknown length, for example. Let's get started learning about these interesting iterables!

count(start=0, step=1)

The **count** iterator will return evenly spaced values starting with the number you pass in as its **start** parameter. Count also accept a **step** parameter. Let's take a look at a simple example:

```
>>> from itertools import count
>>> for i in count(10):
...     if i > 20:
...         break
...     else:
...         print(i)
...
10
11
12
13
14
15
16
17
18
19
20
```

Here we import **count** from itertools and we create a **for** loop. We add a conditional check that will break out of the loop should the iterator exceed 20, otherwise it prints out where we are in the iterator. You will note that the output starts at 10 as that was what we passed to **count** as our start value.

Another way to limit the output of this infinite iterator is to use another sub-module from itertools, namely **islice**. Here's how:

```
>>> from itertools import islice
>>> for i in islice(count(10), 5):
...     print(i)
...
10
11
12
13
14
```

Here we import **islice** and we loop over **count** starting at 10 and ending after 5 items. As you may have guessed, the second argument to islice is when to stop iterating. But it doesn't mean "stop when I reach the number 5". Instead, it means "stop when we've reached five iterations".

cycle(iterable)

The **cycle** iterator from itertools allows you to create an iterator that will cycle through a series of values infinitely. Let's pass it a 3 letter string and see what happens:

```
>>> from itertools import cycle
>>> count = 0
>>> for item in cycle('XYZ'):
...     if count > 7:
...         break
...     print(item)
...     count += 1
...
X
Y
Z
X
Y
Z
X
Y
```

Here we create a **for** loop to loop over the infinite cycle of the three letter: XYZ. Of course, we don't want to actually cycle forever, so we add a simple counter to break out of the loop with.

You can also use Python's **next** built-in to iterate over the iterators you create with itertools:

```
>>> polys = ['triangle', 'square', 'pentagon', 'rectangle']
 >>> iterator = cycle(polys)
 >>> next(iterator)
 'triangle'
 >>> next(iterator)
 'square'
 >>> next(iterator)
 'pentagon'
 >>> next(iterator)
 'rectangle'
 >>> next(iterator)
 'triangle'
 >>> next(iterator)
 'square'
```

In the code above, we create a simple list of polygons and pass them to **cycle**. We save our new iterator to a variable and then we pass that variable to the **next** function. Every time we call next, it returns the next value in the iterator. Since this iterator is infinite, we can call next all day long and never run out of items.

repeat(object[, times])

The **repeat** iterators will return an object an object over and over again forever unless you set its **times** argument. It is quite similar to **cycle** except that it doesn't cycle over a set of values repeatedly. Let's take a look at a simple example:

```
>>> from itertools import repeat
>>> repeat(5, 5)
repeat(5, 5)
>>> iterator = repeat(5, 5)
>>> next(iterator)
5
>>> next(iterator)
5
>>> next(iterator)
5
>>> next(iterator)
5
>>> next(iterator)
5
>>> next(iterator)
Traceback (most recent call last):
  Python Shell, prompt 21, line 1
builtins.StopIteration:
```

Here we import **repeat** and tell it to repeat the number 5 five times. Then we call **next** on our new iterator six times to see if it works correctly. When you run this code, you will see that **StopIteration** gets raised because we have run out of values in our iterator.

Iterators that terminate

Most of the iterators that you create with itertools are not infinite. In this sections, we will be studying the finite iterators of itertools. To get output that is readable, we will be using Python's built-in **list** type. If you do not use **list**, then you will only get an itertools object printed out.

accumulate(iterable[, func])

The **accumulate** iterator will return accumulated sums or the accumulated results of a two argument function that you can pass to **accumulate**. The default of accumulate is addition, so let's give that a quick try:

```
>> from itertools import accumulate
>>> list(accumulate(range(10)))
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
```

Here we import **accumulate** and pass it a range of 10 numbers, 0-9. It adds each of them in turn, so the first is 0, the second is 0+1, the 3rd is 1+2, etc. Now let's import the **operator** module and add it into the mix:

```
>>> import operator
>>> list(accumulate(range(1, 5), operator.mul))
[1, 2, 6, 24]
```

Here we pass the number 1-4 to our **accumulate** iterator. We also pass it a function: **operator.mul**. This functions accepts to arguments to be multiplied. So for each iteration, it multiplies instead of adds (1×1=1, 1×2=2, 2×3=6, etc).

