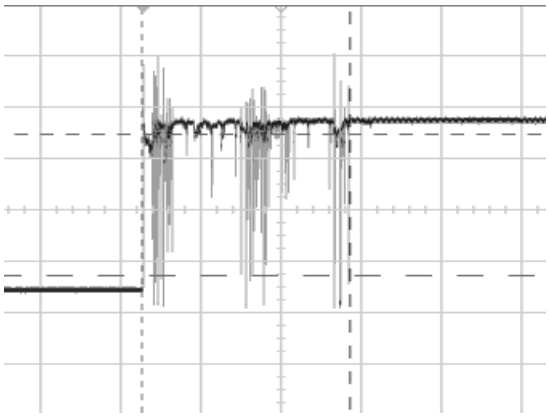
adafruit learning system

Python Debouncer Library for Buttons and Sensors

Created by Dave Astels



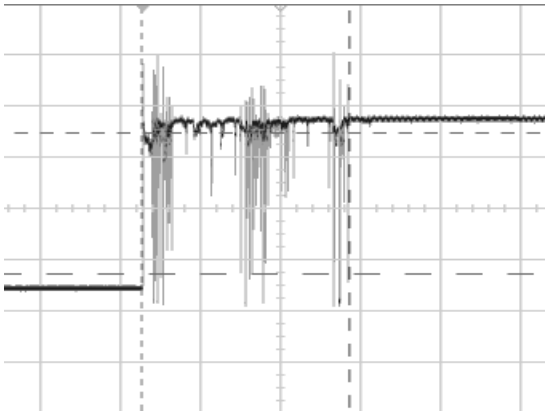
Last updated on 2020-12-29 04:23:58 PM EST

Guide Contents

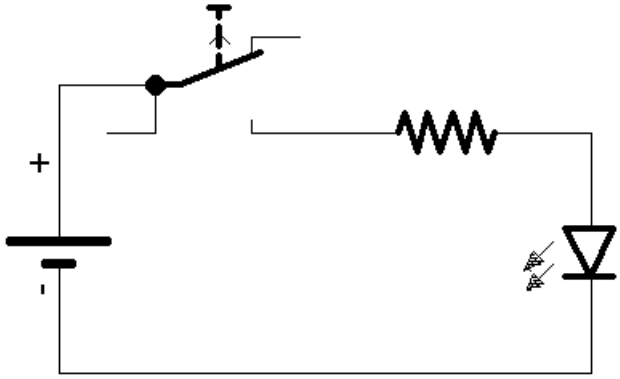
Overview

We use switches a lot in our projects. Pushbutton switches, slide switches, micro switches. We also have touch sensors, light sensors, sound sensors, and many others.

These physical devices are messy. By that I mean that the change from one state to the other (e.g. pressing or releasing a button) bounces back and forth before settling into the new state. You can see this in the oscilloscope trace below.



These bounces are fast. The bouncy section of the above trace lasts for a couple milliseconds. If you hooked a switch to an LED as below, you'd never notice the bounce. They're far to fast for that.



However code running on a microcontroller is a lot faster that our eyes and brains. It **can** see those bounces. If you have code that's looking for the push and release of a switch, it might be something like this:

```
import board
import digitalio

pin = digitalio.DigitalInOut(board.D12)
pin.direction = digitalio.Direction.INPUT
pin.pull = digitalio.Pull.UP

button_state = False
while True:
    pressed = pin.value
    if pressed != button_state:
        print(pressed)
    button_state = pressed
```

Some switches bounce more than others and with some CircuitPython code won't see the bounces whereas C++ code would. Python's slower performance can actually do you a favor sometimes. But not always. And as the microcontrollers CircuitPython runs on get faster, bouncing will be more of a problem.

This guide introduces the adafruit_debouncer library and its Debouncer class. It starts with the basic usage and proceeds to more advanced capabilities.

Basic Debouncing

To debounce an input pin we simply create an instance of `Debouncer`, passing it a configured `DigitalInOut`, `TouchIn`, or any other object with a `value` property:

```
import board
import digitalio
from adafruit_debouncer import Debouncer

pin = digitalio.DigitalInOut(board.D12)
pin.direction = digitalio.Direction.INPUT
pin.pull = digitalio.Pull.UP
switch = Debouncer(pin)
```

For the debouncer to do its job, it has to sample the pin frequently, track it's value, etc. etc. That's done by calling the `update()` method, typically at the start of your main loop.

```
while True:
    switch.update()
```

The debouncer has three properties that we can use to see what the current state is.

