adafruit learning system

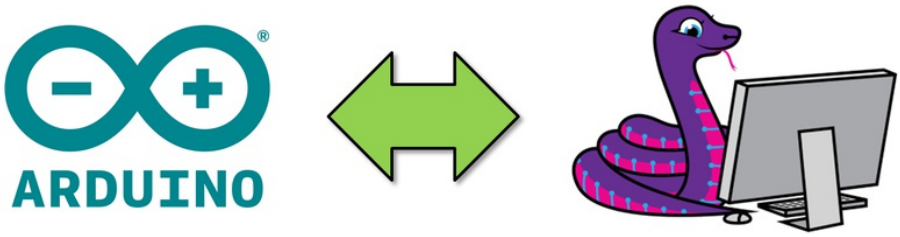
Arduino to CircuitPython

Created by Dave Astels

Last updated on 2018-10-25 05:47:47 PM UTC

Guide Contents

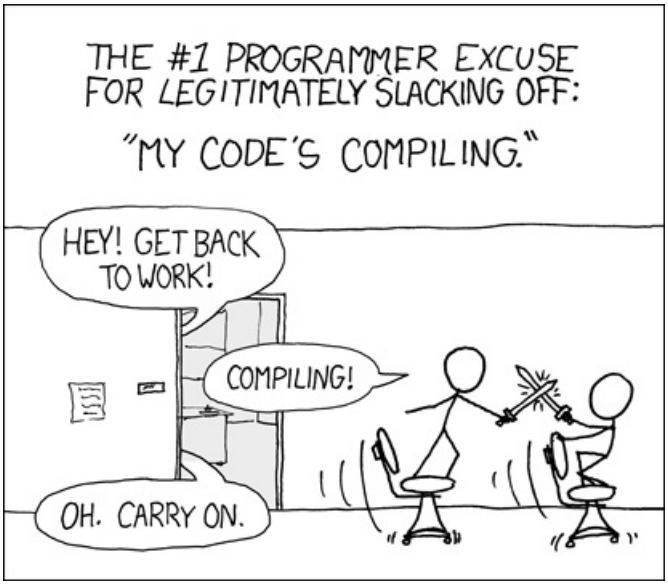
Overview



There are several ways to program microcontrollers. Some methods are more difficult than others and this can often depend on how much familiarity someone has with programming basics.

This guide is primarily for Arduino developers to learn the ins and outs of using CircuitPython by demonstrating use of program code in both languages. This may be a reference for people as they get comfortable in their CircuitPython skills.

Just as valid could be the Python or CircuitPython programmer who has never used Arduino before. Showing the code that Arduino uses for various things may help to become more comfortable with the Arduino environment.



Interpreted vs. Compiled

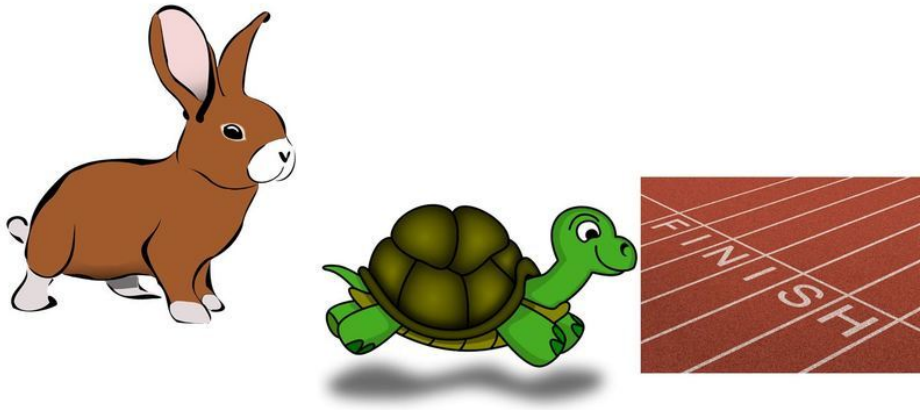
Arduino takes a great deal from tools that have been in use for decades. This includes the concept of *compiled* code. Code is written in a text editor type environment with no debugging. Then, when commanded, it is fed to a series of programs which go about taking the code and in the end compiling it from C or C++ to the machine language of the microcontroller. With a broad variety of microcontrollers on the market, the machine code is unique to that code and that processor.

