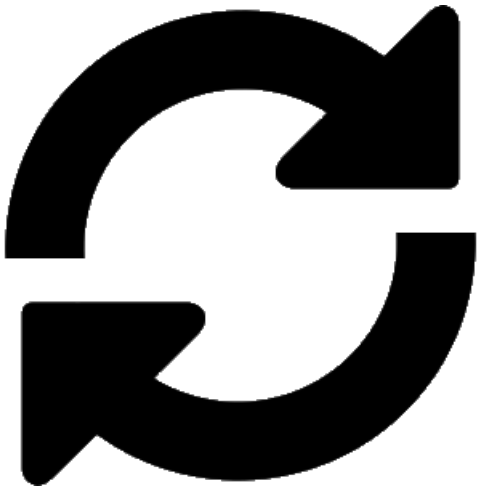
adafruit learning system

Itertools for CircuitPython

Created by Dave Astels



Last updated on 2019-04-04 06:56:12 PM UTC

Overview

Iteration

What is iteration? Doing the same thing repeatedly, generally with a different, but related, piece of data. Programmers use iteration all the time, often even the ones who are using recursion (https://adafru.it/Emr).

Every time you are using `for` to process items in a list, you're using iteration. Every time you use a list comprehension, you're using iteration.

It's so common that Python has some builtin constructs to make it easier: iterators, iterables, and generators. These were covered in another guide (https://adafru.it/Ems). As glimpsed in that guide, Python has a module that provides a wealth of tools for working with these: **itertools**.

In that guide we briefly looked at the small subset of itertools that has been ported to MicroPython (https://adafru.it/Emt).

This guide describes a fuller port that has been done to CircuitPython. Additionally, the itertools recipes functions (https://adafru.it/Czv) have been largely made available as well.

Two modules are available as helpers in the CircuitPython Bundle:

adafruit_itertools - a port of the majority of Python's itertools functions.

adafruit_itertools_extras - a port of many of the itertools recipes functions. These are very useful function that are built on top of the core itertools module.

https://adafru.it/zB-

https://adafru.it/zB-

Why itertools?

Similar to how the math module provides functions that operate on numbers, itertools gives you functions that operate on iterators. This lets you code in a more *functional** or *dataflow-centric* way, also sometimes known as *stream-based* programming..

The Python3/CPython version is implemented in C. The CircuitPython implementation is a pure Python implementation so it's performance isn't nearly as good, but should be adequate in most, if not all, situations. It has the additional benefit of being readily studied and understood.

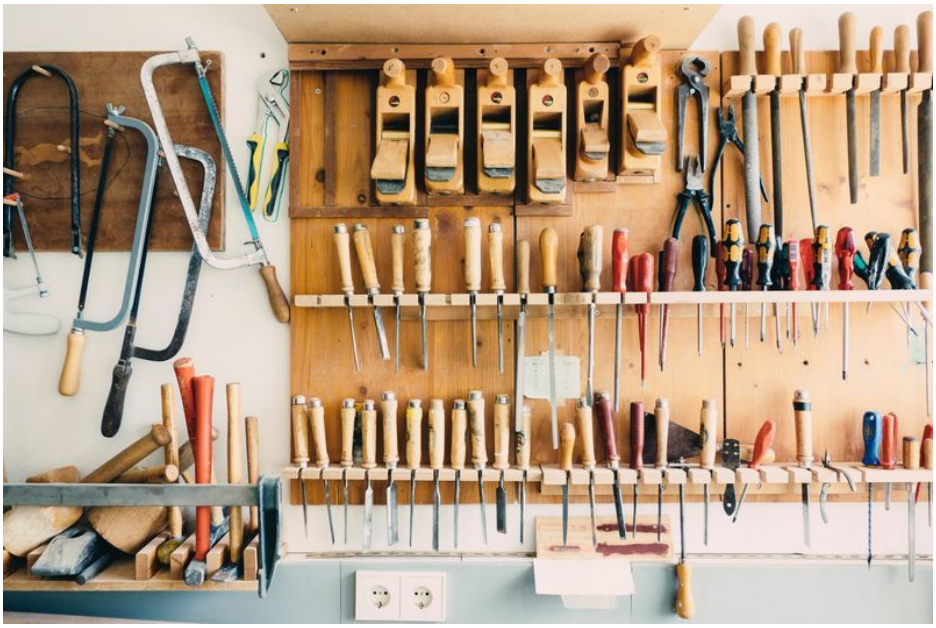
One of the major advantage of using iterators is that they can be used to compute one value at a time rather than all the values at once. This has several implications:

1. Computing time is spread out as required rather than all at once to populate (for example) a list of values.
2. Since elements are computed as needed, memory to store the entire list is not required. This means you can process more data with less memory. This is very handle in a memory-limited environment.
3. Since time and space is only required for one element at a time, we can deal with sequences of any length, even infinite, without any additional requirements.

