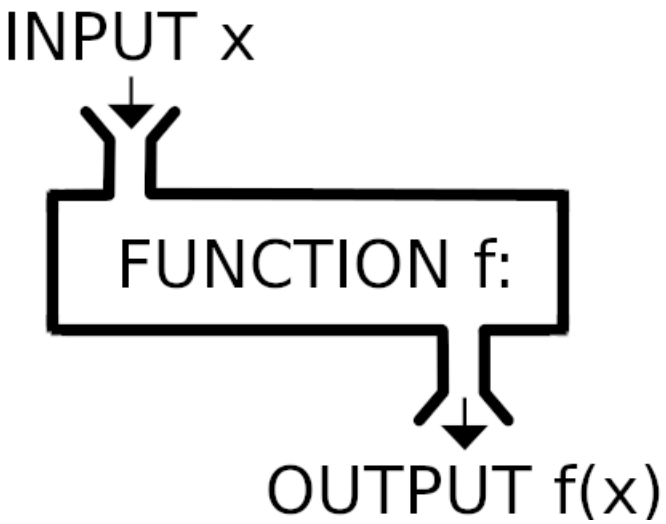
adafruit learning system

CircuitPython 101: Functions

Created by Dave Astels



Last updated on 2018-08-22 04:10:23 PM UTC

Guide 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 (https://adafru.it/cpy-welcome).

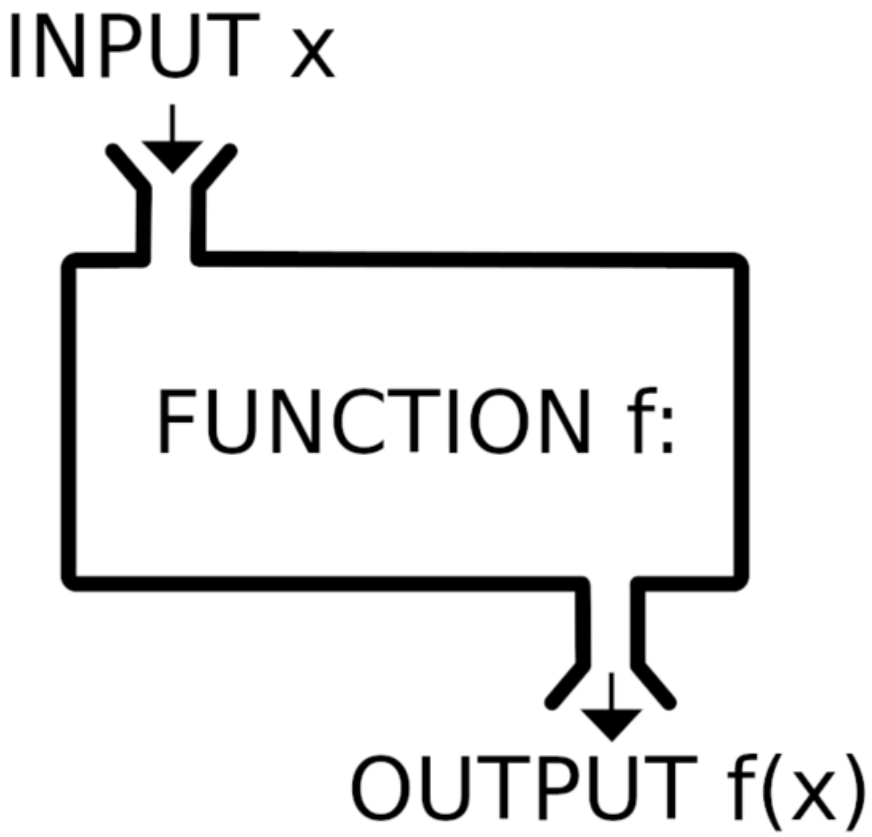
Adafruit suggests using the Mu editor to edit your code and have an interactive REPL in CircuitPython.You can learn about Mu and its installation in this tutorial (https://adafru.it/ANO).

If you've been introduced to Python through CircuitPython, you might not have written many functions yet. With the ATSAMD51 series of boards with its Cortex-M4 core what you can do with CircuitPython expands greatly. That means you can write larger, more complex programs that do more complex and interesting things. That complexity can quickly get out of hand. Functions are a tool to help manage that complexity.