If the code needs changing in Arduino, the text code must be edited and submitted back through the compile process. Making changes, fixing syntax errors, "dialing in" on optimum values can take a great deal of time in the compile and load processes that happen each time.

Python in general and CircuitPython specifically are *interpreted*. The code is not turned into machine code until it must

be. This has many advantages. The code can give error messages during run time. Any subsequent change does not require recompiling. Loading code is as simple as copying a code text file to a flash drive. CircuitPython program development is often a fraction of the time needed for an Arduino program. The code is also highly portable to other microcontrollers.

The disadvantage to an interpreted code is speed. Converting code to machine code happens on the fly so it takes time. The interpreter also uses RAM and Flash on the microcontroller, more than the equivalent Arduino user code and libraries.



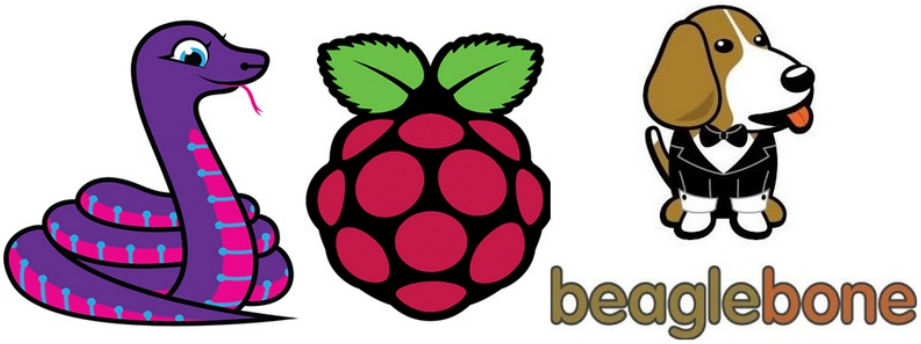
The Reality

Modern microcontrollers are gaining in speed, and memory capacity, often at similar or lower price points than older microcontrollers (which manufacturers will probably want to discontinue at some point). The speed penalty for using Python on microcontrollers is not a concern in modern chips. And the benefits of Python's flexibility and it being taught in schools puts Python in the spotlight.

Providing tools which help migrate Arduino coders to CircuitPython seem particularly appropriate in this age as this is certainly a direction the industry is moving.

You can be slow and still win. And turtles live a very long time.

Computer Python



And what about Linux boards?

Arduino code generally is not run on small Linux boards like Raspberry Pi, BeagleBone, etc. Why?

Linux is a full operating system, which allows many things to be run at once. Arduino code would have to learn to share resources and not use the whole processor, especially in infinite loops. Maybe this functionality will be added one day but it is not available now.

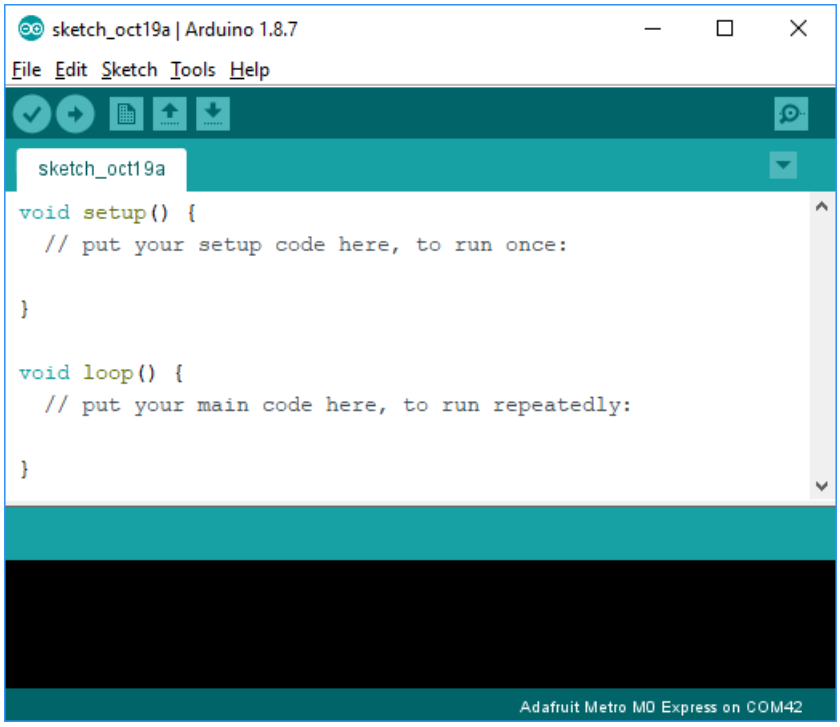
Python is very, very popular on Linux boards. CircuitPython can run on small Linux boards via a helper layer from Adafruit called Blinka. This provides the needed mapping between what CircuitPython might do on a microcontroller and how it may run on Linux. They are designed to play well together (as long as your code is written "well" also.

