# Currying in CircuitPython

Created by Dave Astels

https://learn.adafruit.com/partials-in-circuitpython

Last updated on 2023-08-29 04:05:56 PM EDT

## Table of Contents

## Overview

If you happen to have at least dabbled with the programming language Haskell, or one similar, you may be familiar with currying, aka partially applied functions. Partial function application is an important mechanic in functional programming. Functional programming is a hot topic these days: many of its core features are showing up in other languages, including Python.

A partially applied function is a function that has been given some arguments, but not all. In Haskell you might see a function declaration such as:

```
add     :: Integer -> Integer -> Integer
add x y =   x + y
```

Our traditional understanding of functions would lead us to believe that this declares a function `add` which takes two integer arguments and returns an integer result which is the sum of the arguments.  We would be wrong. This, in fact, declares a function that takes an integer and returns another function that takes an integer and returns one. I.e.

```
add:: Integer -> (Integer -> Integer)
```

When `add` is called, it returns a function that has been partially applied (to the first argument). The resulting function can then be applied to the second (and final) argument to generate a result.

Normally, we'd call `add` with both parameters:

```
Prelude> add 2 3
5
```

but because of being able to partially apply functions, we can call `add` with one parameter and get a function back. For example:

```
Prelude> inc = add 1
Prelude> :t inc
inc :: Num a => a -> a
```

`inc` is a function that takes an number and returns the number plus one. (`:t` tells you the type of its argument) That function is essentially:

```
inc y = 1 + y
```

So now we can use `inc`:

```
Prelude> inc 2
3
```

This is called currying in honor of Haskell Curry, after whom the Haskell language is also named.

This is just how things work in Haskell, but Python gives us the tools we need to do it there as well.

## CircuitPython

We'll be using CircuitPython for this guide. 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 ().

Install the latest release of CircuitPython, version 4 () (or higher) for your particular CircuitPython-compatible board. Follow the instructions here () using the appropriate CircuitPython UF2 file.

## Simple Partial Application

We can make simple partials using the Python `lambda`. Here's a Python version of the `add` function we looked at in Haskell:

```
def add(x, y):
    return x + y
```

Python doesn't support currying as part of the language, so we can't just do:

`add(1)`

If we try, an exception is raised:

```
>>> inc = add(1)
Traceback (most recent call last):
File "", line 1, in
TypeError: function takes 2 positional arguments but 1 were given
```

What we can do is use a lambda as described in this guide ():

```
>>> inc = lambda x: add(1, x)
>>> inc(2)
3
```

We can go a bit further with this approach and define a function that does this for us, taking the first argument on which to partially apply add:

```
def make_adder(x):
    return lambda y: add(x, y)
```

Now we can use `make_adder` to, in essence, partially apply `add` to an argument:

```
>>> inc = make_adder(1)
>>> inc(2)
3
```

Since `make_adder` is general, we can use it to create any "add a constant" functions. E.g.

```
>>> add10 = make_adder(10)
>>> add10(5)
15
```

This approach can be useful, but requires a specialized partial application function for each case. Python provides the capability to come up with a more general solution.

## More Complex Partial Application

When thinking about a more general way to partially apply a functions, it would seem to be nice to partially apply to any number of argument, and to handle keyword arguments. As an exercise, we'll build it up step by step.

Let's start by considering the general form of the partial application function. We'll need to provide the function to be partially applied and the arguments to which it should be partially applied. We'll start with positional arguments.

```
def partial(func, *args):
    pass
```

The next step is to create the function to capture the partial application:

```
def partial(func, *args):
    def newfunc(*fargs):
        pass
    return newfunc
```

`newfunc` has to take the arguments that are passed to it and combine them with those supplied earlier (to the `partial` function), then call the original function with the resulting arguments:

```
def newfunc(*fargs):
    return func(*(args + fargs))
```

It can simply concatenate the positional arguments (keeping the order correct) since they are in lists.

The `partial` function is now:

```
def partial(func, *args):
    def newfunc(*fargs):
        return func(*(args + fargs))
    return newfunc
```

Now we can say:

```
>>> inc = partial(add, 1)
>>> inc(2)
3
```

## With Keywords

Now we can add keyword arguments. We can make a simple function to build strings:

```
>>> def wrap(s, prefix='', suffix=''):
... return prefix + s + suffix
...
>>> wrap('hello')
'hello'
>>> wrap('hello', suffix=' world')
'hello world'
>>> wrap('hello', suffix=' world', prefix='>>> ')
'>>> hello world'
```

We can use partial application to take `wrap` and make a greeter function:

```
>>> greet = partial(wrap, prefix='Hello, ')
>>> greet('Dave')
'Hello, Dave'

>>> greet('Phil', suffix='!')
'Hello, Phil!'
```

To expand `partial` to extend partial application to keyword arguments, we need to add them to the functions, and combine them for the fully applied call. Since they are passed in dictionaries, we can't simply concatenate the way we did with the positional arguments; we need to merge the dictionaries using `update`. Our new partial function is now:

```
def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*(args + fargs), **newkeywords)
    return newfunc
```

## Wrap up

Functional Programming has been around for quite a while and concepts from it are making their way into more mainstream languages. Partial function application is one example that can be very handy to have in your toolbox.

A big advantage of using partial application is allowing us to take general functions and specialize them at runtime.

Instead of writing special purpose functions, we can reuse general functions. This allows us to have less code, especially if the functions are complicated. It helps our code to be simpler, since using situation specific functions (with fewer arguments and more specific names) makes our code less cluttered and more readable.

Finally, reusing general purpose functions and specializing them by partial application doesn't add new logic which could have bugs.