Clases Abstractas

(`abc`)

Fuente: https://zaiste.net/2013/01/abstract_classes_in_python/

Abstract Classes in Python

python

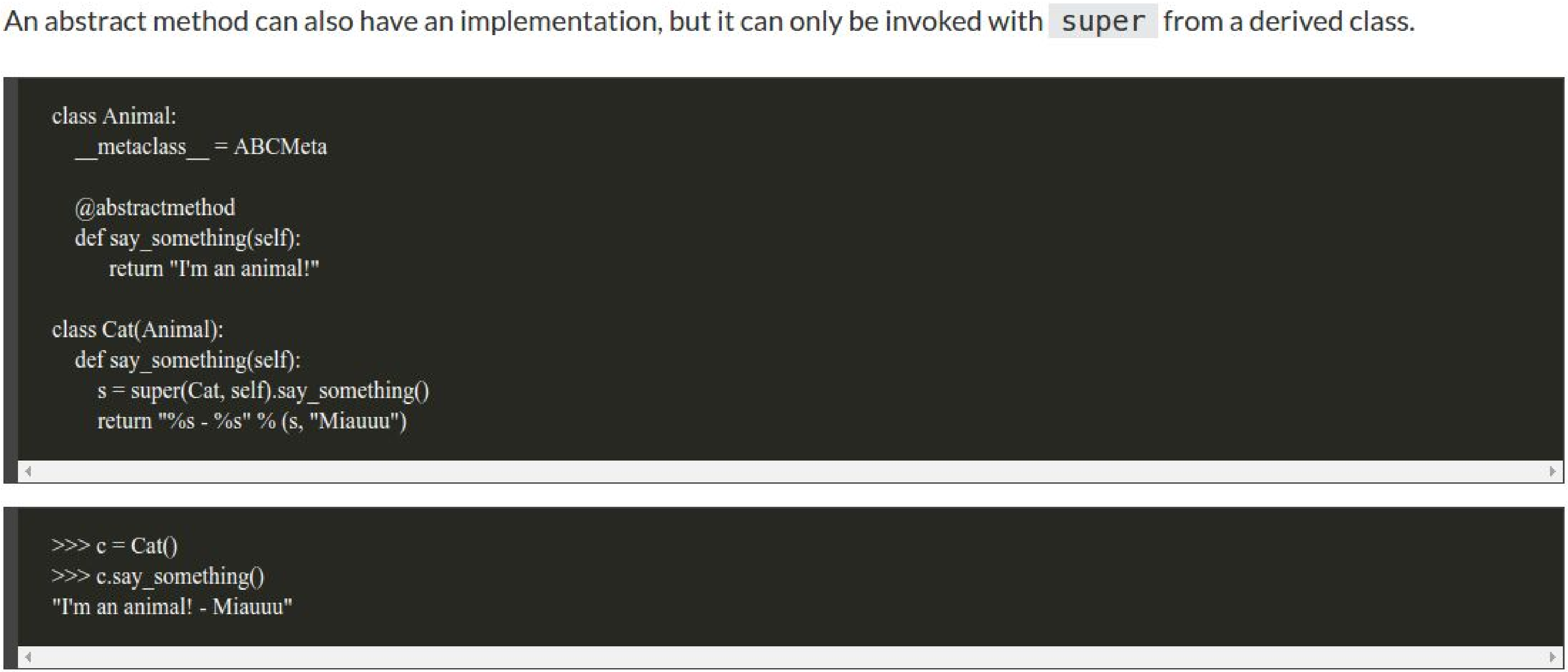
Before Python 2.6 there was no explicit way to declare an abstract class. It changed with the `abc` (Abstract Base Class) module from the standard library.

abc module

`abc` module allows to enforce that a derived class implements a particular method using a special `@abstractmethod` decorator on that method.

```
from abc import ABCMeta, abstractmethod

class Animal:
    __metaclass__ = ABCMeta

    @abstractmethod
    def say_something(self): pass

class Cat(Animal):
    def say_something(self):
        return "Miauuu!"
```

```
>>> a = Animal()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Animal with abstract methods say_something
```



An abstract method can also have an implementation, but it can only be invoked with `super` from a derived class.

```
class Animal:
    __metaclass__ = ABCMeta

    @abstractmethod
    def say_something(self):
        return "I'm an animal!"

class Cat(Animal):
    def say_something(self):
        s = super(Cat, self).say_something()
        return "%s - %s" % (s, "Miauuu")
```