So for small single board computers based on Linux, CircuitPython is your only choice at present. But it is a good choice as Python and Linux work so well together.

We won't cover the details on using hardware via CircuitPython on Linux in this guide. Check out our guide on Blinka here (https://adafru.it/BSN)

Simple Code Structure

Arduino



The picture above shows the simplest of Arduino programs, what you get when you select **File** -> **New** from the menu. There are two mandatory function calls:

- `setup` - code that will be executed once when the program begins
- `loop` - code that is continuously run in a loop, over & over

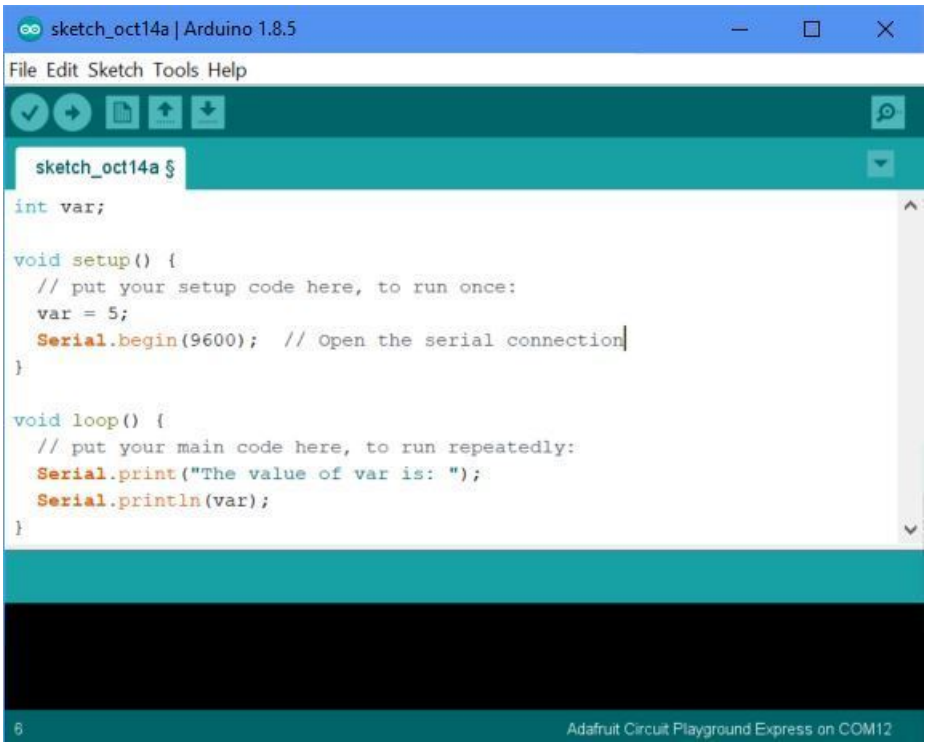
Neither function must do anything. There are sketches that do everything just once in `setup` and some programs that don't use `setup` at all and just use `loop`. But both must be defined, even if they are empty like shown above.

CircuitPython

CircuitPython does not put restrictions on having either code which is executed at the start like the Arduino `setup` function or any sort of repetitive main code like the Arduino `loop` function.