*Functional as in *functional programming*, not as in *it works*.

Header image icon made by Designerz Base (https://adafru.it/Emu) from www.flaticon.com (https://adafru.it/Emv)

Itertools



The **adafruit_itertools** module contains the main iterator building block functions. This provides you with everything you should need to take full advantage of iterators in your code. This section enumerates the functions present in the module, describing each and providing examples. The `list` function is used to pull out the values in cases where an iterator is returned. In cases where the returned iterator is infinite, `take` (from **adafruit_itertools_extras**) is used to pull out a fixed number of values. `take` returns those values in a list.

accumulate(iterable, func=lambda x, y: x + y)

Make an iterator that generates accumulated sums, or accumulated results of other binary functions (specified via the optional `func` argument). If `func` is supplied, it should be a function of two arguments that returns a value of the same type. Elements of `iterable` may be any type that can be accepted as arguments to `func`. (For example, with the default operation of addition, elements may be any addable type.) If `iterable` is empty, no values will be generated.

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

```
>>> accumulate(range(1, 10), lambda x, y: x * y))
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
```

chain(*iterables)

Make an iterator that generates elements from the first `iterable` until it is exhausted, then proceeds to the next `iterable`, until all of them are exhausted. Used for treating consecutive sequences as a single sequence.

```
>>> list(chain('ABC', 'DEF'))
['A', 'B', 'C', 'D', 'E', 'F']
```

chain_from_iterable(iterables)

An alternate approach to `chain()`. Iterables to be chanined are generated from a single iterable argument.

So, for example, instead of passing in multiple strings as you would with `chain()`, you can pass in a list of strings.

```
>>> list(chain_from_iterable(['ABC', 'DEF']))
['A', 'B', 'C', 'D', 'E', 'F']
```

combinations(iterable, r)

Generate `r` length subsequences of elements from `iterable`. Combinations are generated in lexicographic sort order (the order they appear in `iterable`). So, if the `iterable` is sorted, the combination tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. So if the input elements are unique, there will be no repeat values in each combination.

```
>>> list(combinations('ABCD', 2))
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')]
```

```
>>> list(combinations(range(4), 3))
[(0, 1, 2), (0, 1, 3), (0, 2, 3), (1, 2, 3)]
```

combinations_with_replacement(iterable, r)

Generate `r` length subsequences of elements from `iterable` allowing individual elements to be repeated more than once.

Combinations are generated in lexicographic sort order. So, if `iterable` is sorted, the combination tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. So if the input elements are unique, the generated combinations will also be unique.

```
>>> list(combinations_with_replacement('ABCD', 2))
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'B'), ('B', 'C'), ('B', 'D'), ('C', 'C'), ('C', 'D'), ('D', 'D')]
```

```
>>> list(combinations_with_replacement(range(4), 3))
[(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 0, 3), (0, 1, 1), (0, 1, 2), (0, 1, 3), (0, 2, 2), (0, 2, 3), (0, 3, 3), (1, 1, 1), (1, 1, 2), (1, 1, 3), (1, 2, 2), (1, 2, 3), (1, 3, 3), (2, 2, 2), (2, 2, 3), (2, 3, 3), (3, 3, 3)]
```

compress(data, selectors)

Make an iterator that filters elements from `data` returning only those that have a corresponding element in `selectors` that evaluates to `True`. Stops when either the `data` or `selectors` iterables has been exhausted.

```
>>> list(compress('ABCDEF', [1,0,1,0,1,1]))
['A', 'C', 'E', 'F']
```

count(start=0, step=1)

Make an infinite iterator that returns evenly spaced values starting with number `start`. The spacing between values is set by `step`. Often used as an argument to `map()` to generate consecutive data values. Also, used with `zip()` to add sequence numbers.

