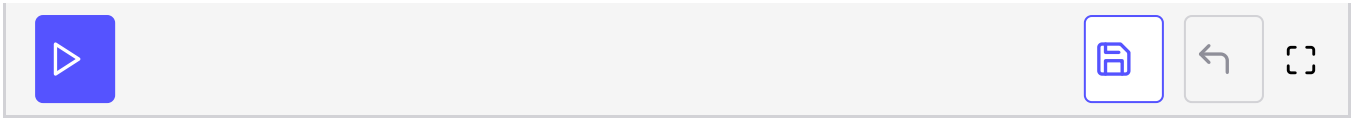
# Iterators That Terminate

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)

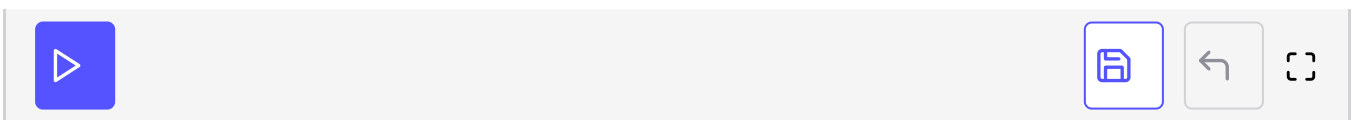
The **accumulate** iterator will return accumulated sums or the accumulated results of a two argument function that you can pass to **accumulate**. The default of accumulate is addition, so let's give that a quick try:

```
from itertools import accumulate
print (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:

```
from itertools import accumulate
import operator
print (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 (1x1=1, 1x2=2, 2x3=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 append the items in each list to the original list instead of creating a list of lists. 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]]

#Traceback (most recent call last):
# File "/usercode/__ed_file.py", line 4, in <module>
# my_list.extend(cmd, numbers)
#TypeError: extend() takes exactly one argument (2 given)
```

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**:

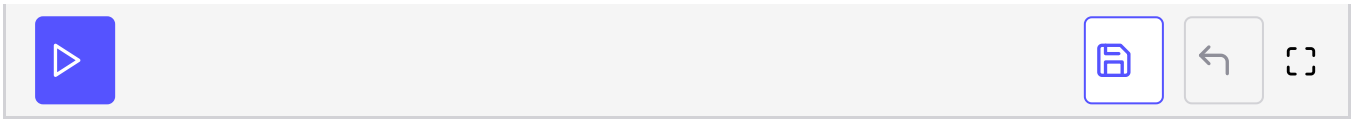
```
from itertools import chain
my_list = ['foo', 'bar']
numbers = list(range(5))
cmd = ['ls', '/some/dir']
my_list = list(chain(['foo', 'bar'], cmd, numbers))

print (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
print (my_list)
#['foo', 'bar', 'ls', '/some/dir', 0, 1, 2, 3, 4]
```

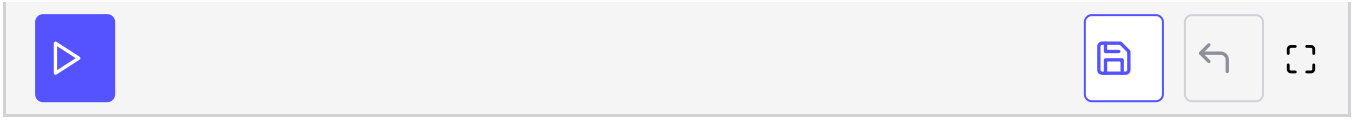


Both of these methods are certainly valid and before I knew about **chain** I would have probably gone this route, but I think chain is a more elegant and easier to understand solution in this particular case.

## chain.from_iterable(iterable)

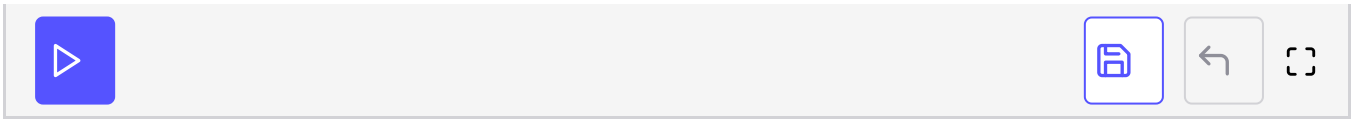
You can also use a method of **chain** called **from_iterable**. This method works slightly differently then using chain directly. Instead of passing in a series of iterables, you have to pass in a nested list. Let's take a look:

```
from itertools import chain
numbers = list(range(5))
cmd = ['ls', '/some/dir']
print (chain.from_iterable(cmd, numbers))
#Traceback (most recent call last):
#  File "/usercode/__ed_file.py", line 4, in <module>
# print (chain.from_iterable(cmd, numbers))
#TypeError: from_iterable() takes exactly one argument (2 given)
```