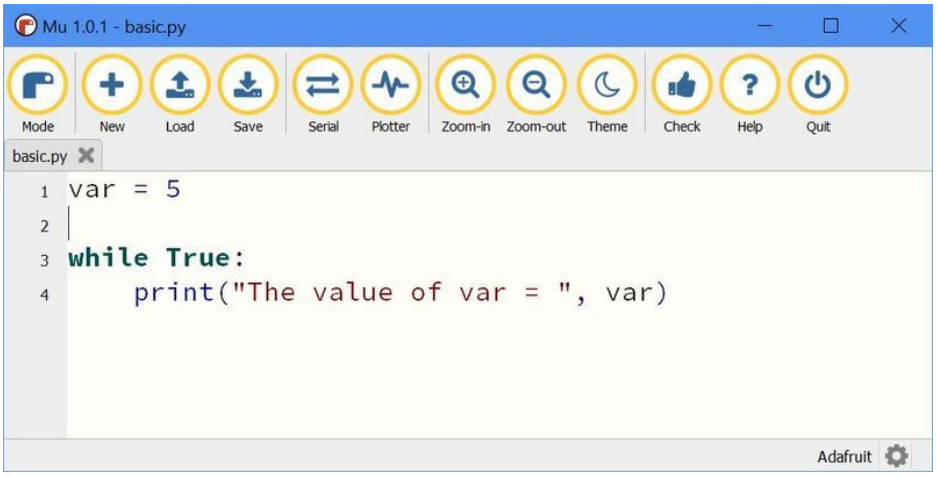
That said, many CircuitPython programs are structured very similarly to how an Arduino sketch program is structured.

A simple example. A program that defines a variable `var` to be the value `5` and then prints that value over and over, kinds similar to the old BASIC Hello World programs. In Arduino, you might write this as follows:



The program adds a bit of code to open the serial connection and print the output.

The equivalent CircuitPython program would be:



Any `setup` type statements are placed near the top of the program.

An infinite loop like the Arduino `loop` function can be done in Python via a `while` loop with the condition set to `True` so that it never exits the `while`.

The serial monitor is "baked in" to CircuitPython, the user does not have to set anything up to use it and this will be discussed more in-depth in this guide.

There is no constraint that CircuitPython must do an infinite loop at all. A simple program might read a temperature sensor, print the output and end.

But, both Arduino and CircuitPython run on microcontrollers. In a class, you might want to do something like read and print a temperature value once. If this project were deployed to a farmer's field, it would probably read the temperature over and over, probably so many times a minute or hour. The code should never stop when out in the farmer's field.

So an infinite loop to do certain functions is of great utility and is used in a majority of microcontroller programs. This is why Arduino has simplified things for beginners, to show that you may want to have a `setup` and `loop`. You can do the exact same things in CircuitPython, you just structure your code to provide the same functionality.

Working with Numbers

You would think "numbers would be numbers" in all languages but it truly isn't quite the case. All numbers are internally represented in binary inside the microcontroller/computer but how the programming language provides number support tends to vary from language to language.

Types of Numbers

- Integers
- Floating Point Numbers
- Boolean (true/false)

Often times, you need to know the *precision* - how big or small the number might be to select the best way to use it in a computer program. This makes using numbers a bit trickier but it is fairly easy to learn.

Arduino

In Arduino, you have the following types of variables:

- **int** for an integer, a value without a decimal point. typical ranges for an integer are -32,768 to zero to 32,767. Examples are 279, 1001, 0, -23, -990.
- **long** is a large integer and can be a value from -2,147,483,648 to 2,147,483,647.
- **float** for floating point numbers (numbers with a decimal point and fractional amount). Examples are 3.1415, -22.2, 0.0, 430000.01. Numbers can be as large as 3 x 10 to the 38th power
- **char** for a single character. For example, reading serial data may involve a receive function providing a character value when data is received. A character may basically be any symbol on the keyboard (0 to 255).

The **types.h** library provides a more modern representation of numbers that tend to behave the same on different devices.

An unsigned short integer, **uint8_t**, is often used by functions, it also ranges from 0 to 255.

Boolean math is generally done with **int** values. 0 is **false** and anything that is not zero, such as 1, is **true**.

Python / CircuitPython

CircuitPython tries to follow the standard Python implementation (often called CPython) as close as possible. Given microcontrollers do not have an operating system, some things are done a bit differently, but these are documented.

Here are the current types in CircuitPython:

- **Integers,** but not currently long integers
- **Floating-Point Numbers**.
- **Strings**
- **Boolean**

Boolean is an actual type in CircuitPython and can be the values **True** or **False**.

Rather than use arrays of characters (null/zero terminated) as Arduino does, CircuitPython has a dedicated **Strings** type. Note that CircuitPython **Strings** are NOT null/zero terminated, so use of the length properties is how you work with string length.