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

```
>>> take(5, count(3))
[3, 4, 5, 6, 7]
>>> take(5, count(1, 2))
[1, 3, 5, 7, 9]
```

cycle(iterable)

Make an iterator returning elements from the `iterable` and saving a copy of each. When the iterable is exhausted, return elements from the saved copy. Repeats indefinitely.

```
>>> take(10, cycle("ABCD"))
['A', 'B', 'C', 'D', 'A', 'B', 'C', 'D', 'A', 'B']
```

dropwhile(predicate, iterable)

Make an iterator that drops elements from `iterable` as long as `predicate` is true; afterwards, generates every element. Note, the iterator does not produce any output until `predicate` first becomes false, so it may have a lengthy start-up time.

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

filterfalse(predicate, iterable)

Make an iterator that filters elements from `iterable` generating only those for which `predicate` is `False`. If `predicate` is `None`, return the items that are logically false, i.e. `bool(x)` evaluates to `False`.

```
>>> list(filterfalse(lambda x: x%2, range(10)))
[0, 2, 4, 6, 8]
```

groupby(iterable, key_func=None)

Make an iterator that generates consecutive keys and groups from `iterable`. `key_func` is a function computing a key value for each element. If not specified or is `None`, `key_func` defaults to an identity function that generates the element unchanged. Generally, it is desirable that `iterable` is already sorted on the same key function.

The operation of `groupby()` is similar to the **uniq** filter in Unix. It generates a break or new group every time the value of the key function changes (which is why it is usually necessary to have sorted the data using the same key function).

The returned group is itself an iterator that shares the underlying iterable with `groupby()`. Because the source is shared, when the `groupby()` object is advanced, the previous group is no longer visible. So, if that data is needed later, it should be stored as a list, like so:

```
groups = []
uniquekeys = []
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g)) # Store group iterator as a list
    uniquekeys.append(k)
```

```
>>> [k for k, g in groupby('AAAABBBCCDAABBB')]
['A', 'B', 'C', 'D', 'A', 'B']
>>> [list(g) for k, g in groupby('AAAABBBCCD')]
[['A', 'A', 'A', 'A'], ['B', 'B', 'B'], ['C', 'C'], ['D']]
```

Let's do something a bit more useful. Say we have a list of numbers and we want to divide them up by some criteria. For simplicity let's go with odd/even. We can write a classifier function to do this:

```
>>> def even_odd(x):
...     if x % 2 == 0:
...         return 'even'
...     else:
...         return 'odd'

>>> even_odd(2)
'even'
>>> even_odd(5)
'odd'
```

Next, let's create some random integers:

```
numbers = list(repeatfunc(lambda: random.randint(0, 100), 25)))
>>> numbers
[12, 17, 0, 82, 37, 34, 3, 41, 53, 60, 62, 35, 27, 75, 43, 31, 98, 56, 97, 26, 73, 43, 62, 74, 72 ]
```

We can go ahead and group those using our even/odd function:

```
>>> for k, g in groupby(numbers, even_odd):
...     print('{0}: {1}'.format(k, list(g)))
...
even: [12]
odd: [17]
even: [0, 82]
odd: [37]
even: [34]
odd: [3, 41, 53]
even: [60, 62]
odd: [35, 27, 75, 43, 31]
even: [98, 56]
odd: [97]
even: [26]
odd: [73, 43]
even: [62, 74, 72]
```

Possibly useful, depending on the need, but we might want one group of even numbers, and one of odd. To get that we need to sort them based on our grouping function:

```
>>> sorted(numbers, key=even_odd)
[72, 98, 62, 60, 56, 74, 62, 26, 0, 34, 82, 12, 97, 41, 17, 73, 43, 37, 35, 3, 53, 27, 31, 43, 75]
```

We can now group that:

```
>>> for k, g in groupby(sorted(numbers, key=even_odd), even_odd):
...     print('{0}: {1}'.format(k, list(g)))
...
even: [72, 98, 62, 60, 56, 74, 62, 26, 0, 34, 82, 12]
odd: [97, 41, 17, 73, 43, 37, 35, 3, 53, 27, 31, 43, 75]
```

islice(iterable, start, stop=None, step=1)

Make an iterator that generates selected elements from `iterable`.

If `start` is non-zero and stop is unspecified, then the value for start is used as stop, and start is taken to be 0. Thus the supplied value specifies how many elements are to be generated, starting the the first one. In this sense, it functions as `take`.

If stop is specified, then elements from `iterable` are skipped until `start` is reached. Afterward, elements are generated consecutively unless `step` is set higher than one which results in items being skipped. If `stop` is None, then iteration continues until `iterable` is exhausted, if at all; otherwise, it stops at the specified position. If stop is specified and is not None, and is not greater than start then nothing is generated.