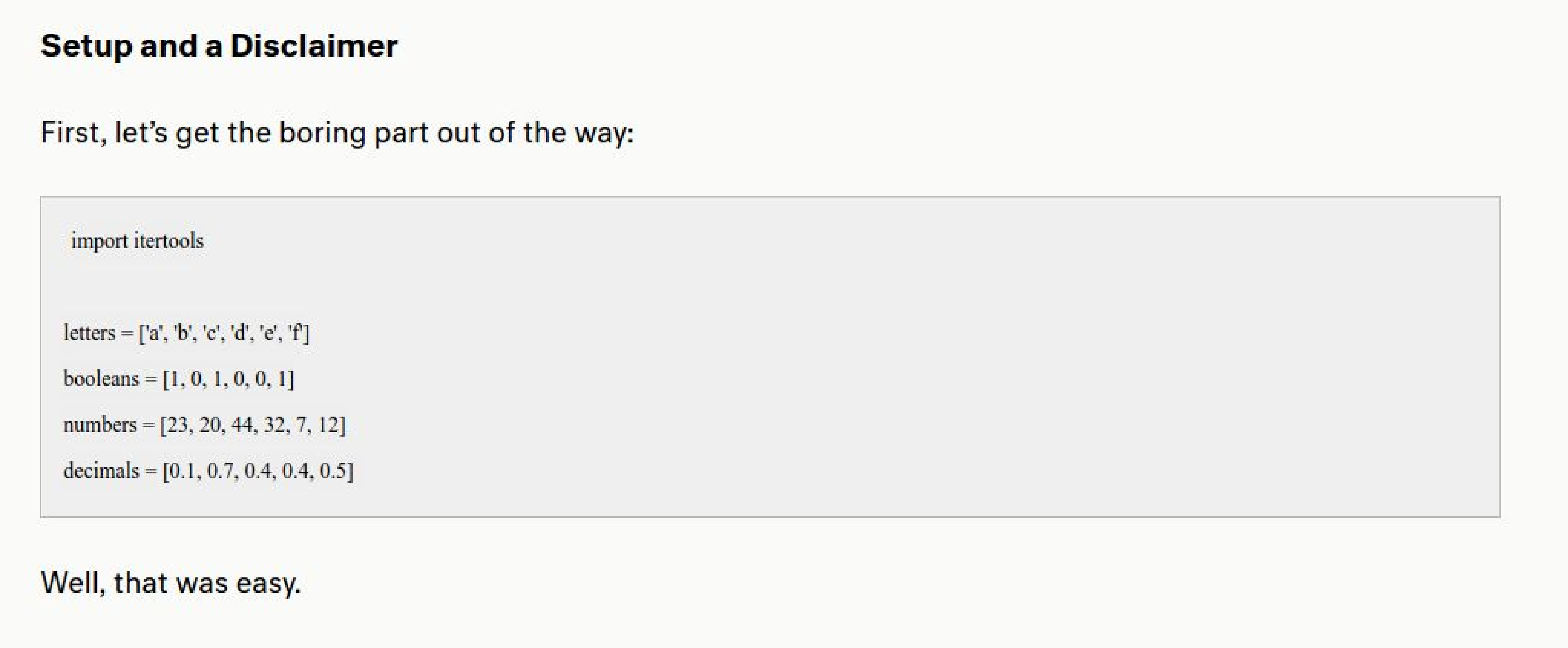
```
>>> c = Cat()
>>> c.say_something()
"I'm an animal! - Miauuu"
```

There is more feautres provided by `abc` module, but they are less common in use than these described in this post. For details check the documentation.