Changing the Type of Numbers

In Arduino/C this is called casting. Place the type of the number you want in parenthesis before the variable.

```
int a;
float b;

b = (float)a;
a = (int)b;
```

In CircuitPython, you use function-like syntax:

```
x = 5
y = 3.14

z = float(x)
z = int(y)
```

Division, One Slash vs. Two

A single forward slash / is floating point division in both languages.

A double slash // in Python is special. It divides and drops any values past the decimal point, often called a floor function.

Example:

```
# Python code for / and // operators
x = 15
y = 4

# Output: x / y = 3.75
print('x / y =', x/y)

# Output: x // y = 3
print('x // y =', x//y)
```

Logical Operators

Logical operators are used mainly in `if` statements and other block / loop statements. These are NOT the operators used for bitwise and/or/not, only for constructing logical comparisons.

	Arduino / C	**Python / CircuitPython**
AND	&&	and
OR	||	or
NOT	!	not

```
// Arduino / C Logical Operators

a = 5;
b = 7;

if( a > 2 && b < 10) {
  Serial.println("Success");
}
```

```
# CircuitPython Logical Operators

a = 5
b = 7

if a > 2 and b < 10:
    print("Success")
```

Variables, types, scope

Quick reference

Variables

Arduino

```
int a;
int b = 5;
```

CircuitPython

```
b = 5
```

Arrays

Arduino

```
int a[5];
int a = {1, 2, 3, 4, 5};
```

CircuitPython

```
a = [1, 2, 3, 4, 5]
```

Discussion

Variables

Variables are like little boxes or pigeonholes in computer memory in which you can keep values.

To create a variable in C/C++ you must declare it. That entails giving it a type, a name, and optionally a value.

```
int a;
int b = 5;
```

In CircuitPython you don't declare variables. They get allocated the first time you assign a value.

```
b = 5
```

You use the value of a variable in both languages simply by using the name.

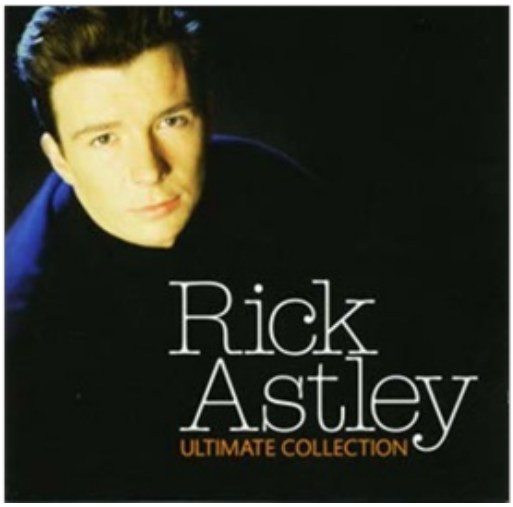
```
int x = a_number;    // in C/C++
x = a_number         # in CircuitPython
```

In both languages, it is an error to use a variable that isn't known: either it hasn't been declared (in C/C++) or initialized (in CircuitPython)

In C/C++, variables are declared with a type, and are statically typed. They only have values of that type.

In contrast, CircuitPython is dynamically typed. Variables do not have an associated type, and can have values of different types at different times. There are advantages and disadvantages to each approach, but in reading someone's CircuitPython code, you should remember this in case someone reused a variable and changed its type.

Collections



Both languages have a way to create and manipulate collections of values.

"Ultimate" - Better than any other collection. Just don't click it.
(image from wikipedia, under fair use license)

Arduino

Arduino has only one way to create collections as part of the language: arrays. Since the language is C++, you are able to use most C++ libraries. With enough CPU performance and memory you can even make use of an Arduino version of the standard C++ (https://adafru.it/COS) and Boost (https://adafru.it/COT) libraries with their collection classes, but for the purposes of this discussion we'll assume you are using the basic language features.

An array is simple a fixed size, fixed order sequence of values. All values in a C array have the same type. In fact, since

C is statically typed, the type of the array items is part of the type of the array itself.