Functions let you package up blocks of code to help make your programs more reusable, shareable, modular, and understandable.

This guide will go over the basics of using functions in CircuitPython.

Function Basics



Simply put, functions are a block a code that is packaged up so as to be executed independently, how and when you choose.

Functions have parameters, and you provide arguments when you execute the function. That's generally referred to as *calling* the function, and the code that calls it is referred to as the *caller*.

The idea of functions in programming goes all the way back to early assembly languages and the concept of the subroutine. Most assembly languages have a `CALL` operation used to transfer control to the subroutine, and a `RET` operation to return control back to the calling code. The BASIC language also used the term subroutine and had a `GOSUB` statement that was used to call them, and a `RETURN` statement.

Most modern languages have dropped the explicit `CALL` or `GOSUB` statement in favor of implicit syntax to indicate a function call:

```
function_name(arg1, arg2, ..., argn)
```

In assembly and BASIC, a subroutine had to end with an explicit `RET` or `RETURN`. Python loosens that up somewhat. If there is no value that needs to be sent back as the result of the function, the return can be left implicit, as in the following:

```
def foo(x):
    print("X is {}".format(x))
```

```
>>> foo(5)
x is 5
```

We can use an explicit return, but it gains nothing and takes an additional line of code.

```
def foo(x):
    print("X is {}".format(x))
    return
```

There are two cases in Python where we do need to make the return explicit.

1. When you need to return from the function before it naturally reaches the end.
2. There is a value (or values) that needs to be sent back to the caller.

Guard Clauses

Below is a trivial example of the first case.

```
def foo(x):
    if x > 5:
      return
    print("X is {}".format(x))
```

```
>>> foo(4)
X is 4
>>> foo(10)
```

Using an `if/return` combination like this is often referred to as a *guard clause*. The idea being that it gaurds entry into the body of the function much like security at an airport: you have to get past inspection before being allowed in. If you fail any of the checks, you get tossed out immediately.

Preconditions

This is also something called a *precondition*, although that usually implies a stronger response than simply returning early. Raising an exception, for example. You use a precondition when the "bad" argument value should never be sent into the function, indicating a programming error somewhere. This can be done using `assert` instead of the `if/return`. This will raise an `AssertionError` if the condition results in `False`.

```
def foo(x):
    assert x <= 5, "x can not be greater than 5"
    print("X is {}".format(x))
```

```
>>> foo(10)
Traceback (most recent call last):
File "", line 1, in
File "", line 2, in foo
AssertionError: x can not be greater than 5
```

Since preconditions are meant to catch programming errors, they should only ever raise an exception while you are working on the code. By the time it's finished, you should have fixed all the problems. Preconditions provide a nice way of helping make sure you have.

Although `assert` is supported, it's a little different in the context of CircuitPython. Your running project likely does not have a terminal connected. This means that you have no way to see that an assert triggered, or why. The CircuitPython runtime gets around this for runtime errors by flashing out a code on the onboard NeoPixel/DotStar. This lets you know there's a problem and you can connect to the board and see the error output. But as I said, by the time you get to that point, there should be no programming errors remaining to cause unexpected exceptions.

That's the key difference between things like guards and preconditions: your code should check for, and deal with, problems that legitimately could happen; things that could be expected. There's no way to handle situations that should never happen; they indicate that somewhere there's some incorrect code. All you can do is figure out what's wrong and fix it.

Returning a Result

The second case of requiring a `return` statement is when the function has to return a value to the caller. For example:

```
def the_answer():
    return 42
```

So what if you have a function that is expected to return a value and you want to use a guard? There is no universal answer to this, but there is often some value that can be used to indicate that it wasn't appropriate to execute the function.

For example, if a function returns a natural number (a positive integer, and sometimes zero depending on which definition you use), you could return a -1 to indicate that a guard caused an early return. A better solution is to return some sort of default value, maybe 0. This, also, is very situation dependant.