Justin
Duke

A gentle introduction to itertools

http://jmduke.com/posts/a-gentle-introduction-to-itertools/

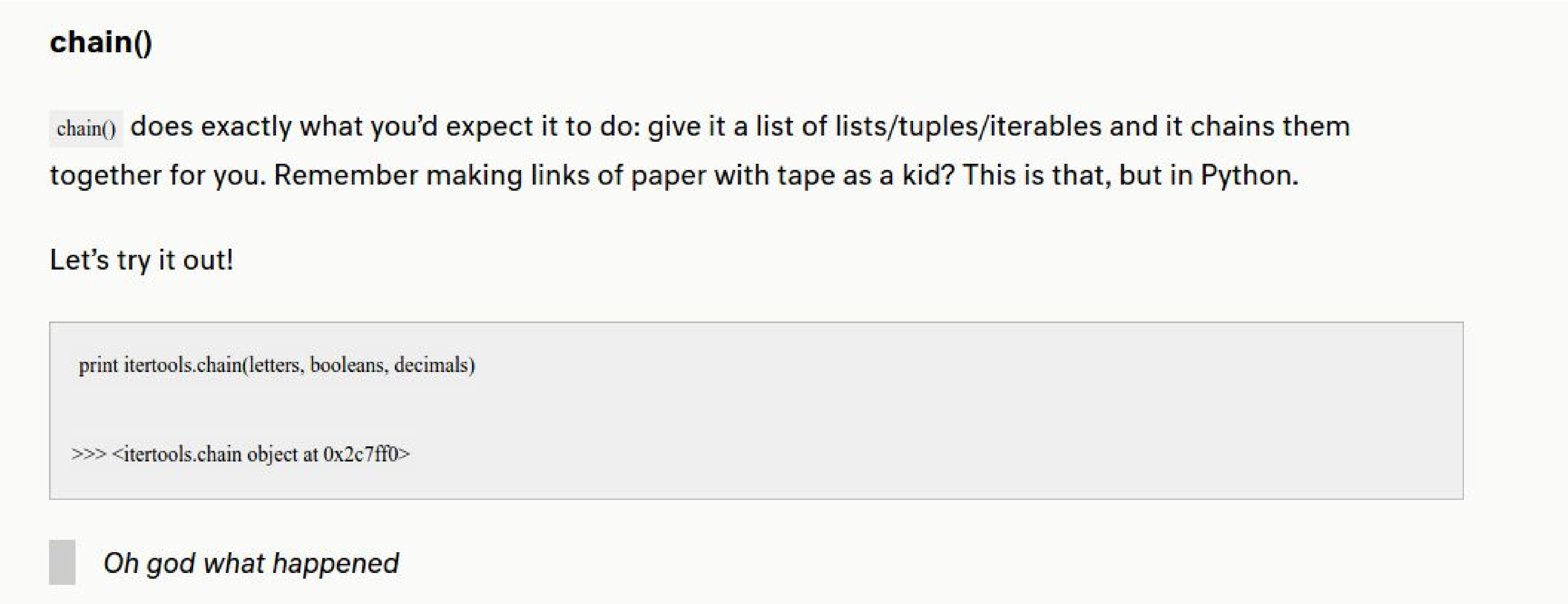


Setup and a Disclaimer

First, let’s get the boring part out of the way:

```
import itertools


letters = ['a', 'b', 'c', 'd', 'e', 'f']

booleans = [1, 0, 1, 0, 0, 1]

numbers = [23, 20, 44, 32, 7, 12]

decimals = [0.1, 0.7, 0.4, 0.4, 0.5]
```

Well, that was easy.



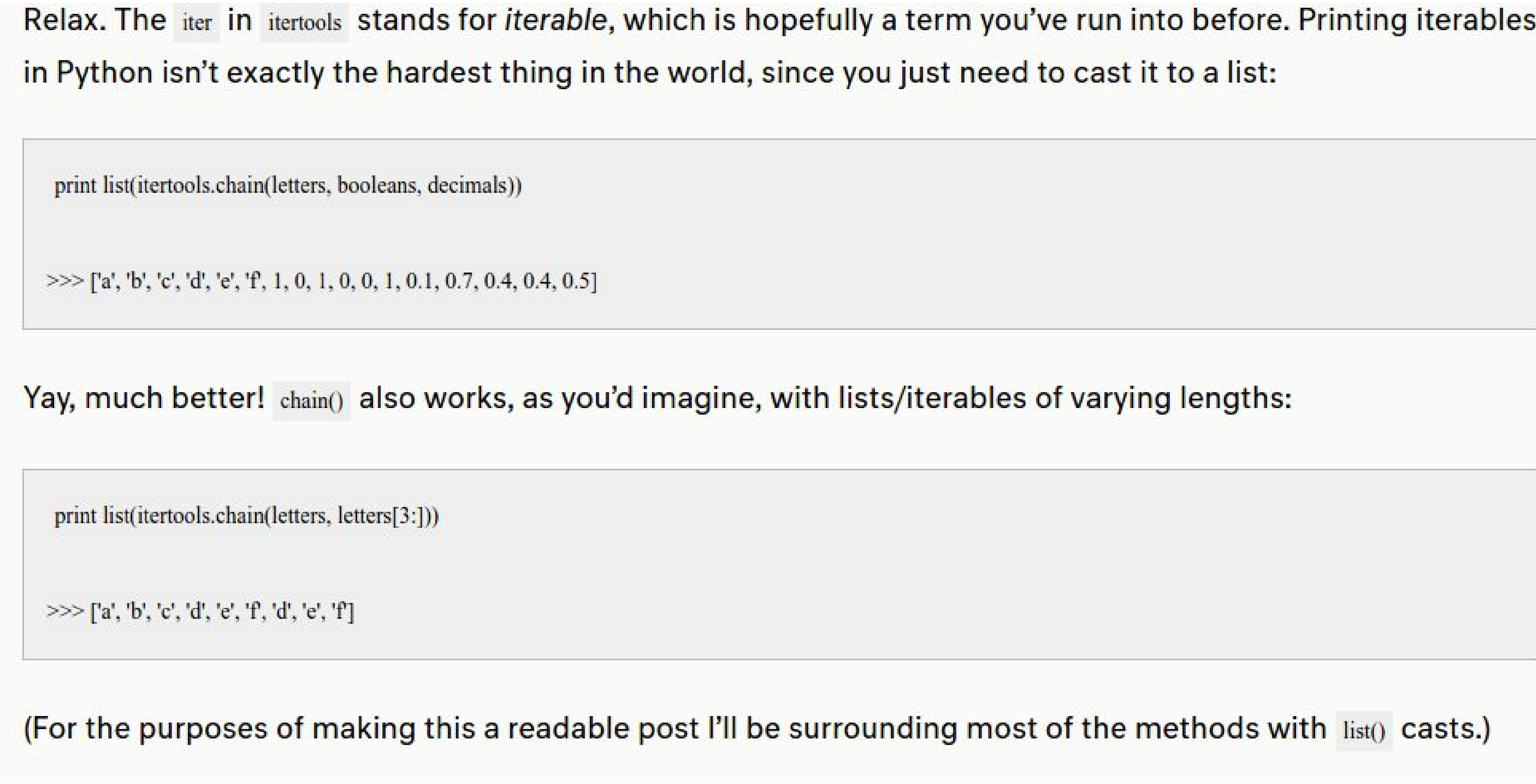
chain()

`chain()` does exactly what you'd expect it to do: give it a list of lists/tuples/iterables and it chains them together for you. Remember making links of paper with tape as a kid? This is that, but in Python.