The documentation for accumulate shows some other interesting examples such as the amortization of a loan or the chaotic recurrence relation. You should definitely give those examples a look as they are will worth your time.

chain(*iterables)

The **chain** iterator will take a series of iterables and basically flatten them down into one long iterable. I actually recently needed its assistance in a project I was helping with. Basically we had a list with some items already in it and two other lists that we wanted to add to the original list, but we only wanted to add the items in each list. Originally I tried something like this:

```
>>> my_list = ['foo', 'bar']
>>> numbers = list(range(5))
>>> cmd = ['ls', '/some/dir']
>>> my_list.extend(cmd, numbers)
>>> my_list
['foo', 'bar', ['ls', '/some/dir'], [0, 1, 2, 3, 4]]
```

Well that didn't work quite the way I wanted it to. The itertools module provides a much more elegant way of flattening these lists into one using **chain**:

```
>>> from itertools import chain
>>> my_list = list(chain(['foo', 'bar'], cmd, numbers))
>>> my_list
['foo', 'bar', 'ls', '/some/dir', 0, 1, 2, 3, 4]
```

My more astute readers might notice that there's actually another way we could have accomplished the same thing without using itertools. You could do this to get the same effect:

```
>>> my_list = ['foo', 'bar']
>>> my_list += cmd + numbers
>>> my_list
['foo', 'bar', 'ls', '/some/dir', 0, 1, 2, 3, 4]
```

Both of these methods are certainly valid and before I knew about **chain** I would have probably gone this route, but I think chain is a more elegant and easier to understand solution in this particular case.

chain.from_iterable(iterable)

You can also use a method of **chain** called **from_iterable**. This method works slightly differently then using chain directly. Instead of passing in a series of iterables, you have to pass in a nested list. Let's take a look:

```
>>> from itertools import chain
>>> numbers = list(range(5))
>>> cmd = ['ls', '/some/dir']
>>> chain.from_iterable(cmd, numbers)
Traceback (most recent call last):
  Python Shell, prompt 66, line 1
builtins.TypeError: from_iterable() takes exactly one argument (2
given)
>>> list(chain.from_iterable([cmd, numbers]))
['ls', '/some/dir', 0, 1, 2, 3, 4]
```

Here we import chain as we did previously. We try passing in our two lists but we end up getting a **TypeError**! To fix this, we change our call slightly such that we put **cmd** and **numbers** inside a **list** and then pass that nested list to **from_iterable**. It's a subtle difference but still easy to use!

compress(data, selectors)

The **compress** sub-module is useful for filtering the first iterable with the second. This works by making the second iterable a list of Booleans (or ones and zeroes which amounts to the same thing). Here's how it works:

```
>>> from itertools import compress
>>> letters = 'ABCDEFG'
>>> bools = [True, False, True, True, False]
>>> list(compress(letters, bools))
['A', 'C', 'D']
```

In this example, we have a group of seven letters and a list of five Bools. Then we pass them into the compress function. the compress function will go through each respective iterable and check the first against the second. If the second has a matching True, then it will be kept. If it's a False, then that item will be dropped. Thus if you study the example above, you will notice that we have a True in the first, third and fourth positions which correspond with A,C and D.

dropwhile(predicate, iterable)

There is a neat little iterator contained in itertools called **dropwhile** This fun little iterator will drop elements as long as the filter criteria is True. Because of this, you will not see any output from this iterator until the predicate becomes False. This can make the start-up time lengthy, so it's something to be aware of.

Let's look at an example from Python's documentation:

```
>>> from itertools import dropwhile
>>> list(dropwhile(lambda x: x<5, [1,4,6,4,1]))
[6, 4, 1]
```

Here we import **dropwhile** and then we pass it a simple **lambda** statement. This function will return True if **x** is less than 5. Other it will return False. The dropwhile function will loop over the list and pass each element into the lambda. If the lambda returns True, then that value gets dropped. Once we reach the number 6, the lambda returns False and we retain the number 6 and all the values that follow it.