Multiple Return Values

Not only can functions return a value, they can return more than one. This is done by listing the values after the return keyword, separated by commas.

```
def double_and_square(x):
    return x+x, x*x
```

A function that returns multiple values actually returns a tuple containing those values:

```
>>> double_and_square(5)
(10, 25)

>>> type(double_and_square(5))
<class 'tuple'>
```

You can then use Python's parallel assignment to extract the values from the tuple into separate variables

```
>>> d, s = double_and_square(5)
>>> d
10
>>> s
25
```

Defining Functions

As you can gather from the examples above, you use the `def` keyword to define a function. The general form is:

```
def function_name (parameter_name_1, ..., parameter_name_n):
    statement
    ...
    statement
```

You can then call the function by using the name and providing arguments:

```
function_name(argument_1, ..., argument_n)
```

This much we can see in the examples. Note that a function doesn't require parameters and arguments If it does have some, those names are available inside the function, but not beyond it. We say that the *scope* of the parameters is the body of the function. The scope of a name is the part of the program where it is usable.

If you use a name outside of its scope, CircuitPython will raise a `NameError`.

```
>>> foo
Traceback (most recent call last):
File "", line 1, in
NameError: name 'foo' is not defined
```

Choosing good names for your functions is very important. They should concisely communicate to someone reading your code exactly what the function does. There's an old programmer joke that the person trying to read and understand your code will quite likely be you in a couple months. This becomes even more crucial if you share your code and others will be reading it.

> You don't want to name a function the same name as a Python command, the name of a library function, or any other name that might be confusing to others reading your code.

Default Arguments

When a function is called, it must be provided with an argument for each parameter. This is generally done as part of the function call, but Python provides a way to specify default arguments: follow the parameter name by an equals and the value to be used if the function call doesn't provide one.

```
def foo(x, msg=''):
    print("X is {}. {}".format(x, msg))
```

```
>>> foo(5, 'hi')
X is 5. hi
>>> foo(5)
X is 5.
```

We can see that if we provide an argument for the `msg` parameter, that is the value that will be used. If we don't provide an argument for it, the default value specified when the function was defined will be used.

Keyword Arguments

So far the examples have been using what's called *positional arguments*. That just means that arguments are matched to parameters by their positions: the first argument is used as the value of the first parameter, the second argument is used as the value of the second parameter, and so on. This is the standard and is used by pretty much every language that uses this style of function definition/call.

Python provides something else: *keword arguments* (sometimes called *named arguments*). These let you associate an argument with a specific parameter, regardless of it's position in the argument list. You name an argument by prefixing the parameter name and an equals sign. Using the previous function `foo`:

```
>>> foo(5, msg="hi")
X is 5. hi
>>> foo(msg='hi', x=5)
X is 5. hi
```

Notice that by naming arguments their order can be changed. This can be useful to draw attention to arguments that are usually later in the list. There's one limitation: any (and all) positional arguments have to be before any keyword arguments:

```
>>> foo(msg='hi', 5)
File "", line 1
SyntaxError: positional argument follows keyword argument
```

Even without changing the order of arguments, keywords arguments lets us skip arguments that have default values and only provides the ones that are meaningful for the call. Finally, keyword arguments put labels on the arguments, and if the parameters are well named it's like attaching documentation to the arguments. Trey Hunner has a great write-up on the topic (https://adafru.it/C2m). To pull an example from there, consider this call:

```
GzipFile(None, 'wt', 9, output_file)
```

What's all this? It's dealing with a file so `'wt'` is probably the mode (write and truncate). The `output_file` argument is clearly the file to write to, assuming it's been well named. Even then, it could be a string containing the name of the

output file. `None` and `9`, however, are pretty vague. Much clearer is a version using keyword arguments:

```
GzipFile(fileobj=output_file, mode='wt', compresslevel=9)
```

Here it's clear that `output_file` is the file object, and not the name. `'wt'` is, indeed, the mode. And `9` is the compression level.

This is also a prime example of the problems using numeric constant. What is compression level 9? Is it 9/9, 9/10, or 9/100? Making things like this named constants removes a lot of ambiguity and misdirection.