Let's try it out!

```
 print itertools.chain(letters, booleans, decimals)


>>> <itertools.chain object at 0x2c7ff0>
```

> *Oh god what happened*



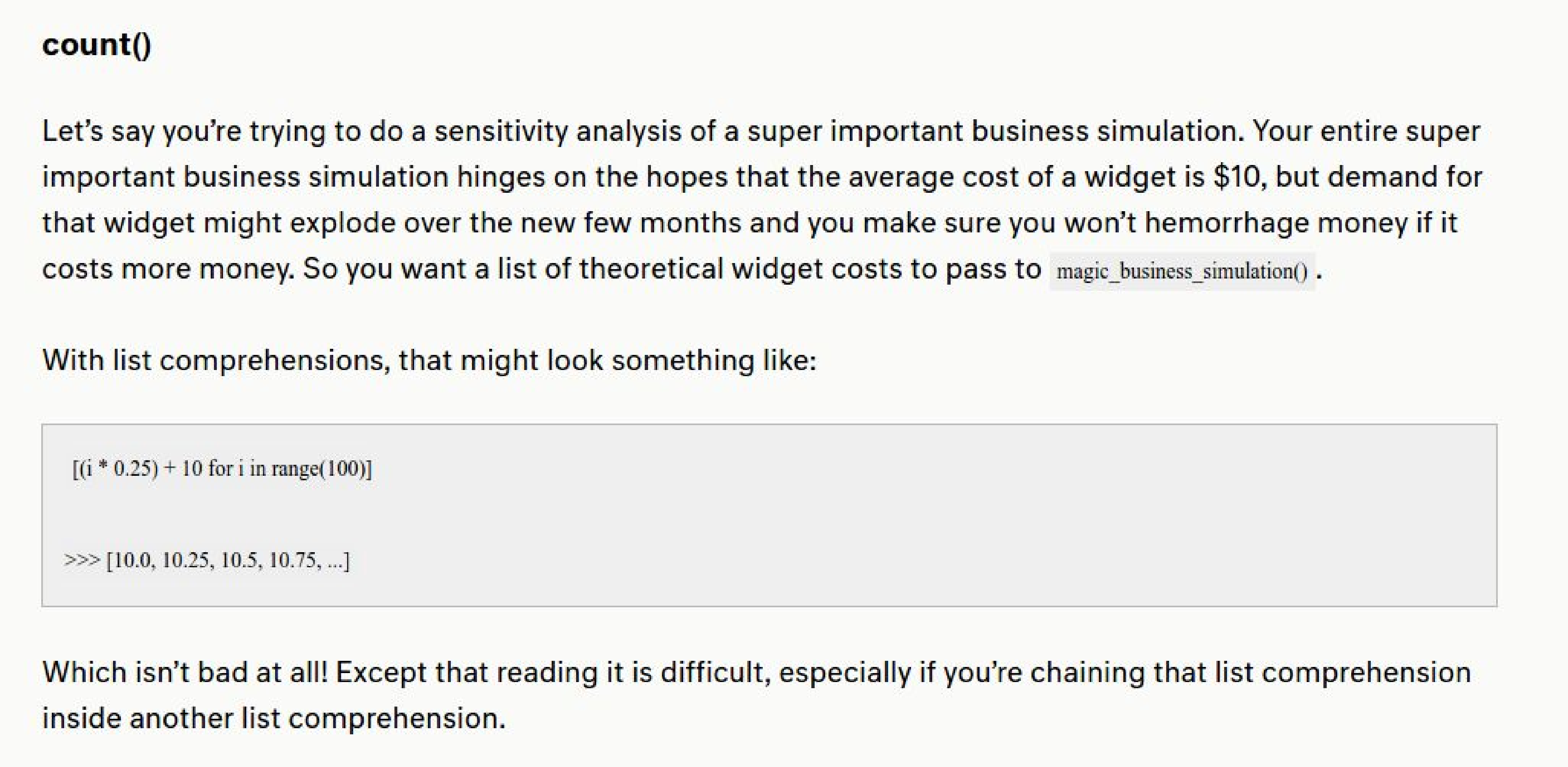
Relax. The `iter` in `itertools` stands for *iterable*, which is hopefully a term you've run into before. Printing iterables in Python isn't exactly the hardest thing in the world, since you just need to cast it to a list:

```
print list(itertools.chain(letters, booleans, decimals))


>>> ['a', 'b', 'c', 'd', 'e', 'f', 1, 0, 1, 0, 0, 1, 0.1, 0.7, 0.4, 0.4, 0.5]
```

Yay, much better! `chain()` also works, as you'd imagine, with lists/iterables of varying lengths:

```
print list(itertools.chain(letters, letters[3:]))


>>> ['a', 'b', 'c', 'd', 'e', 'f', 'd', 'e', 'f']
```

(For the purposes of making this a readable post I'll be surrounding most of the methods with `list()` casts.)