I find it useful to use a regular function over a lambda when I'm learning something new. So let's flip this on its head and create a function that returns True if the number is greater than 5.

```
>>> from itertools import dropwhile
>>> def greater_than_five(x):
...     return x > 5
...
>>> list(dropwhile(greater_than_five, [6, 7, 8, 9, 1, 2, 3,
10]))
[1, 2, 3, 10]
```

Here we create a simple function in Python's interpreter. This function is our predicate or filter. If the values we pass to it are True, then they will get dropped. Once we hit a value that is less than 5, then ALL the values after and including that value will be kept, which you can see in the example above.

filterfalse(predicate, iterable)

The **filterfalse** function from itertools is very similar to **dropwhile**. However instead of dropping values that match True, filterfalse will only return those values that evaluated to False. Let's use our function from the previous section to illustrate:

```
>>> from itertools import filterfalse
>>> def greater_than_five(x):
...     return x > 5
...
>>> list(filterfalse(greater_than_five, [6, 7, 8, 9, 1, 2, 3,
10]))
[1, 2, 3]
```

Here we pass filterfalse our function and a list of integers. If the integer is less than 5, it is kept. Otherwise it is thrown away. You will notice that our result is only 1, 2 and 3. Unlike dropwhile, filterfalse will check each and every value against our predicate.

groupby(iterable, key=None)

The **groupby** iterator will return consecutive keys and groups from your iterable. This one is kind of hard to wrap your head around without seeing an example. So let's take a look at one! Put the following code into your interpreter or save it in a file:

```
from itertools import groupby

vehicles = [('Ford', 'Taurus'), ('Dodge', 'Durango'),
            ('Chevrolet', 'Cobalt'), ('Ford', 'F150'),
            ('Dodge', 'Charger'), ('Ford', 'GT')]

sorted_vehicles = sorted(vehicles)

for key, group in groupby(sorted_vehicles, lambda make: make[0]):
    for make, model in group:
        print('{model} is made by {make}'.format(model=model,
                                                 make=make))
        print ("**** END OF GROUP ***\n")
```

Here we import **groupby** and then create a list of tuples. Then we sort the data so it makes more sense when we output it and it also lets groupby actually group items correctly. Next we actually loop over the iterator returned by groupby which gives us the key and the group. Then we loop over the group and print out what's in it. If you run this code, you should see something like this:

```
Cobalt is made by Chevrolet
**** END OF GROUP ***

Charger is made by Dodge
Durango is made by Dodge
**** END OF GROUP ***
```

```
F150 is made by Ford
GT is made by Ford
Taurus is made by Ford
**** END OF GROUP ***
```

Just for fun, try changing the code such that you pass in **vehicles** instead of **sorted_vehicles**. You will quickly learn why you should sort the data before running it through groupby if you do.

islice(iterable, start, stop[, step])

We actually mentioned **islice** way back in the **count** section. But here we'll look at it a little more in depth. islice is an iterator that returns selected elements from the iterable. That's kind of an opaque statement. Basically what islice does is take a slice by index of your iterable (the thing you iterate over) and returns the selected items as an iterator. There are actually two implementations of islice. There's **itertools.islice(iterable, stop)** and then there's the version of islice that more closely matches regular Python slicing: **islice(iterable, start, stop[, step])**.

Let's look at the first version to see how it works:

```
>>> from itertools import islice
>>> iterator = islice('123456', 4)
>>> next(iterator)
'1'
>>> next(iterator)
'2'
>>> next(iterator)
'3'
>>> next(iterator)
'4'
>>> next(iterator)
Traceback (most recent call last):
  Python Shell, prompt 15, line 1
builtins.StopIteration:
```

In the code above, we pass a string of six characters to islice along with the number 4 which is the stop argument. What this means is that the iterator that islice returns will have the first 4 items from the string in it. We can verify this by calling **next** on our iterator four times, which is what we do above. Python is smart enough to know that if there are only two arguments passed to islice, then the second argument is the **stop** argument.