`value` which is the current stable value of the input. Stable means that it has stayed the same for some amount of time.

This is the most basic way to use the debouncer: getting the debounced value.

```
    if switch.value:
        print('not pressed')
    else:
        print('pressed')
```

You can set the amount of time it takes for the value to stabilize by passing it (in seconds) into the constructor. If you don't, it defaults to 0.01 seconds (10 milliseconds). For example:

```
switch = Debouncer(pin, interval=0.05)
```

rose - if this is `True` if means that during the most recent `update()` call, the stable value changed from `False` to `True`.

fell - if this is `True` if means that during the most recent `update()` call, the stable value changed from `True` to `False`.

```
if switch.fell:
    print('Just pressed')
if switch.rose:
    print('Just released')
```

These last two are especially useful. In fact, they are generally far more useful than `value`. While `value` will give you the current stable value, `fell` and `rose` tells you that it just changed. For example if you want to know when a button is pushed or released (as opposed as to whether or not it's currently being held down) these will provide that information. They tell you that the stable value has changed as of the most recent call to `update()`. Furthermore, these will report `True` once, and only once, for each change (i.e. for the most recent call to `update`). By using them you can do something once, as soon as the state has changed.

For example, in his rotating drum sequencer project, John Park used debouncers on his infrared sensors to clean up their transition between black and white on the disk. Additionally, on the clock track sensor he used `rose` to tell him when a clock pulse started. He used that to know when to read the instrument track sensors (using `value`).

Other Basic Examples

See the examples folder in the adafruit_debouncer repository (https://adafru.it/PBC) for `TouchIn` and some other examples.

Advanced Debouncing

We've gone over how to set up debouncers on input pins: you create and configure a `digitalio.DigitalInOut` instance and pass it to `Debouncer()`.

But what if you want to debounce input signals to a Crickit? Or its touch inputs?

To do that you can use the more general way of making a debouncer: instead of passing in a `DigitalInOut` you pass in a predicate. Specifically, a zero-argument predicate. What's a predicate? Simply a function that returns a `Boolean` (i.e. `True` or `False`).

```
from adafruit_crickit import crickit
from adafruit_debouncer import Debouncer

ss = crickit.seesaw
ss.pin_mode(crickit.SIGNAL1, ss.INPUT_PULLUP)

def read_signal():
    return ss.digital_read(crickit.SIGNAL1)

signal_1 = Debouncer(read_signal)

while True:
    signal_1.update()
    if signal_1.fell:
        print('Fell')
```

Remember when we talked about lambdas in the guide on CircuitPython functions (https://adafru.it/Czt)? No? I encourage you to go read it now.

To reiterate (or if you didn't read that guide... yet), lambdas are simple, one line functions that return a value. The general form of a lambda is.

`lambda` arg_1, ..., arg_n: `value_to_return`

As shown above, lambdas can take some number of arguments, including none. These are followed by a colon and the code that computes the return value. Note that the `return` keyword is implicit.

So how does this help? Well, notice in the above code, we defined the function `read_signal` solely to be passed to the `Debouncer` constructor. All it does is return a single value. That makes it a prime candidate to be a lambda: simple, returns a value, and only used once. Using a lambda we can rewrite the above code as:

```
from adafruit_crickit import crickit
from adafruit_debouncer import Debouncer

ss = crickit.seesaw
ss.pin_mode(crickit.SIGNAL1, ss.INPUT_PULLUP)
signal_1 = Debouncer(lambda: ss.digital_read(crickit.SIGNAL1))

print('Starting')
while True:
    signal_1.update()
    if signal_1.fell:
        print('Fell')
```

Not a huge difference in this case: it let us get rid of four lines and a function. It does have some other impact though. If you define a function, that implies that what the function does is important enough to create a function around it. It implies it's an important enough concept in your code that it's worth giving a name to, and quite likely that it is something to be used more than once. In this case, none of that is true. It's simply to wrap the signal read in a zero-argument predicate. To be used once: passed into the `Debouncer` constructor. That's a perfect job for a lambda.