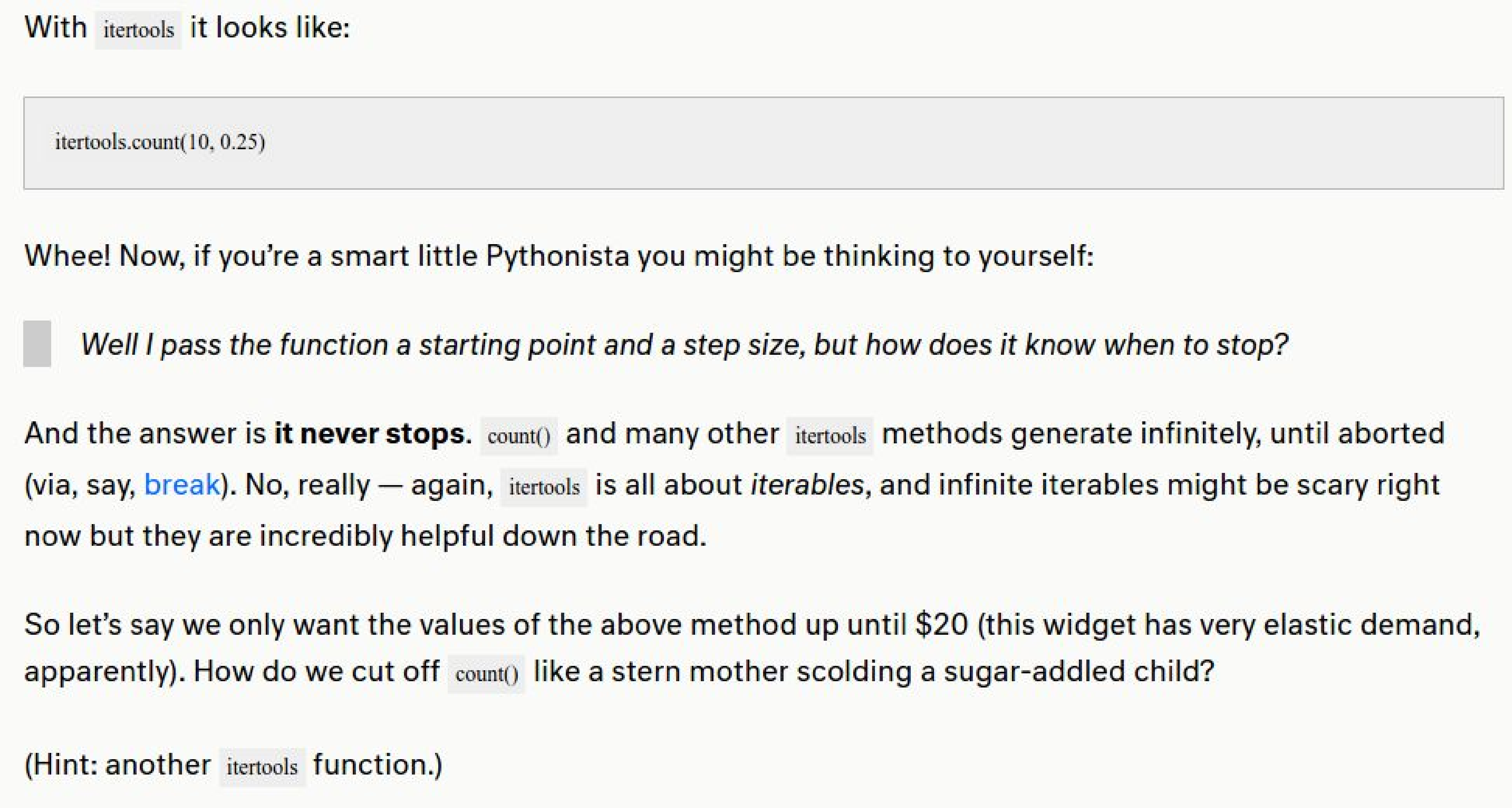
count()

Let’s say you’re trying to do a sensitivity analysis of a super important business simulation. Your entire super important business simulation hinges on the hopes that the average cost of a widget is $10, but demand for that widget might explode over the new few months and you make sure you won’t hemorrhage money if it costs more money. So you want a list of theoretical widget costs to pass to `magic_business_simulation()`.

With list comprehensions, that might look something like:

```
 [(i * 0.25) + 10 for i in range(100)]


>>> [10.0, 10.25, 10.5, 10.75, ...]
```

Which isn’t bad at all! Except that reading it is difficult, especially if you’re chaining that list comprehension inside another list comprehension.