For example, if you want to create an array that can hold 5 `int` values in the variable `a`, you would declare it as:

```
int a[5];
```

If the initial values are known at compile-time, you can initialize the array when it's created, and omit the size as it will be inferred from the number of initial values.

```
int a[] = {1, 2, 3, 4, 5};
```

To access a specific item in an array you use a subscript notation. This can be used to fetch a value out of the array, or to set it to a new value.

```
int x = a[0];
a[1] = x + 1;
```

CircuitPython

The basic CircuitPython collection is the List, which looks and works a lot like C's array. However, it's a far more dynamic structure: you can freely remove and insert items. It's length is a function of how many items are currently in it, not what it was created to hold. Since they are dynamic, you generally don't need to allocate them in advance. You create a list by enclosing the items in square brackets.

```
>>> a = [1, 2, 3, 4, 5]
```

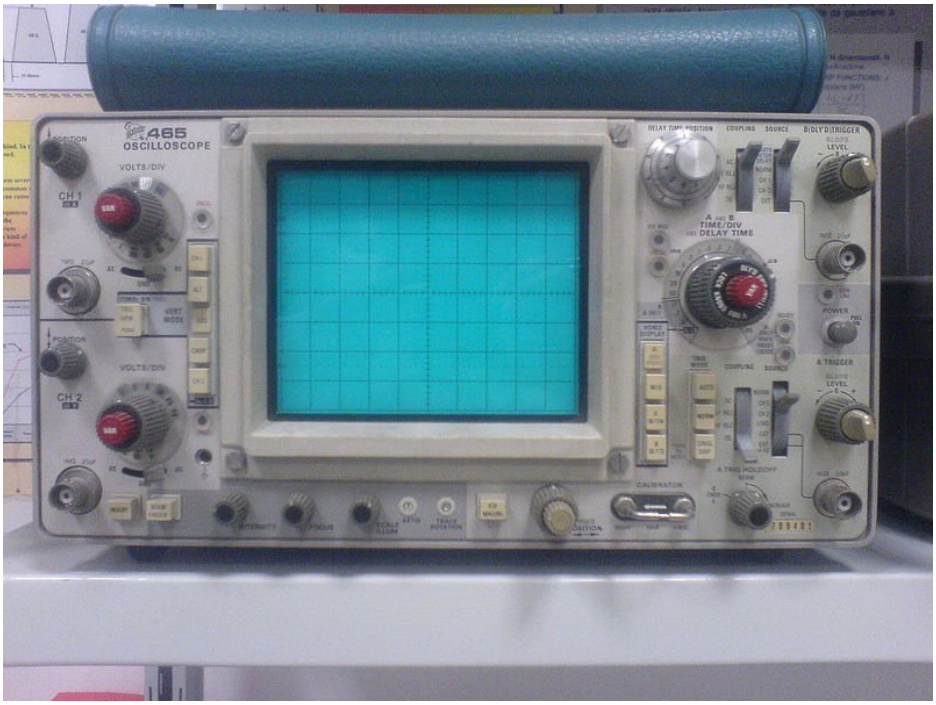
As with C arrays, you use a subscript notation to access items.

```
x = a[0]
a[1] = x + 1
```

CircuitPython's list provides far more functionallity than C arrays, but that's beyond the scope of this guide. CircuitPython also has additional builtin collections: tuples and dictionaries. See this guide on Python's basic data structures (https://adafru.it/Czs) and this one more focused on lists and streams (https://adafru.it/COU).

> Both Arduino/C and Python both start groups of elements/arrays at 0. So for a 3 element item, it is x[0], x[1], x[2] unlike some languages like FORTRAN that start arrays at element 1.

Scope



The scope of a variable is where and when it is available in the code. Both C/C++ and CircuitPython are lexically scoped. That means that a variable's scope is defined by the structure of the code. If you create a variable in a function, it is available in that function but not outside.

```
void f() {
  int a = 5;
  print(a);    // 1
}

print(a);      // 2
```

The line at 1 is valid since `a` is *in scope*. I.e. it's defined in the function `f` and is available to be used. The line at 2, however, will cause a compile error because the name `a` is not known at that point. The situation is similar in CircuitPython:

```
def f():
  a = 5
  print(a)    # 1

print(a)      # 2
```

Local and Global

The code above illustrates local state. I.e. local to a function. That's why referencing the variable from outside the function was a problem. Another scope is global scope. Things with global scope are available throughout the code.

```
int b = 1;

void f() {
  int a = 5;
  print(a + b);
}

print(b);

// prints:
// 6
// 1
```