Function Factories

Let's look back to debouncing `DigitalInOut` objects (i.e. input pins on the microcontroller):

```
pin = digitalio.DigitalInOut(board.D12)
pin.direction = digitalio.Direction.INPUT
pin.pull = digitalio.Pull.UP
switch = Debouncer(pin)
```

That's 3 lines to create and configure the `DigitalInOut`, and another to create the `Debouncer` for it. But what if you have a Grand Central M4 Express and want to debounce a couple dozen (or more) inputs? For 24 inputs that's 96 lines. And that's just configuration.. .no actual program logic. Worse, it's not really 96 lines: it's 4 lines repeated almost identically 24 times.

Now we'll see the real value of lambdas. Not only are they a short-hand way to make simple functions, but they are closures. That might take a bit of explaining.

Closures

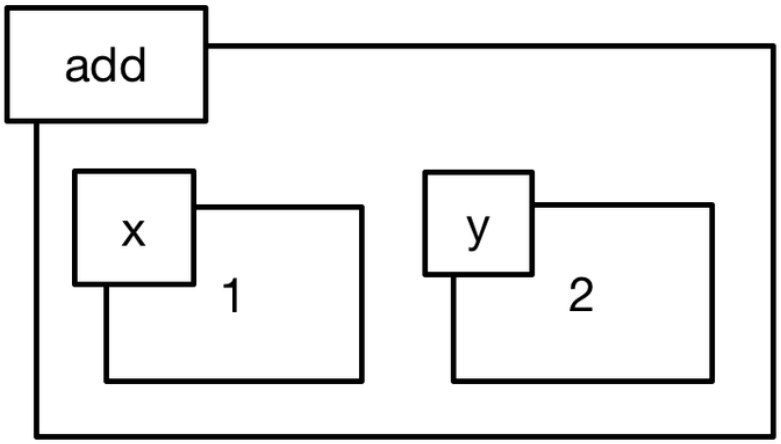
The best way to explain closures is to show how they work. Consider this function:

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

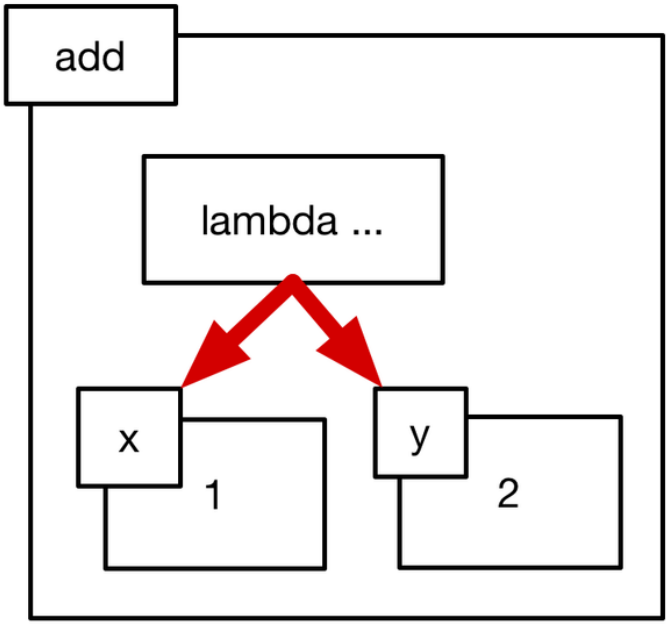
Then we can evaluate things like:

```
>>> add(1, 2)
3
```

At the prompt, or anywhere outside the `add` function, code can see `add`, but not `x` or `y`. They're hidden inside the function as shown here.

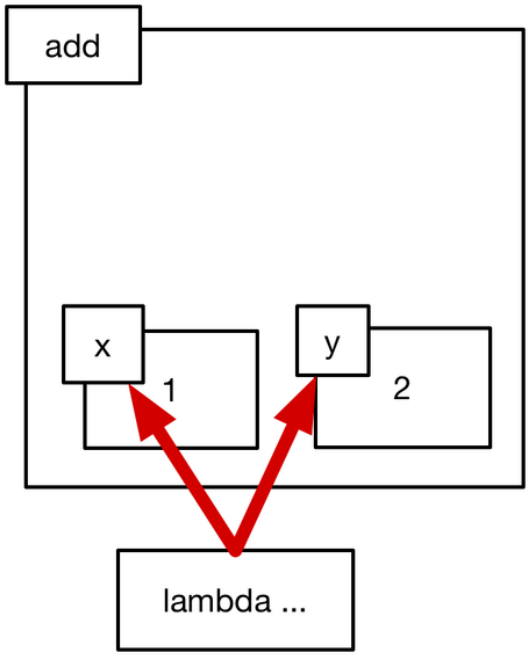


If a lambda is created inside the `add` function, the code in it has access to `x` and `y`:



A lambda is an object like any other (incidentally, so is any function). Because it's an object it can be stored in a variable and passed into another function as we've already seen when constructing a `Debouncer`. Key to this discussion is the fact that it can also be returned from a function.

Now, recall that the lambda has access to the variables (and parameters) in the function (more generally, the lexical scope) where it was created. When a lambda is returned from a function it **maintains access to those variables** that are hidden inside the function:



This allows us to do all manner of cool things. Here's a simple example. Lets riff on our add function to make an incrementer that adds 1 to whatever value is passed in, returning the result:

```
>>> def inc(x):
...     return x + 1
...
>>> inc(2)
3
```

That's fine, but all it does is add 1 to it's parameter. What if we needed functions to add arbitrary values instead of just 1? We could make a bunch of functions like `inc_1`, `inc_2`, `inc_42`, and so on. But what if we didn't know until runtime which ones we needed? Here's where lambdas shine. Let's make a function that makes custom lambdas:

```
>>> def make_adder(y):
...     return lambda x: x + y
...
>>> inc_1 = make_adder(1)
>>> inc_1(2)
3
>>> inc_42 = make_adder(42)
>>> inc_42(5)
47
```

The lambda that is created in and returned from `make_adder` maintains a reference to `make_adder`'s `y` parameter and, more importantly, it's value when the lambda was created. So, when the resulting lambda is saved to a variable and later executed it can add that specific value of `y` to it's argument.

Use with the Debouncer

Let's go back to that issue of having a couple dozen input pins to debounce. We can create a function that takes a pin (from the `board` module), creates and configures a `DigitalInOut` object, and returns a lambda that reads it.

```
def make_pin_reader(pin):
    io = digitalio.DigitalInOut(pin)
    io.direction = digitalio.Direction.INPUT
    io.pull = digitalio.Pull.UP
    return lambda: io.value
```

We can use this to make reader lambdas for however many debouncers we need.

```
pin5 = Debouncer(make_pin_reader(board.D5))
pin6 = Debouncer(make_pin_reader(board.D6))
...
```

Beyond Debouncing

We started with wanting to debounce a mechanical switch, and now we have the ability to debounce any zero-argument predicate function/lambda. From this we can get stable boolean values, as well as being informed when that stable value charges and in which direction (i.e. rising or falling)

We can use a debouncer for these three things even if the signal may not be all that noisy. Think about a temperature monitor that has an acceptable range. We can use a debouncer to monitor whether the temperature is in (or out of) that range.

Let's say we need to know if/when the temperature exceeds 30C and when it returns to an acceptable value. Maybe we turn a fan on when it's too hot and off when it's ok. On a Circuit Playground Express we could do something like:

```
import adafruit_debouncer
import adafruit_thermistor
import board

thermistor = adafruit_thermistor.Thermistor(board.TEMPERATURE, 10000, 10000, 25, 3950)

in_range = adafruit_debouncer.Debouncer(lambda: thermistor.temperature <= 30)

while True:
    in_range.update()
    if in_range.fell:
        print('High temperature alert')
    elif in_range.rose:
        print('Temperature ok again')
```

You can use a debouncer and an appropriate lamdba to monitor any sort of threshold like this.

Wrap Up

We can use a debouncer to clean up the transitions of input signals, or any computation that results in a boolean value.

We can also use it to monitor for changes in such a value. As such it's useful as an edge detector even if the value transitions aren't noisy.

Something to keep in mind is that a debouncer will do nothing unless you call its `update()` method frequently and regularly. If your debouncer isn't working, check this first.

Given that, you can use the `value`, `rose`, and `fell` properties to get the current stable value, whether it just went to `True`, or just went to `False`, respectively.