Unlike regular slicing, `islice()` does not support negative values for `start`, `stop`, or `step`. It can be used to extract related fields from data where the internal structure has been flattened (for example, a multi-line report may list a name field on every third line).

```
>>> list(islice('ABCDEF', 3))
['A', 'B', 'C']
>>> list(islice('ABCDEF', 3, stop=None))
['D', 'E', 'F']
>>> list(islice('ABCDEF', 3, stop=2))
[]
>>> list(islice('ABCDEF', 3, stop=4))
['D']
>>> list(islice('ABCDEF', 0, stop=None, step=2))
['A', 'C', 'E']
```

permutations(iterable, r=None)

Return successive r length permutations of elements in `iterable`.

If `r` is not specified or is `None`, then it defaults to the length of `iterable` and all possible full-length permutations are generated.

Permutations are generated in lexicographic sort order. So, if `iterable` is sorted, the permutation tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. So if the input elements are unique, there will be no repeat values in each permutation.

```
>>> list(permutations('ABCD', 2))
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'A'), ('B', 'C'), ('B', 'D'), ('C', 'A'), ('C', 'B'), ('C', 'D'), ('D', 'A'), ('D', 'B'), ('D', 'C')]
>>> list(permutations(range(3)))
[(0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 2, 0), (2, 0, 1), (2, 1, 0)]
```

product(*iterables, repeat=1)

Cartesian product of `iterables`.

Roughly equivalent to nested for-loops in a generator expression. For example, `product(A, B)` generates the same as `((x,y) for x in A for y in B)`.

The nested loops cycle like an odometer with the rightmost element advancing on every iteration. This pattern creates a lexicographic ordering so that if `iterables` are sorted, the product tuples are emitted in sorted order.

To compute the product of an iterable with itself, specify the number of repetitions with the optional repeat keyword argument. For example, `product(A, repeat=4)` means the same as `product(A, A, A, A)`.

```
>>> list(product('ABCD', 'xy'))
[('A', 'x'), ('A', 'y'), ('B', 'x'), ('B', 'y'), ('C', 'x'), ('C', 'y'), ('D', 'x'), ('D', 'y')]
>>> list(product(range(2), repeat=3))
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]
```

repeat(object, times=None)

Make an iterator that generates `object` over and over again. Runs indefinitely unless the `times` argument is specified. Used as argument to `map()` for invariant parameters to the called function. Also used with `zip()` to create an invariant part of a tuple record.

```
>>> list(repeat(1, 5))
[1, 1, 1, 1, 1]
```

starmap(function, iterable)

Make an iterator that computes `function` using arguments obtained from `iterable`. Used instead of `map()` when argument parameters are already grouped in tuples from a single iterable (the data has been “pre-zipped”). The difference between `map()` and `starmap()` parallels the distinction between `function(a,b)` and `function(*c)`.

```
>>> list(map(lambda x, y: x + y, [1, 2, 3], [4, 5, 6]))
[5, 7, 9]
>>> list(starmap(lambda x, y: x + y, zip([1, 2, 3], [4, 5, 6])))
[5, 7, 9]
>>> list(starmap(lambda x, y: x + y, [[1, 4], [2, 5], [3, 6]]))
[5, 7, 9]
```

takewhile(predicate, iterable)

Make an iterator that generates elements from `iterable` as long as `predicate` is true when applied to them.

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

tee(iterable, n=2)

Return `n` independent iterators from a single `iterable`. The resulting iterators *contain* the elements from the original by generate them completely independently.

```
>>> a, b = tee("ABCDE", 2)
>>> next(a)
'A'
>>> next(b)
'A'
>>> take(2, a)
```

```
['B', 'C']
>>> take(3, b)
['B', 'C', 'D']
```

zip_longest(*iterables, fillvalue=None)

Make an iterator that aggregates elements from each of the `iterables`. If the `iterables` are of uneven length, missing values are filled-in with `fillvalue`. Iteration continues until the longest iterable is exhausted. Contrast this with the builtin function `zip` which will stop when the *shortest* iterable is exhausted.

```
>>> list(zip_longest('ABCD', 'xy', fillvalue='-'))
[('A', 'x'), ('B', 'y'), ('C', '-'), ('D', '-')]
```

Extras

The CPython documentation provides a selection of useful functions that are built on top of itertools. These have been made available in the **adafruit_itertools_extras** module. not only are these useful, but looking through the implementations is instructive on how iterators and the itertools functions can be used.