With `itertools` it looks like:

```
itertools.count(10, 0.25)
```

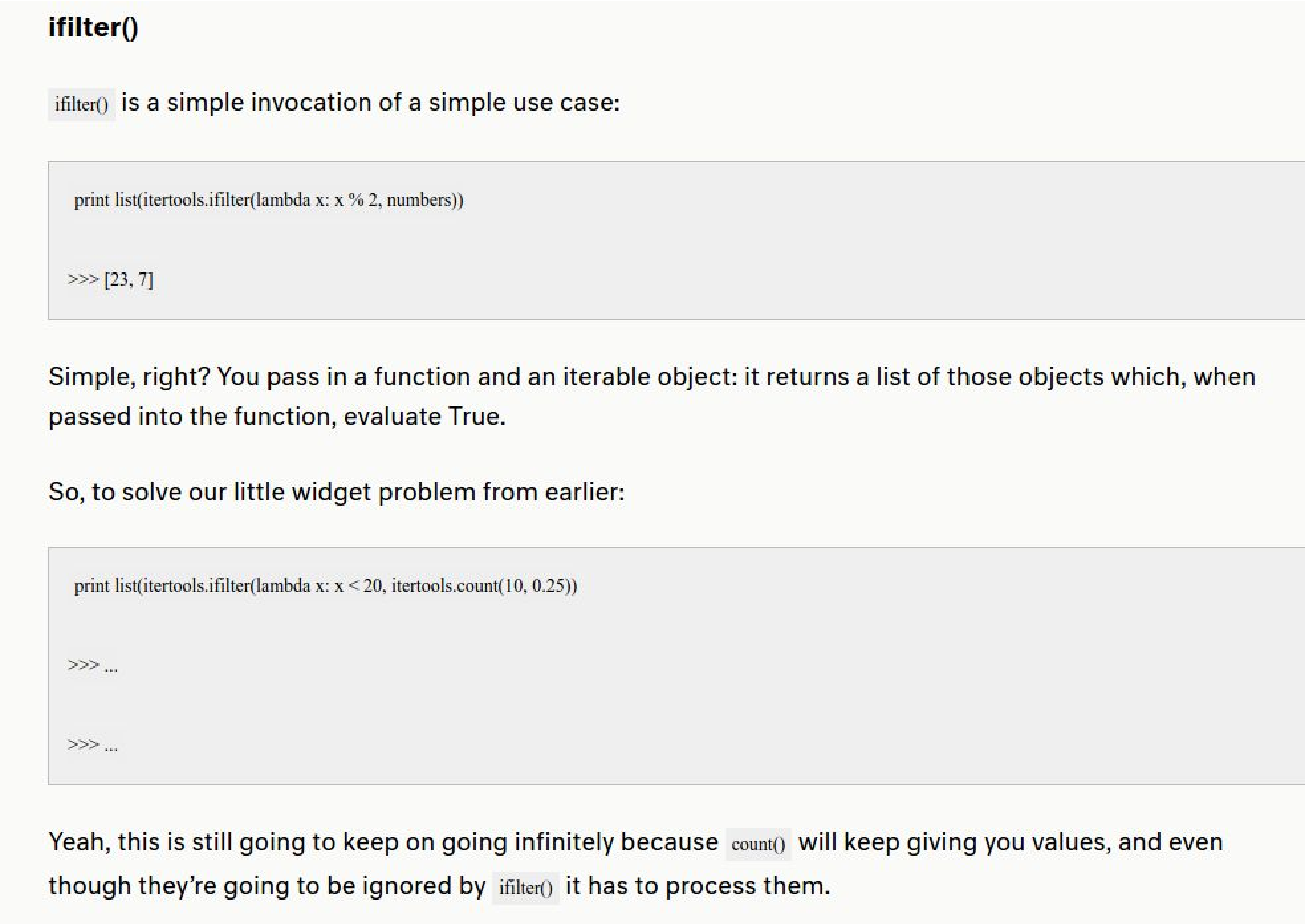
Whee! Now, if you’re a smart little Pythonista you might be thinking to yourself:

> *Well I pass the function a starting point and a step size, but how does it know when to stop?*

And the answer is **it never stops.** `count()` and many other `itertools` methods generate infinitely, until aborted (via, say, break). No, really — again, `itertools` is all about *iterables*, and infinite iterables might be scary right now but they are incredibly helpful down the road.

So let’s say we only want the values of the above method up until $20 (this widget has very elastic demand, apparently). How do we cut off `count()` like a stern mother scolding a sugar-addled child?

(Hint: another `itertools` function.)



ifilter()

`ifilter()` is a simple invocation of a simple use case:

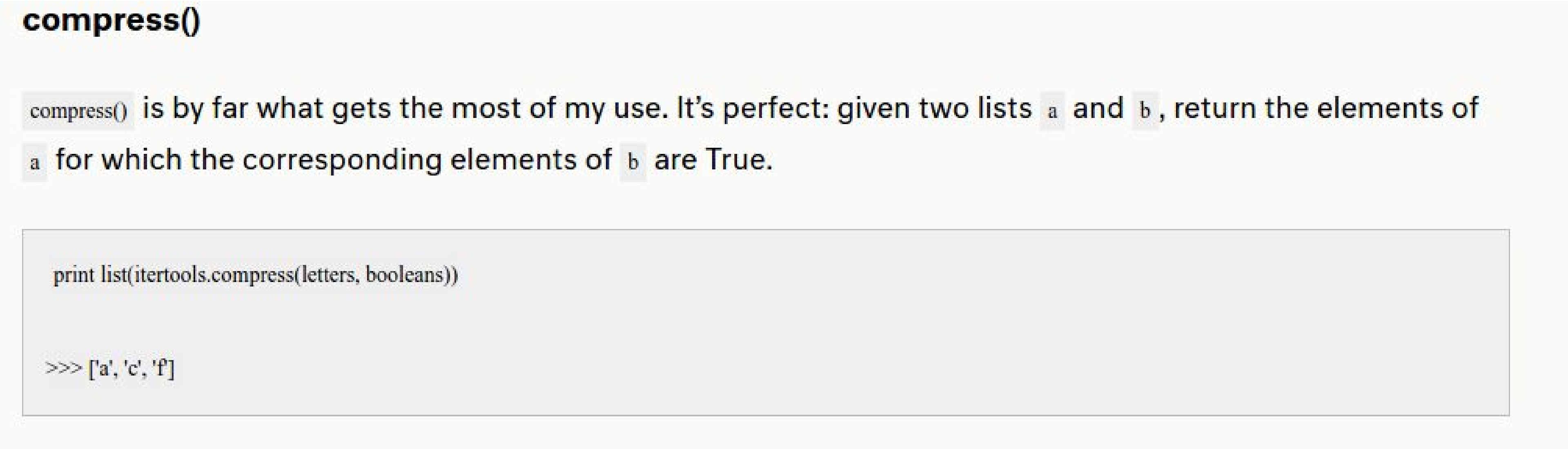
```
print list(itertools.ifilter(lambda x: x % 2, numbers))

>>> [23, 7]
```