Functions as Data

What `def` really does

```
def function_name (parameter_name_1, ..., parameter_name_n):
    statement
    ...
    statement
```

The def keyword takes the body of the function definition (everything indented beneath the line starting with def) and packages it into a function object that can be evaluated later. It then associates that function object with the `function_name` you provided. It also takes the list of parameter names along with any default values and stores them in the function object.

```
>>> def the_answer():
... return 42
...
>>> the_answer()
42
>>> the_answer
<function>
>>> f = the_answer
>>> f()
42
```

Notice that if we type the name of the function rather than a call to the function at the REPL prompt, the system tells us that it is a `function`. Furthermore, notice that we can assign the value of a function (not it's result) to a variable, then use that to call the function.

Creating a function within a function

Remember when we talked about scope: the part of the code in which a name is known and has a value? It so happens that the body of a function is a scope. Things like the function's parameters live in that scope. So do local variables you create (in Python you create local variables simply by assigning a value to a name within the scope). So do functions you define in that scope. Yes, you can create functions that are local to a function.

Here is a function to find square roots. It uses several helper functions. Since these helper functions are specific to the sqrt function, we define them inside it.

```
    def sqrt(x):

    def square(a):
        return a * a

    def good_enough(g):
        return abs(square(g) - x) < 0.001

    def average(a, b):
        return (a + b) / 2

    def improve(g):
        return average(g, x / g)

    guess = 1.0
    while not good_enough(guess):
        guess = improve(guess)
    return guess
```

If you take a minute to understand this code, you might think "Why can't you just write it like this:"

```
def sqrt(x):
    guess = 1.0
    while abs(guess * guess - x) > 0.001:
        guess = (guess + x / guess) / 2
    return guess
```

You could. It works the same. And if you were constrained for space, like on a SAMD21, it could be worth the denser code. But this series is about things you can do on the SAMD51 MCU. You have room to write nicer, cleaner, better code. The first version is easier to read, understand, and tweak as required. That's worth some space, if you can afford it. The first version is simply the second decomposed into meaningful chunks and given names that explain what each one does. Because they are defined inside the `sqrt` function, they don't pollute the namespace with single use functions.

Returning functions

Functions in Python are first class data. One thing that means is that they can be created in another function and returned. That's pretty cool. As an example, consider the following:

```
def make_adder(x):
    def adder(a):
     return a + x
    return adder
```

The function `make_adder` defines and returns a function that has one parameter and returns the sum of it's argument and the argument to `make_adder` when the returned function was defined. Whew! Now we can try it out.

```
>>> inc = make_adder(1)
>>> inc
<closure>
>>> inc(3)
4
>>> inc(5)
6
```

That's interesting. The type of the returned function isn't `function`, it's `closure`. A closure is the combination of a function and the scope in which it was defined. In this example, the returned function refers to its creator function's parameter `x`. When the function is executed later (and is a different scope) it still has a hold of the value that `x` had when it was created: 1 in this case.

We can save the returned closure in a variable (we called it `inc`) and call it later just like a regularly defined function. Given the above we can now do:

```
>>> dec = make_adder(-1)
>>> dec(1)
0
>>> dec(5)
4
>>> inc(3)
4
```

The `dec` function is totally separate from `inc`, which continues to work as before. Each time a new function is created and returned by `make_adder` it has a different copy of the scope, and so its own value for `x`.

Functions as Arguments

Let's reconsider the square root function:

```
def sqrt(x):

    def square(a):
        return a * a

    def good_enough(g):
        return abs(square(g) - x) < 0.001

    def average(a, b):
        return (a + b) / 2

    def improve(g):
        return average(g, x / g)

    guess = 1.0
    while not good_enough(guess):
        guess = improve(guess)
    return guess
```

At the core of this is a general algorithm for finding a solution:

```
while not good_enough(guess):
   guess = improve(guess)
```

In English this is "While your guess isn't good enough, make a better guess."