Let's try giving it three arguments to demonstrate that you can pass it a start and a stop argument. The **count** tool from itertools can help us illustrate this concept:

```
>>> from itertools import islice
>>> from itertools import count
>>> for i in islice(count(), 3, 15):
...     print(i)
```

```
...
3
4
5
6
7
8
9
10
11
12
13
14
```

Here we just call count and tell it to islice to start at the number 3 and stop when we reach 15. It's just like doing a slice except that you are doing it to an iterator and returning a new iterator!

starmap(function, iterable)

The **starmap** tool will create an iterator that can compute using the function and iterable provided. As the documentation mentions, "the difference between map() and starmap() parallels the distinction between *function(a,b)* and *function(*c)*."

Let's look at a simple example:

```
>>> from itertools import starmap
>>> def add(a, b):
...     return a+b
...
>>> for item in starmap(add, [(2,3), (4,5)]):
...     print(item)
...
5
9
```

Here we create a simple adding function that accepts two arguments. Next we create a **for** loop and call **starmap** with the function as its first argument and a list of tuples for the iterable. The starmap function will then pass each tuple element into the function and return an iterator of the results, which we print out.

takewhile(predicate, iterable)

The **takewhile** module is basically the opposite of the **dropwhile** iterator that we looked at earlier. takewhile will create an iterator that returns elements from the iterable only as long as our predicate or filter is True. Let's try a simple example to see how it works:

```
>>> from itertools import takewhile
>>> list(takewhile(lambda x: x<5, [1,4,6,4,1]))
[1, 4]
```

Here we run takewhile using a lambda function and a list. The output is only the first two integers from our iterable. The reason is that 1 and 4 are both less than 5, but 6 is greater. So once takewhile sees the 6, the condition becomes False and it will ignore the rest of the items in the iterable.

tee(iterable, n=2)

The **tee** tool will create *n* iterators from a single iterable. What this means is that you can create multiple iterators from one iterable. Let's look at some explanatory code to how it works:

```
>>> from itertools import tee
>>> data = 'ABCDE'
>>> iter1, iter2 = tee(data)
>>> for item in iter1:
...     print(item)
...
A
B
C
D
E
>>> for item in iter2:
...     print(item)
...
A
B
C
D
E
```

Here we create a 5-letter string and pass it to **tee**. Because tee defaults to 2, we use multiple assignment to acquire the two iterators that are returned from tee. Finally we loop over each of the iterators and print out their contents. As you can see, their content are the same.

zip_longest(*iterables, fillvalue=None)

The **zip_longest** iterator can be used to zip two iterables together. If the iterables don't happen to be the same length, then you can also pass in a **fillvalue**. Let's look at a silly example based on the documentation for this function:

```
>>> from itertools import zip_longest
>>> for item in zip_longest('ABCD', 'xy', fillvalue='BLANK'):
...     print (item)
...
```

```
('A', 'x')
('B', 'y')
('C', 'BLANK')
('D', 'BLANK')
```

In this code we import zip_longest and then pass it two strings to zip together. You will note that the first string is 4-characters long while the second is only 2-characters in length. We also set a fill value of "BLANK". When we loop over this and print it out, you will see that we get tuples returned. The first two tuples are combinations of the first and second letters from each string respectively. The last two has our fill value inserted.

It should be noted that if the iterable(s) passed to zip_longest have the potential to be infinite, then you should wrap the function with something like islice to limit the number of calls.

The combinatoric generators

The itertools library contains four iterators that can be used for creating combinations and permutations of data. We will be covering these fun iterators in this section.

combinations(iterable, r)

If you have the need to create combinations, Python has you covered with **itertools.combinations**. What combinations allows you to do is create an iterator from an iterable that is some length long. Let's take a look:

```
>>>from itertools import combinations
>>>list(combinations('WXYZ', 2))
[('W', 'X'), ('W', 'Y'), ('W', 'Z'), ('X', 'Y'), ('X', 'Z'),
('Y', 'Z')]
```