Simple, right? You pass in a function and an iterable object: it returns a list of those objects which, when passed into the function, evaluate True.

So, to solve our little widget problem from earlier:

```
print list(itertools.ifilter(lambda x: x < 20, itertools.count(10, 0.25))

>>> ...

>>> ...
```

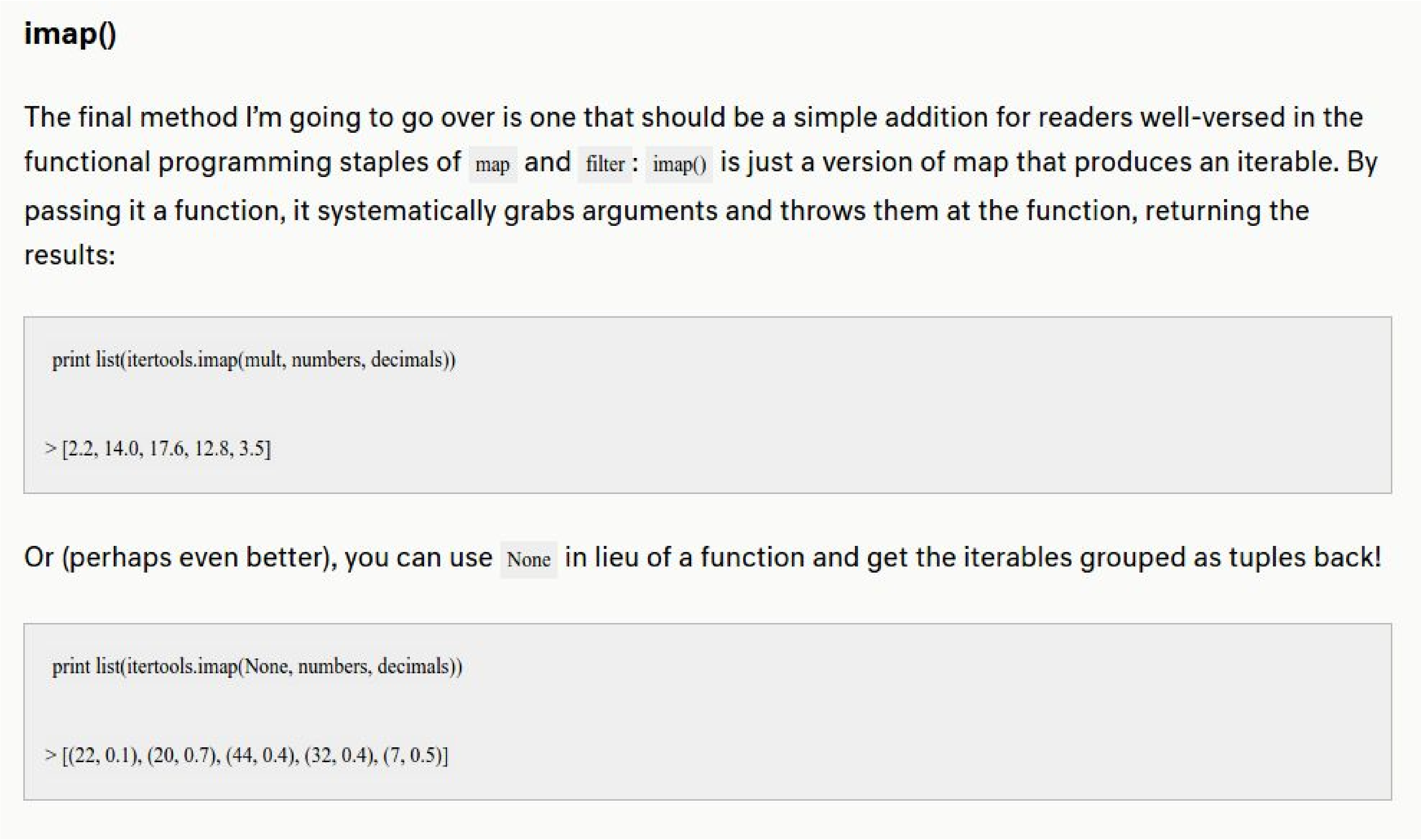
Yeah, this is still going to keep on going infinitely because `count()` will keep giving you values, and even though they're going to be ignored by `ifilter()` it has to process them.



compress()