This works because `b` has global scope so it can be used both inside and outside of `f`. The same holds in CircuitPython.

```
b = 1

def f():
  a = 5
  print(a + b)

print(b)

# prints:
# 6
# 1
```

What if we have variables with the same name, but in both scopes. Now things start differing. In C/C++ there is no trouble. The variable with the most local scope is the one used.

```
int a = 1;

void f() {
  int a = 5;
  print(a);
}

print(a);

// prints:
// 5
// 1
```

However, things work quite differently in CircuitPython.

```
a = 1

def f():
  a = 5      # 1
  print(a)

print(a)

# prints:
# 5
# 1
```

The assignment of `a` is potentially a problem. This is because there is no variable in the local scope named "a", so one will be created because this is the first assignment to `a` in this scope. For the rest of the function, this local variable `a` will be used and the variable `a` in the global scope will be untouched (and unused).

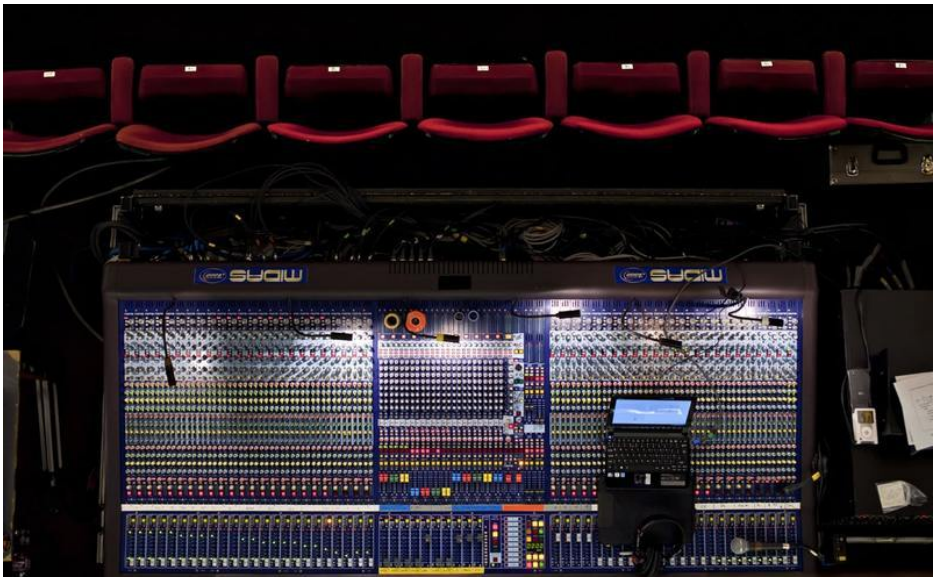
If that's what you want, good. However it's possible that your intent was to change the value of the global `a` to be 5. If that's the case, then you have a problem. A tool like pylint will alert you to the fact that the name "a" from an outer scope (the global scope in this case) is being redefined. That will at least draw your attention to it. If you did want to be using the global `a`, you can tell CircuitPython that is your intent by using the `global` statement.

```
a = 1

def f():
  global a
  a = 5      # 1
  print(a)

print(a)

# prints:
# 5
# 5
```

Now pylint might warn you that you're using a global statement. This is because global variables are generally frowned upon. But in a simple script that's fine [in this author's opinion]. In a larger, more structured program they have less use and you have ways to avoid them.

An interesting capability of CircuitPython and C/C++ doesn't share is being able to make closures. This is a somewhat more advanced feature and has been covered in another guide (https://adafru.it/Czt), so find out more there.

Digital In/Out



Part of programming microcontrontrollers is working with I/O. The most basic form of I/O is digital I/O pins. These are known as GPIO pins (**G**eneral **P**urpose **I**nput **O**utput)

Quick Reference

Configuring a pin for output

Arduino

```
pinMode(13, OUTPUT);
```

CircuitPython

```
import digitalio
import board
led = digitalio.DigitalInOut(board.D13)
led.direction = digitalio.Direction.OUTPUT
```

Configuring a pin for input without pullup

Arduino

```
pinMode(13, INPUT);
```

CircuitPython

```
import digitalio
import board
button_a = digitalio.DigitalInOut(board.BUTTON_A)
button_a.direction = digitalio.Direction.INPUT
```