When you run this, you will notice that combinations returns tuples. To make this output a bit more readable, let's loop over our iterator and join the tuples into a single string:

```
>>> for item in combinations('WXYZ', 2):
...     print(''.join(item))
...
WX
WY
WZ
XY
XZ
YZ
```

Now it's a little easier to see all the various combinations. Note that the combinations function does its combination in lexicographic sort order, so if you the iterable is sorted, then your combination

tuples will also be sorted. Also worth noting is that combinations will not produce repeat values in the combinations if all the input elements are unique.

combinations_with_replacement(iterable, r)

The **combinations_with_replacement** with iterator is very similar to **combinations**. The only difference is that it will actually create combinations where elements do repeat. Let's try an example from the previous section to illustrate:

```
>>> from itertools import combinations_with_replacement
>>> for item in combinations_with_replacement('WXYZ', 2):
...     print(''.join(item))
...
WW
WX
WY
WZ
XX
XY
XZ
YY
YZ
ZZ
```

As you can see, we now have four new items in our output: WW, XX, YY and ZZ.

product(*iterables, repeat=1)

The itertools package has a neat little function for creating Cartesian products from a series of input iterables. Yes, that function is **product**. Let's see how it works!

```
>>> from itertools import product
>>> arrays = [(-1,1), (-3,3), (-5,5)]
>>> cp = list(product(*arrays))
>>> cp
[(-1, -3, -5),
 (-1, -3, 5),
 (-1, 3, -5),
 (-1, 3, 5),
 (1, -3, -5),
 (1, -3, 5),
 (1, 3, -5),
 (1, 3, 5)]
```

Here we import product and then set up a list of tuples which we assign to the variable **arrays**. Next we call product with those arrays. You will notice that we call it using ***arrays**. This will cause the list to be "exploded" or applied to the product function in sequence. It means that you are passing in 3

arguments instead of one. If you want, try calling it with the asterisk pre-pended to arrays and see what happens.

permutations

The **permutations** sub-module of itertools will return successive *r* length permutations of elements from the iterable you give it. Much like the combinations function, permutations are emitted in lexicographic sort order. Let's take a look:

```
>>> from itertools import permutations
>>> for item in permutations('WXYZ', 2):
...     print(''.join(item))
...
WX
WY
WZ
XW
XY
XZ
YW
YX
YZ
ZW
ZX
ZY
```

You will notice that the output it quite a bit longer than the output from combinations. When you use permutations, it will go through all the permutatations of the string, but it won't do repeat values if the input elements are unique.

Wrapping Up

The itertools is a very versatile set of tools for creating iterators. You can use them to create your own iterators all by themselves or in combination with each other. The Python documentation has a lot of great examples that you can study to give you ideas of what can be done with this valuable library.

Related Reading

- Python documentation on itertools
- A gentle introduction to itertools
- A look at some of Python's useful itertools
- Python Module of the Week: itertools

4 Comments **Mouse Vs. the Python** **Ashish Patel**

Recommend 3 **Share** **Sort by Best**

Join the discussion…

starenka • a year ago

In the `starmap` example you can also use builtin add function (`operator.add`). Cheers for the article.

1 • Reply • Share ›

MaT • a year ago

Is the example for islice really correct? Because title says:
islice(iterable, start, stop[, step])

And then you have:

iterator = islice('123456', 4)

I thought that the position to stop is the third argument, not the second one...

• Reply • Share ›

Mike Driscoll Mod → MaT • a year ago

Actually islice is overloaded, which you don't see much in Python. Take a look at the documentation here: https://docs.python.org/3/l.... Basically what's happening is that if you pass in just two arguments, an iterable and an int, then Python knows that the second argument is the stop argument. If you pass in more then two, then Python uses the second version of islice and knows that you're setting the start AND stop (and optionally step).

But I forgot to mention that in the article, so I just fixed that. Thanks for pointing that out.

• Reply • Share ›

Vasudev Ram • a year ago

Nice post.

I had written a function some time earlier, that can flatten a nested list, in a different way:

http://code.activestate.com...

Some readers also commented with solutions of their own, some using Python 3's 'yield from'.

http://jugad2.blogspot.in/2...

• Reply • Share ›