`all_equal(iterable)`

Returns `True` if all the elements of `iterable` are equal to each other.

```
>>> all_equal([6, 2*3, 12//2, 4+2])
True
>>> all_equal([6, 2*3, 12//2, 5, 4+2])
False
```

`dotproduct(vec1, vec2)`

Compute the dot product of two vectors. This is the sum of the products of pairs (one item from each vector)

```
>>> dotproduct([1, 2, 3], [1, 2, 3])
14
```

This is equivalent to (1 * 1) + (2 * 2) + (3 * 3) = 1 + 4 + 9 = 14

`first_true(iterable, default=False, pred=None)`

Returns the first *truthy* value in `iterable`, i.e. the first value `x` for which `bool(x)` evaluates to `True`.

If no true value is found, returns `default`.

If `pred` is not `None`, returns the first item for which `pred(item)` is true.

`first_true([a,b,c], x)` is equivalent `a or b or c or x`
`first_true([a,b], x, f)` is equivalent `a if f(a) else b if f(b) else x`

flatten(iterable_of_iterables)

Flatten one level of nesting.

```
>>> list(flatten(['ABC', 'DEF']))
['A', 'B', 'C', 'D', 'E', 'F']

>>> list(flatten([[1, [2, 3]], [4, [5], 6]]))
[1, [2, 3], 4, [5], 6]
```

grouper(iterable, n, fillvalue=None)

Collect data from `iterable` into fixed-length chunks of size `n`. The final chunk will be filled in with `fillvalue` if iterable has less that a multiple of `n` elements.

```
>>> list(grouper('ABCDEFG', 3, 'x'))
[('A', 'B', 'C'), ('D', 'E', 'F'), ('G', 'x', 'x')]
>>> list(grouper('ABCDEFG', 2))
[('A', 'B'), ('C', 'D'), ('E', 'F'), ('G', None)]
```

iter_except(func, exception)

Call `func` repeatedly until `exception` is raised, yielding the result of each call.

```
>>> s = [1, 2, 3]
>>> s.pop()
3
>>> s.pop()
2
>>> s.pop()
1
>>> s.pop()
Traceback (most recent call last):
File "", line 1, in
IndexError: pop from empty list

>>> s = [1, 2, 3]
>>> list(iter_except(s.pop, IndexError))
[3, 2, 1]
```

ncycles(iterable, n)

Returns an iterable from `n` copies of the elements from `iterable`.

```
>>> list(ncycles('ABC', 3))
['A', 'B', 'C', 'A', 'B', 'C', 'A', 'B', 'C']
```

nth(iterable, n, default=None)

Returns the `nth` (0-based) item of `iterable` or a `default` value (`None` by default) if `n` is out of range.

```
>>> nth('ABCD', 0)
'A'
>>> nth('ABCD', 3)
'D'
```

```
>>> nth('ABCD', 4)
>>> nth('ABCD', 4, 'Z')
'Z'
```

padnone(iterable)

Returns the elements from `iterable` and then returns `None` indefinitely.

Useful for emulating the behavior of the built-in `map()` function.

`take(5, padnone([1, 2, 3]))` -> `1 2 3 None None`

pairwise(iterable)

Pair up values in `iterable`.

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

partition(predicate, iterable)

Use `predicate` to partition entries in `iterable` into false entries and true entries.

```
>>> t, f = partition(lambda x: x % 2, range(10))
>>> list(t)
[0, 2, 4, 6, 8]
>>> list(f)
[1, 3, 5, 7, 9]
```

prepend(value, iterator)

Prepend a `value` in front of `iterator`.

`prepend(1, [2, 3, 4])` -> `1 2 3 4`

quantify(iterable, predicate=bool)

Count the number of elements in `iterable` for which `predicate` is `True`. By default it counts elements `x` where `bool(x)` results in `True`.

```
>>> quantify([2, 56, 3, 10, 85], lambda x: x >= 10)
3
```

repeatfunc(func, times=None, *args)

Lazily repeat calls to `func` a number of times set by `times` with specified arguments (if any). If `times` is `None` (the default), call it infinitely. Return values from the function are generated by the resulting iterator.

```
>>> take(5, repeatfunc(random.random))
[0.777482, 0.718999, 0.953034, 0.388373, 0.982035]
```

roundrobin(*iterables)

Return an iterable created by repeatedly picking a value from each of `iterables`, in order. As individual iterables are

exhausted, they are dropped from the picking.

```
>>> list(roundrobin('ABC', 'D', 'EF'))
['A', 'D', 'E', 'B', 'F', 'C']
```

tabulate(function, start=0)