`compress()` is by far what gets the most of my use. It’s perfect: given two lists `a` and `b`, return the elements of `a` for which the corresponding elements of `b` are True.

```
  print list(itertools.compress(letters, booleans))


>>> ['a', 'c', 'f']
```



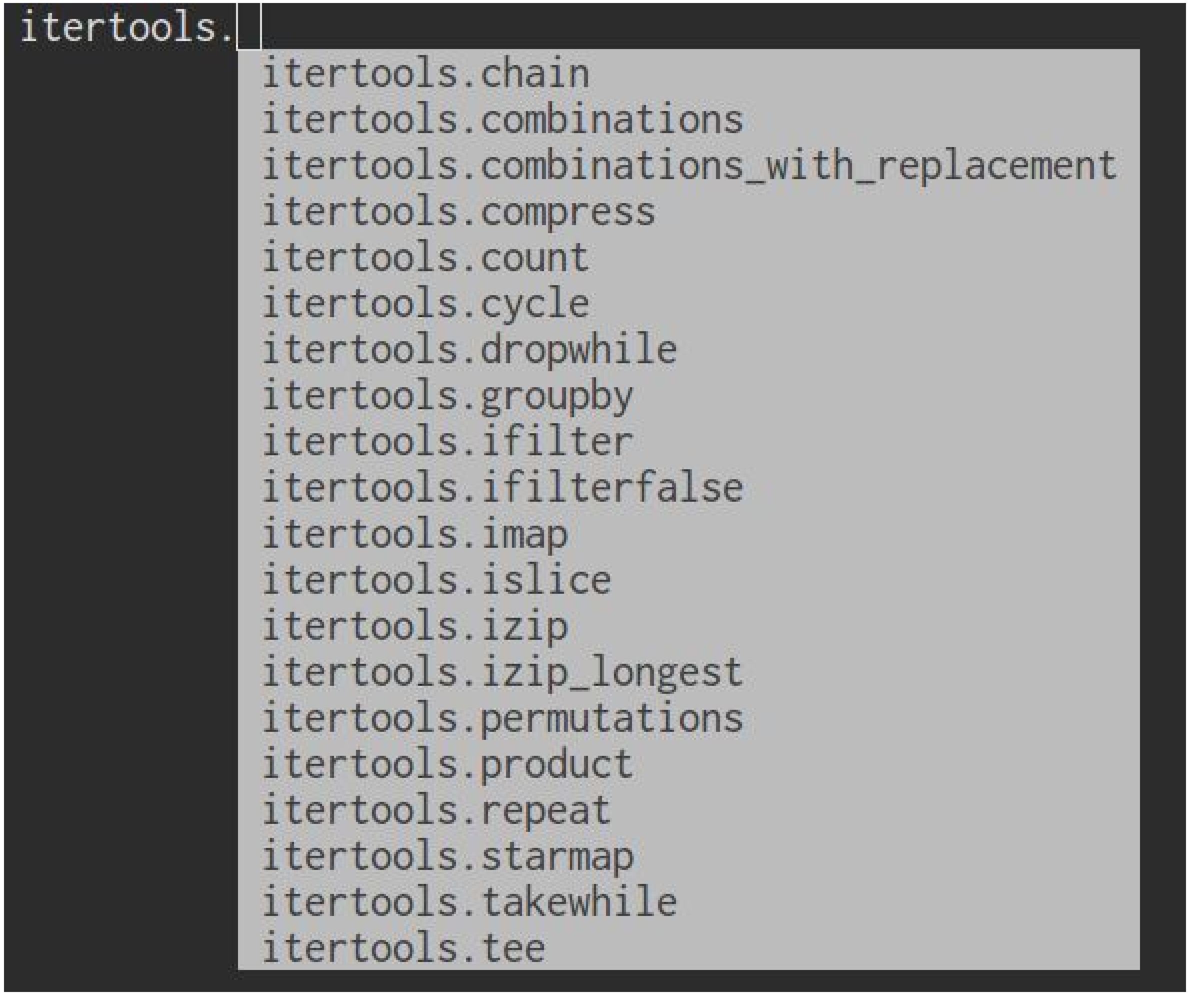
imap()

The final method I'm going to go over is one that should be a simple addition for readers well-versed in the functional programming staples of `map` and `filter` : `imap()` is just a version of map that produces an iterable. By passing it a function, it systematically grabs arguments and throws them at the function, returning the results:

```
 print list(itertools.imap(mult, numbers, decimals))


> [2.2, 14.0, 17.6, 12.8, 3.5]
```

Or (perhaps even better), you can use `None` in lieu of a function and get the iterables grouped as tuples back!

```
 print list(itertools.imap(None, numbers, decimals))


> [(22, 0.1), (20, 0.7), (44, 0.4), (32, 0.4), (7, 0.5)]
```



```
itertools.
itertools.chain
itertools.combinations
itertools.combinations_with_replacement
itertools.compress
itertools.count
itertools.cycle
itertools.dropwhile
itertools.groupby
itertools.ifilter
itertools.ifilterfalse
itertools.imap
itertools.islice
itertools.izip
itertools.izip_longest
itertools.permutations
itertools.product
itertools.repeat
itertools.starmap
itertools.takewhile
itertools.tee
```