```
from itertools import chain
numbers = list(range(5))
cmd = ['ls', '/some/dir']

print (list(chain.from_iterable([cmd, numbers])))
#['ls', '/some/dir', 0, 1, 2, 3, 4]
```

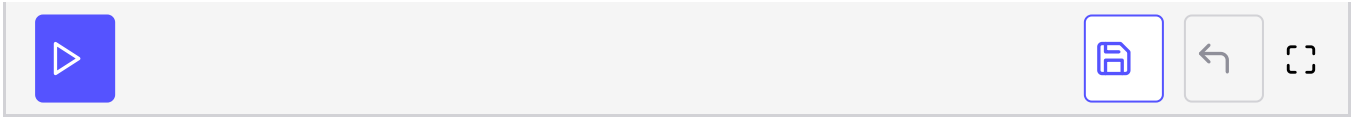


Here we import chain as we did previously. We try passing in our two lists but we end up getting a **TypeError**! To fix this, we change our call slightly such that we put **cmd** and **numbers** inside a **list** and then pass that nested list to **from_iterable**. It's a subtle difference but still 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]
print (list(compress(letters, bools)))
#['A', 'C', 'D']
```



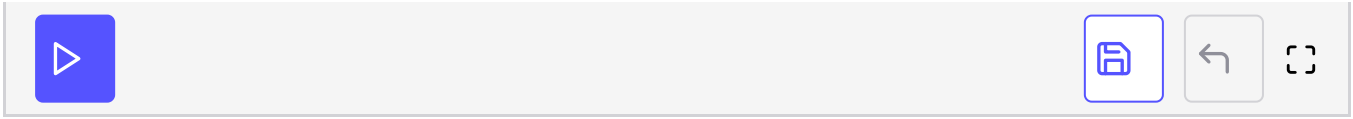
In this example, we have a group of seven letters and a list of five Bools. Then we pass them into the compress function. the compress function will go through each respective iterable and check the first against the second. If the second has a matching True, then it will be kept. If it's a False, then 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
print (list(dropwhile(lambda x: x<5, [1,4,6,4,1])))
#[6, 4, 1]
```



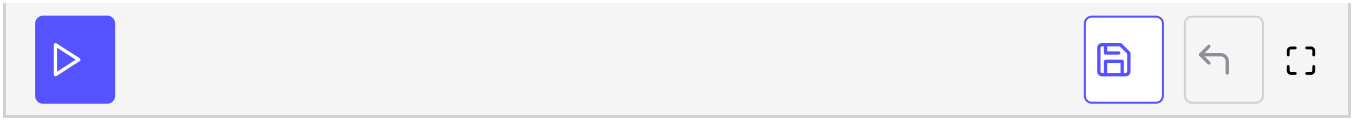
Here we import **dropwhile** and then we pass it a simple **lambda** statement. This function will return True if **x** is less than 5. Other it will return False. The

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

print (list(dropwhile(greater_than_five, [6, 7, 8, 9, 1, 2, 3, 10])))
#[1, 2, 3, 10]
```



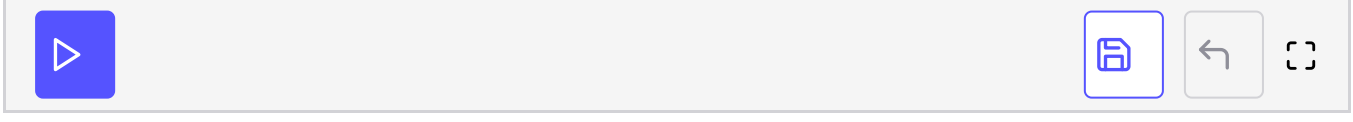
Here we create a simple function in Python's interpreter. This function is our predicate or filter. If the values we pass to it are True, then they will get dropped. Once we hit a value that is less than 5, then ALL the values after and including that value will be kept, which you can see in the example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

print (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)

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)
print (next(iterator))
#'1'

print (next(iterator))
#'2'

print (next(iterator))
#'3'

print (next(iterator))
#'4'

print (next(iterator))
#Traceback (most recent call last):
#  File "/usercode/__ed_file.py", line 15, in <module>
# print (next(iterator))
#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 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
print (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:

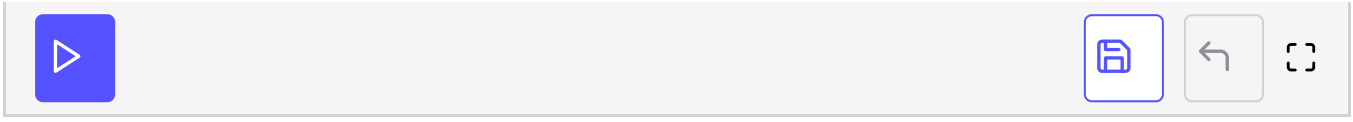
```
from itertools import tee
data = 'ABCDE'
iter1, iter2 = tee(data)
for item in iter1:
    print(item)

#A
#B
#C
#D
#E

for item in iter2:
    print(item)
```

```
#A
#B
#C

#D
#E
```



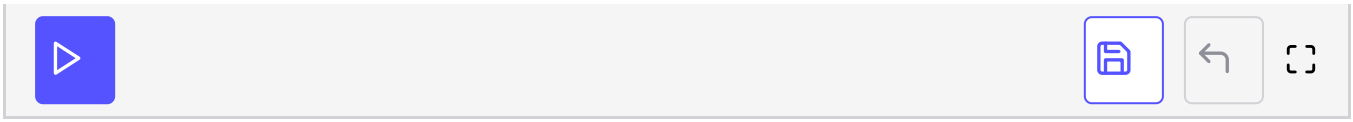
Here we create a 5-letter string and pass it to **tee**. Because tee defaults to 2, we use multiple assignment to acquire the two iterators that are returned from tee. Finally we loop over each of the iterators and print out their contents. As you can see, their content are the same.

## 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.