Return an iterable populated by applying `function` to a sequence of consecutive integers, starting at `start` (defaults to 0).

```
>>> take(5, tabulate(lambda x: x * x))
[0, 1, 4, 9, 16]
```

tail(n, iterable)

Return an iterator over the last `n` items of `iterable`. This only works if `iterable` is finite.

```
>>> list(tail(3, 'ABCDEFG'))
['E', 'F', 'G']
```

take(n, iterable)

Return first `n` items of `iterable` as a list.

```
>>> take(3, 'ABCDEF')
['A', 'B', 'C']
```

Examples

Assembling Data

Let's say you are logging readings from a temperature sensor (the Si7021 for example). For some reason you need a reading number and a timestamp in addition to the reading, rounded to an integer.

You could do this the *standard* way:

```
import time
import board
import busio
import adafruit_si7021

i2c = busio.I2C(board.SCL, board.SDA)
sensor = adafruit_si7021.SI7021(i2c)

def read_temperature():
    return sensor.temperature

def now():
    return time.monotonic()

def non_iter():
    reading_number = 1
    while True:
        datapoint = (count, now(), int(read_temperature()))
        print(datapoint)
        count += 1
        time.sleep(20.0)
```

Using iterators we can do this in a slightly different way:

```
import time
import board
import busio
import adafruit_si7021
from adafruit_itertools import count
from adafruit_itertools_extras import repeatfunc, roundrobin

i2c = busio.I2C(board.SCL, board.SDA)
sensor = adafruit_si7021.SI7021(i2c)

def read_temperature():
    return sensor.temperature

def now():
    return time.monotonic()

def iter():
    datapoints = zip(count(1), repeatfunc(now), map(int, repeatfunc(read_temperature)))
    while True:
        print(next(datapoints))
        time.sleep(20.0)
```

The iterator based version is a couple lines shorter, but what's the advantage beyond that?

In the non-iterator version the data point is being constructed in the loop each time. In the iterator-based version, an iterator is constructed that will generate data tuples as requested. More importantly tuple generation is encapsulated in an iterator. That means it can be stored in a variable (as it is in this example, in `datapoints`), but also passed as an argument. This last thing is significant in that it lets you separate (*decouple* in programming lingo) the generate of data, from the use of it. The code in the loop doesn't care how to make a datapoint, it just asks for the next one. The iterator doesn't care how or when it's elements are being used, it just hands back the next one when asked. It doesn't even really know or care how the elements are being made. there's code that puts together the iterator, and other code that asks for elements. All the iterator does is construct a single element based on how it's been wired up. As such it serves as a great intermediary between the code that knows how to get/make the data and the code that knows how to use it.

Filtering by Divisibility

Here's an example from SICP*: let's say for some reason you need numbers that are not divisible by 7.

First we can make a simple predicate to determine divisibility:

```
def divisible(x, y):
    return (x % y) == 0
```

To get numbers that are not divisible by 7 we can use `filterfalse` with a lambda:

```
>>> list(filterfalse(lambda x: divisible(x, 7), range(30)))
[1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 15, 16, 17, 18, 19, 20, 22, 23, 24, 25, 26, 27, 29]
```

In practice we probably don't want to pass in a range, but rather an infinite sequence of integers:

```
>>> take(20, filterfalse(lambda x: divisible(x, 7), count()))
[1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 15, 16, 17, 18, 19, 20, 22, 23]
```

The next step would be to package this up to pass to whatever function wants a sequence of integers that are not divisible by 7:

```
>>> no_sevens = filterfalse(lambda x: divisible(x, 7), count())
>>> take(20, no_sevens)
[1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 15, 16, 17, 18, 19, 20, 22, 23]
```

We can generalize by using a function that returns a lambda to construct the test:

```
>>> def divisibility_checker(x):
...     return lambda n: (n % x) == 0
>>> by7 = divisibility_checker(7)
>>> no_sevens = filterfalse(by7, count())
>>> take(20, no_sevens)
[1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 15, 16, 17, 18, 19, 20, 22, 23]
```

Once more, this lets us separate the construction of the sequence from the use of it.

Using iterators also let you directly make requests like:

- Give me the next 5 items.
- Give me the 10th item.
- Skip 10 items and give me 5.
- Ignore items until one is over 100.

And so on. All from the same iterator. All the sequencing, traversal, and computation is bundled up in the iterator and all your code has to do is ask for items from it. This can help make your code simpler and clearer.

* The Structure and Implementation of Computer Programs ... one book every programmer should read and study.