CircuitPython 101: Working with Lists, Iterators and Generators

Created by Dave Astels

https://learn.adafruit.com/circuitpython-101-list-and-things-iterators-generators

Last updated on 2023-08-29 03:51:54 PM EDT

Table of Contents

Overview

This guide is part of a series on some of the more advanced features of Python, and specifically CircuitPython. Are you new to using CircuitPython? No worries, there is a full getting started guide here ().

Adafruit suggests using the Mu editor to edit your code and have an interactive REPL in CircuitPython. You can learn about Mu and its installation in this tutorial ().

CircuitPython gets far more interesting when used with the new SAMD51 (M4) based boards () and the Raspberry Pi (). Not only do they have much higher clock speeds, they also have much more RAM. CircuitPython programs run significantly faster as a result, and they can be much larger and/or work with much more data. With this space comes the capability to move beyond simple scripts to more elaborate programs.

The goal of this series of guides is to explore Python's mechanisms and techniques that will help make your more ambitious CircuitPython programs more manageable, understandable, and maintainable.

In CircuitPython 101: Basic Builtin Data Structures () we learned about Python's list data structure. We saw how to manipulate them and how to create literal instances.

In CircuitPython 101: Functions () we learned about lambdas, which are single expression functions that don't have a name assigned to them.

In this guide we'll explore some features of Python for working with lists: some list processing functions and list comprehensions. Then we'll go beyond simple lists with i terators and generators.

List Functions

Slicing

We already saw how you can access items in a list using the `list[index]` syntax. This is the simplest case: it gives you a single item. The more general form of the syntax allows you to extract a contiguous subset of the list.

You can specify the index (starting at 0, remember) of the first item (i.e. inclusive), and the index of the item one past the last one you want (i.e. exclusive). For example `l[0:len(l)]` will give you the entire list.

```
l = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
l[3]
#'d'
l[3:6]
#['d', 'e', 'f']
l[0:len(l)]
#['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

If you are interested in items from the first or last of the list, it can be simplified to omit the `0` or `len(l)`:

```
l[:4]
#['a', 'b', 'c', 'd']
l[4:]
#['e', 'f', 'g']
```

Filtering

The `filter` function takes a predicate (i.e. a function that returns a boolean value) and a list. `filter` returns a filter object that generates a list that contains only those elements from the original list for which the predicate returns `True`. Filter predicates can either be a reference to a named function or, quite often, a lambda.

Say you want to pick out only the numbers in a list that are multiples of 3. If `a` is your original list of numbers, say the result of `range(20)`:

```
a = range(20)
list(filter(lambda x: x % 3 == 0, a))
#[0, 3, 6, 9, 12, 15, 18]
```

Notice the need to use `list` to convert the filter object to a list.

Mapping

Another common thing to do with lists is to perform some operation on each item, accumulating the results into a new list. The `map` function does this. It takes a function (which is often a lambda) and the original list. Similar to `filter`, `map` returns a map object.

Let's take the same list `a`, and generate a list of the squares of its contents.

```
list(map(lambda x: x**2, a))
#[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324,
361]
```

`map` isn't limited to processing a single list; you can pass it any number of lists, with the restriction that the function has to have that number of parameters.

```
list(map(lambda x, y: x + y, [0,1,2,3], [4,5,6,7]))
#[4, 6, 8, 10]
```

We can combine `map` with `filter` to process a subset of a list. Let's get the squares of the multiple of 3:

```
list(map(lambda x: x**2, filter(lambda x: x % 3 == 0, a)))
#[0, 9, 36, 81, 144, 225, 324]
```

While this works, it can be a bit cumbersome at times. Thankfully Python provides a better alternative, which we will examine shortly.

Reducing

The third function we're going to look at is `reduce`. It's different than `filter` and `map` in that it doesn't result in a sequence of values. Instead it takes a list, a function, and an optional initial value and uses them to compute a single value. It, in essence, r educes the list to a single value.

Unlike `filter` and `map`, `reduce` isn't a built-in; it's in the `functools` module. There are plenty of other fun, useful things in that module but we won't be looking at those now. This module isn't included with CircuitPython, but it is with MicroPython. You can find it at https://github.com/micropython/micropython-lib (). To use it with CircuitPython, copy micropython-lib/functools/functools.py (either from a clone of the repository or just the single downloaded file) to the CIRCUITPY drive. Once it's there you'll need to import it:

```
from functools import reduce
```

`reduce` is sometimes confusing at first, but it's actually quite simple. The function you pass it takes two parameters; let's call them `accumulator` and `x`. This function is called for items of the list (in order), that's the second (`x`) argument. The first argument (`accumulator`) is the result from the previous time the function was called.

The first time through is a little different. If the optional third argument is supplied to `reduce`, it is used as the result of the previous call (as mentioned above). If it's omitted, the first value from the list is used instead.

Some examples should illustrate it:

```
reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])
```

this is equivalent to:

```
((((1+2)+3)+4)+5)
```

If, instead, we had:

```
reduce(lambda x, y: x+y, [1, 2, 3, 4, 5], 10)
```

it would be the same as:

```
(((((10+1)+2)+3)+4)+5)
```

We can give it a try and see:

```
reduce(lambda x,y:x+y,[1,2,3,4,5])
#15
reduce(lambda x,y:x+y,[1,2,3,4,5], 10)
#25
```

Note: the type of the result of `reduce` will be the type of the initial value, either the one supplied or the first item in the list.

Of course, like `filter` and `map`, you don't need a lambda. You can use the name of a function if there's one that does exactly what you want. Since Python lambdas are limited to a single expression, you might sometimes need to write a function just to be used for one of these functions.

How about another example.

```
def prepend(l, x):
    l.insert(0, x)
    return l

reduce(prepend, ['a', 'b', 'c'], [])
# returns ['c', 'b', 'a']
```

We can't use a lambda here because `insert` doesn't return the list. We need to do that as a second step, which we can't do in a lambda. Also, we need to provide the empty list as the initial accumulator value so that

1. the result will be a list, and
2. there's something to insert the first item into.

List Comprehensions

List comprehensions provide a more concise way of doing simple filtering and mapping. Their use is very idiomatic in Python, so you'll see them a lot. Understanding them will not only let you write better Python code, it will also help you understand other code.

Let's start with the concept of mapping. A list comprehension can be used to do the mapping we saw earlier:

```
a = range(20)
[x**2 for x in a]
#[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324,
361]
```

One thing to notice is that this results directly in a list unlike `map`, whose result needs to be converted to a list. Secondly, notice how much simpler and direct the syntax is:

`[function_of_variable for variable in a_list]`

Compare this to:

`list(map(lambda x: function_of_variable, a_list))`

This is even more pronounced if we add the filter:

`[variable for variable in a_list if predicate_of_variable]`

Compare this to:

`list(map(lambda x: function_of_variable, (filter lambda x: predicate_of_variable, a_list)))`

To pick out the multiples of 3 as before we can use:

```
[x for x in a if x % 3 == 0]
#[0, 3, 6, 9, 12, 15, 18]
```

As before we can combine mapping and filtering:

```
[x**2 for x in a if x % 3 == 0]
#[0, 9, 36, 81, 144, 225, 324]
```

Compare this to the version using map and filter:

```
list(map(lambda x: x**2, filter(lambda x: x % 3 == 0, a)))
#[0, 9, 36, 81, 144, 225, 324]
```

Iterators

Comprehensions are not without their problems. They require all of the items in the list to be in memory and they create another list in memory for the result. This works fine for smallish lists (depending on how much memory you have available) but don't work as well with large amounts of data and/or memory constrained environments. This is where iterators can be useful.

One big advantage of an iterator is that the items aren't all in memory; only one is at a time. As stated above, that's not much of an advantage with small lists. But it should be clear that this is a huge advantage for large lists; it lets you minimize the amount of memory required. Furthermore, since they only use one item at a time, they're great when you don't know how many items there are. One example of this is if you are processing logged data from a file, or an ongoing series of readings from a sensor. Iterators work by only computing the next value when it's asked for, and only the next one. Another advantage is that iterators allow us to pass them around in our code, pulling out values as needed. Finally, because each value is computed only when it's asked for, an iterator can provide an infinite sequence of values.

Python's for loop uses iterators, or rather iterables, behind the scenes. The general form of the for loop is:

```
for variable in iterable:
    ...
```

So lists, ranges, and strings are all iterables. However, to iterate over them you need to use a `for` loop. An iterator provides a more general way of getting a value from an iterable, then getting the next value, then the next, and so on. The `iter` function wraps an iterator around an interable, allowing us to step through its values as we choose without having to use a `for` loop. We pass the iterator to the `next` function to, as you might expect, get the next value.

If we try to use `next()` with ranges and lists, it clearly doesn't work.

```
r = range(10)
next(r)
#Traceback (most recent call last):
#File "", line 1, in
#TypeError: 'range' object is not an iterator
l = [1, 2, 3]
next(l)
#Traceback (most recent call last):
#File "", line 1, in
#TypeError: 'list' object is not an iterator
```

Wrapping in iterator around them will let this work.