The algorithm doesn't really care what `good_enough` and `improve` mean. Because functions in Python are first class data, they can not only be returned, but also used as arguments. So we can write a general solver:

```
def solver(good_enough, improve, initial_guess):
    def solve(x):
        guess = initial_guess
        while not good_enough(x, guess):
            guess = improve(x, guess)
        return guess
    return solve
```

Now we can use this to create a square root solver given the two functions and an initial guess:

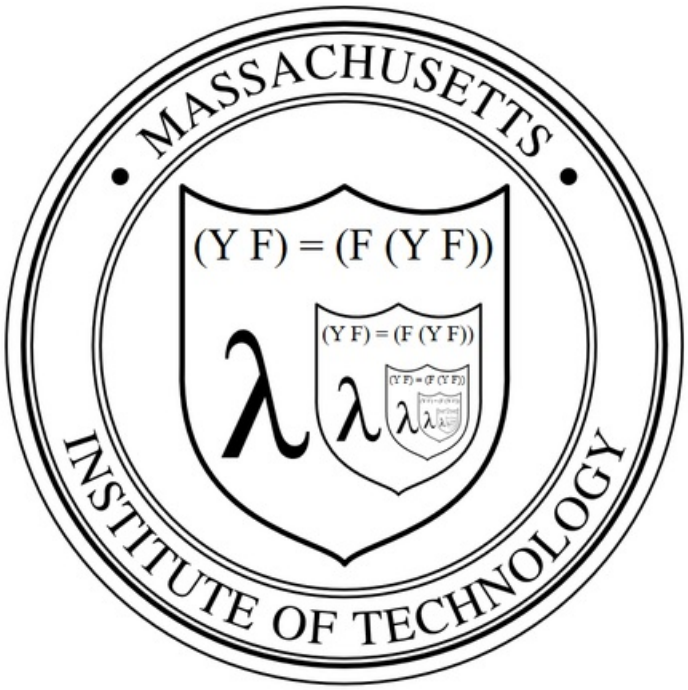
```
def sqrt_good_enough(x, guess):
    def square(a):
        return a * a
    return abs(square(guess) - x) < 0.001

def sqrt_improve(x, guess):
    def average(a, b):
        return (a + b) / 2
    return average(guess, x / guess)
```

Putting it all together:

```
sqrt = solver(sqrt_good_enough, sqrt_improve, 1.0)
>>> sqrt(25)
5.00002
```

The Function With No Name



Notice that since we've pulled the `good_enough` and `improve` functions out, they're not hidden out of sight any more. Python has another capability that we can use to address this: lambdas.

Lambdas are another great thing that comes from the language Lisp. A lambda is essentially an anonymous function: a function object with no name. You might say *"Then how do you use it if you can't get to it by using it's name?"*That's a valid point, except that lambdas are meant to be throw away functions: used once and discarded so there's seldom a reason to hold onto them. Combine this with the ability to pass functions into other functions and be returned by them, and we have some pretty cool abilities.

The syntax to create a lambda is fairly simple:

```
lambda parameter_1, ..., parameter_n : expression
```

The one limitation of lambdas in Python is that they can contain just a single expression that gets implicitly returned. That's not usually a problem, since it's all that's typically needed.

Let's revisit that last example.

```
def sqrt_good_enough(x, guess):
    def square(a):
        return a * a
    return abs(square(guess) - x) < 0.001

def sqrt_improve(x, guess):
    def average(a, b):
        return (a + b) / 2
    return average(guess, x / guess)
```

We will start by inlining the square and average functions.

```
def sqrt_good_enough(x, guess):
    return abs(guess * guess - x) < 0.001

def sqrt_improve(x, guess):
    return average((guess + x / guess) / 2)
```

So now each of these is the return of a single expression. They can easily be replaced by lambdas:

```
sqrt_good_enough = lambda x, guess: abs(guess * guess - x) < 0.001

sqrt_improve = lambda x, guess: (guess + x / guess) / 2
```

This creates two lambdas and assigns them to variables that have the name we used for the original functions. Now we can do the same as we did before:

```
sqrt = solver(sqrt_good_enough, sqrt_improve, 1.0)
>>> sqrt(25)
5.00002
```

However, we can do one better. If we don't need to save these lambdas (and as I said, we seldom do), we can just create them and pass them directly into `solver`:

```
sqrt = solver(lambda x, guess: abs(guess * guess - x) < 0.001,
              lambda x, guess: (guess + x / guess) / 2,
              1.0)
>>> sqrt(25)
5.00002
```

In a similar vein, the `make_adder` function from before can be drastically simplified by returning a lambda:

```
def make_adder(x):
    return lambda a: a + x
```

Something Hardware Related

This example is from a project for an upcoming guide (at the time of writing this) so not all the code is finished or available. But enough around this example is to be a good demo.

The project uses a Crickit and makes heavy use of digital IO. How many still isn't certain, but it will be more than the Crickit or the MCU board provides by themselves. That means some Crickit IO will be used as well as some on-board IO. These will be primarily driven by mechanical switches which will need to be debounced.

I ported a debouncer class to CircuitPython for a previous project (https://adafru.it/BIT) that worked directly against an on-board digital input. That wouldn't work for the new project since dealing with the two input sources is done differently. The solution was to generalize the debouncer to work with a boolean function instead of a pin.

Here's the two relevant methods (we'll have a deep dive into CircuitPython's object-oriented features in a later guide):

```
class Debouncer(object):
    ...

    def __init__(self, f, interval=0.010):
        """Make an instance.
           :param function f: the function whose return value is to be debounced
           :param int interval: bounce threshold in seconds (default is 0.010, i.e. 10 milliseconds)
        """
        self.state = 0x00
        self.f = f
        if f():
            self.__set_state(Debouncer.DEBOUNCED_STATE | Debouncer.UNSTABLE_STATE)
        self.previous_time = 0
        if interval is None:
            self.interval = 0.010
        else:
            self.interval = interval

    ...

    def update(self):
        """Update the debouncer state. Must be called before using any of the properties below"""
        now = time.monotonic()
        self.__unset_state(Debouncer.CHANGED_STATE)
        current_state = self.f()
        if current_state != self.__get_state(Debouncer.UNSTABLE_STATE):
            self.previous_time = now
            self.__toggle_state(Debouncer.UNSTABLE_STATE)
        else:
            if now - self.previous_time >= self.interval:
                if current_state != self.__get_state(Debouncer.DEBOUNCED_STATE):
                    self.previous_time = now
                    self.__toggle_state(Debouncer.DEBOUNCED_STATE)
                    self.__set_state(Debouncer.CHANGED_STATE)
```

The relevant thing is the instance variable `f`. It's a function that gets passed in to the `Debouncer` constructor and saved. In the update function/method it's used to get a `boolean` value that is what's being debounced. The function `f` is simply some function that has no parameters and returns a `boolean` value. In the main project code some debouncers get created. Let's have a look at what those functions are that are being passed to the debouncers.

```
def make_onboard_input_debouncer(pin):
    onboard_input = DigitalInOut(pin)
    onboard_input.direction = Direction.INPUT
    onboard_input.pull = Pull.UP
    return Debouncer(lambda : onboard_input.value)


def make_criket_signal_debouncer(pin):
    ss_input = DigitalIO(ss, pin)
    ss_input.switch_to_input(pull=Pull.UP)
    return Debouncer(lambda : ss_input.value)
```

Here we have two functions: one to create a debouncer on an on-board digital pin, and one to create a debouncer on a crickit (aka Seesaw) digital pin. The setup of the pin's direction and pullup is a bit different in each case. Each of these sets up the pin as appropriate and creates (and returns) a debouncer using a lambda that fetches the pin's value. This

is an example of a closure as well: the pin that the lambda fetches the value of is part of the scope (of the respective `make_*_debouncer` function) that goes along with the lambda. As far as the debouncer is concerned, it's a function that returns a `boolean`; it neither knows or cares how what the function does beyond that. It could be applied to any noisy boolean function, for example reading the Z value from an accelerometer to determine whether a robot is level. A Z value greater than 9 m/s^2 would be a good indicator of that*. As the robot moves about, that value won't be steady. Debouncing would clean it up and answer the question "is the robot level" as opposed to "is the robot level at the instant the accelerometer was read". Assuming `sensor` is an accelerometer interface object that is in scope, the code to create that debouncer would be something like ():

```
robot_level = Debouncer(lambda : sensor.accelerometer[2] >= 9.0)
```

*Assuming the robots is on the surface of the Earth where the gravitational constant it ~ 9.8 m/s^2.