```
r = iter(range(10))
next(r)
#0
next(r)
#1
l = iter([1, 2, 3])
next(l)
#1
next(l)
#2
```

Generators

Generators are a convenient way to make custom iterators. They take two forms: gen erator functions and generator expressions.

Generator Functions

These are functions that result in a generator object (basically an iterator) rather than a function object. That object can then be used in a `for` loop or passed to the `next` function. The differentiating feature of a generator function vs. a regular function is the use of the `yield` statement.

For example, if we wanted something like `range` that instead worked on floats, we could write one as a generator function:

```
def range_f(start, stop, step):
    x = start
    while x &lt;= stop:
        yield x
        x += step
```

Now we can write code such as:

```
for x in range_f(0, 1.0, 0.125):
    print(x)
```

That gives output of:

`0`
`0.125`
`0.25`
`0.375`
`0.5`
`0.625`
`0.75`
`0.875`
`1.0`

And we can use it with `next`:

```
r = range_f(0, 1, 0.125)
next(r)
#0
next(r)
#0.125
next(r)
#0.25
next(r)
#0.375
next(r)
#0.5
```

The first time the resulting generator object is used, the function starts executing from the beginning. In this case, it creates a local variable `x` and sets it to the value of `start`. The `while` loop executes, and the `yield` statement is encountered. This pauses execution of the function and produces its argument (the current value of `x` in this case). The next time the generator object is used (iterating in a for loop or passed to a call to `next`), the function resumes execution immediately after the `yield` statement. In this case, `x` is advanced, and the `while` loop goes back to the condition check. This continues until the iteration is complete (`x > stop` in this example). When that happens, the function continues past the loop, possibly doing some wrap up before exiting. When it exits a `StopIteration` exception is raised.

```
next(r)
#1.0
next(r)
#Traceback (most recent call last):
#File "", line 1, in
#StopIteration:
```

Generator Expressions

Generator expressions use the same syntax as list comprehensions, but enclose it in parentheses rather than brackets:

```
(x for x in range(10))
#&lt;generator object '&lt;genexpr&gt;' at 20002590&gt;
```

Notice that the result isn't a list. It's a generator object as we saw above. To get values from it, we call `next` passing in the generator object:

```
g = (x for x in range(10))
next(g)
#0
next(g)
#1
next(g)
#2
next(g)
#3
next(g)
#4
```

We can also take `g` and use it in a `for` loop:

```
g = (x for x in range(10))
for x in g:
    print(x)
#0
#1
#2
#3
#4
#5
#6
#7
#8
#9
```

Now, think about how you would go about processing items from an infinitely long list. "But," you say, "where would be have an infinite amount of data in a microcontroller context?" How about when you're taking temperature readings, and you continue taking temperature readings for the entire time the system is running. That is effectively an infinitely long list of data. And infinite really just means that: you have no idea how long it is.

While you could just read the temperature in the main `while True` loop (and that's perfectly fine for a simple application that is mainly just a `while True` loop) as your code grows and you want to organize it in a cleaner and more modular way (we are assuming that you're running on at least a SAMD51 chip, if not a Raspberry Pi, so you have plenty of room for large and well structured code). It can be cleaner to encapsulate that sequence of readings in an iterator. This lets it be passed around as needed and asked for another value where and when desired.

Another nice thing about using iterators is that they are now objects and can be worked with. There are ways to combine and ask for values from them in clean, readable ways.

Examples

One neat thing we can do with generators is to make infinite sequences. These don't have a condition to end the iteration. For example, we can make an infinite sequence of integers:

```
def ints():
    x = 0
    while True:
        yield x
        x += 1
```

We can use this with `next` to get a ongoing sequence of integers, but that's not very interesting.

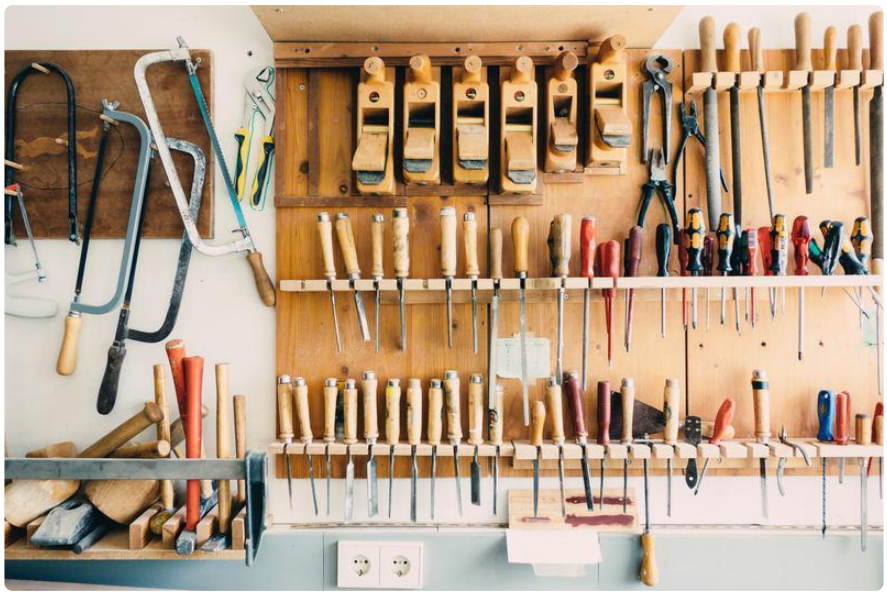
```
>>> i = ints()
>>> next(i)
0
>>> next(i)
1
>>> next(i)
2
>>> next(i)
3
>>> next(i)
4
```

Far more interesting is to use `ints()` in other generators. Here are some simple examples:

```
evens = (x for x in ints() if x % 2 == 0)
odds = (x for x in ints() if x % 2 != 0)
squares = (x**2 for x in ints())
```

Note that each use of `ints()` returns a new and unique generator object.

The itertools Module



Python has a module designed to make iterators (and by association, generators) easier to use and more flexible: `itertools`. It contains several functions; we'll look at a few of the more useful. As with `functools`, this isn't part of CircuitPython, but is part of MicroPython. It provides a subset of CPython's `itertools` module.

Let's look at a few of the more useful functions available. It you look at https://docs.python.org/3.6/library/itertools.html () you will find more detailed documentation. Also, if you look at https://docs.python.org/3.6/library/itertools.html#itertools-recipes () you'll find ways to use the basic functions to build more that can be handy.

The descriptions below should serve to give you a taste of what's possible.

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

With a list, we can take a slice to extract part of the list:

```
l = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
l[2:5]
#['c', 'd', 'e']
```

This doesn't work on an iterable, even though it's still a useful operation.

```
i = (x for x in ['a', 'b', 'c', 'd', 'e', 'f', 'g'])
i[2:5]
#Traceback (most recent call last):
```

```
#File "", line 1, in
#TypeError: 'generator' object is not subscriptable
```

This is because the iterable doesn't know about the sequence as a whole; it only knows how to compute the next item.

`itertools` provides the `islice` function which does much the same thing for iterators.

`islice` returns an iterator that iterates over part of the original one. As with list slicing, the `stop` parameter is the index one past the last item desired. If `start` is omitted, it is taken as `0`. If `step` is omitted, it is taken as `1`. If `start` is supplied, that many items will be skipped first. If `step` is supplied and greater than `1`, items will be skipped between the ones in the iterator returned by `islice`.

```
islice('ABCDEFG', 2)
#A B
islice('ABCDEFG', 2, 4)
#C D
islice('ABCDEFG', 2, 7)
#C D E F G
islice('ABCDEFG', 0, 6, 2)
#A C E
```

count([start, [step]])

The count function returns an infinite iterator that yields a sequence of evenly spaced integers (separated by `step`) that start at `start`.

```
list(islice(count(), 0, 10))
#[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
list(islice(count(5), 0, 10))
#[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
list(islice(count(5, 2), 0, 10))
#[5, 7, 9, 11, 13, 15, 17, 19, 21, 23]
```

cycle(iterable)

This returns an iterator that returns items from the supplied iterable, looping indefinitely. This isn't so good if the supplied iterable is long since the first time through, it might have to cache its contents.

```
c = cycle('abcd')
next(c)
#'a'
next(c)
#'b'
next(c)
#'c'
```

```
next(c)
#'d'
next(c)
#'a'
next(c)
#'b'
next(c)
#'c'
next(c)
#'d'
list(islice(c, 12))
#['a', 'b', 'c', 'd', 'a', 'b', 'c', 'd', 'a', 'b', 'c', 'd']
```

`cycle` can be handy if you, for example, need alternating `True` / `False` values:

```
list(islice(cycle([True, False]), 12))
#[True, False, True, False, True, False, True, False, True, False, True, False]
```

repeat(object[, n])

This creates an iterator that repeatedly returns `object`, `n` times or infinitely if `n` is omitted. At first this might not seem overly useful, but it can be in specific situations.

Going On From Here

In this guide we've looked at various ways to work with lists as well as had an introduction to list comprehensions, iterators, and generators.

The MicroPython port of the itertools module has a few other functions that we haven't mentioned. If your interest is piqued, have a look at the code and the CPython docs linked earlier.

Iterators are often called streams and are incredibly useful things. If you have an understanding of the Scheme language, The Structure and Interpretation of

Computers Programs () has a really good section on streams () and their uses. If you don't have that understanding, the book (and accompanying lecture videos) is a great way to gain it.