Configuring a pin for input with a pullup

Arduino

```
pinMode(13, INPUT_PULLUP);
```

CircuitPython

```
import digitalio
import board
button_a = digitalio.DigitalInOut(board.BUTTON_A)
button_a.direction = digitalio.Direction.INPUT
button_a.pull = digitalio.Pull.UP
```

Reading from a pin

Arduino

```
int pinValue = digitalRead(inputPin);
```

CircuitPython

```
pinValue = button_a.value
```

Writing to a pin

Arduino

```
digitalWrite(13, HIGH);
digitalWrite(13, LOW);
```

CircuitPython

```
led.value = True
led.value = False
```

Discussion

Configuring a Digital I/O Pin

Before you can use a pin for input or output, it must be configured. That involves setting it to be input or output, as well as attaching a pullup or pulldown if required.

Arduino

The Arduino framework provides the `pinMode` function for this. It takes two arguments:

1. the pin number being configured. You use this same pin number later to use the I/O line
2. the mode: `INPUT`, `OUTPUT`, or `INPUT_PULLUP`.

To set the standard pin 13 onboard LED to be usable, you would use:

```
pinMode(13, OUTPUT);
```

CircuitPython

CircuitPython

In CircuitPython, it's a little more work. You need create a DigitalInOut instance. To do this you must first import the `digitalio` module. As in the Arduino example, you give it the pin number to use.

In the above Arduino example, how did we know what pin number to use? The short answer is that we just did. In this example, its a convention that the onboard LED is on pin 13. For other GPIO pins you need to check the board you're using to see what pins are available and what one you connected your circuit to.

In Arduino, there's no compile-time check that the pin you've selected is valid or exists. CircuitPython will only let you use pins that the board knows about, which keeps you from making typos.

(If you import the board module (more on importing later) you can use it to list the pins available.)

```
>>> import digitalio
>>> import board
>>> led = digitalio.DigitalInOut(board.D13)
>>> led.direction = digitalio.Direction.OUTPUT
```

If we needed an input we would use `digitalio.Direction.INPUT` and if we needed to set a pullup we do that separately.

```
>>> import digitalio
>>> import board
>>> button_a = digitalio.DigitalInOut(board.BUTTON_A)
>>> button_a.direction = digitalio.Direction.INPUT
>>> button_a.pull = digitalio.Pull.UP
```

The SAMD boards that CircuitPython runs on also support setting a pulldown on input pins. For that you'd use

```
>>> pin.pull = digitalio.Pull.DOWN
```

Using a Digital I/O Pin

Now that you have a pin set up, how do you read and write values?

Arduino

The framework provides the `digitalRead` function that takes the pin number as an argument.

```
int pinValue = digitalRead(inputPin);
```

The value returned from `digitalRead` is of type `int` and will be one of the constants `HIGH` or `LOW`.

The `digitalWrite` function is used to set the pin value. Its arguments are the pin number and the value to set it to: `HIGH` or `LOW`. For example to turn on the LED connected to pin 13, assuming it has been set as an output as above, you use:

```
digitalWrite(13, HIGH);
```

And to turn it off:

```
digitalWrite(13, LOW);
```

The example here shows using a digital output pin to blink the builtin LED: the defacto *Hello World* of microcontroller programming.

```
void setup()
{
  pinMode(13, OUTPUT);          // sets the digital pin 13 as output
}

void loop()
{
  digitalWrite(13, HIGH);       // sets the digital pin 13 on
  delay(1000);                  // waits for a second
  digitalWrite(13, LOW);        // sets the digital pin 13 off
  delay(1000);                  // waits for a second
}
```

Putting digital input and output together, here's an example that reads from a switch and controls an LED in response.

```
int ledPin = 13;   // LED connected to digital pin 13
int inPin = 7;     // pushbutton connected to digital pin 7
int val = 0;       // variable to store the read value

void setup()
{
  pinMode(ledPin, OUTPUT);      // sets the digital pin 13 as output
  pinMode(inPin, INPUT);        // sets the digital pin 7 as input
}

void loop()
{
  val = digitalRead(inPin);     // read the input pin
  digitalWrite(ledPin, val);    // sets the LED to the button's value
}
```

CircuitPython

Assume `led` and `switch` are set up as above.

Input pins can be read by querying their `value` property:

```
>>> button_a.value
```

We can turn the led on and off by setting its `value` property of the `DigitalInOut` object to a boolean value:

```
>>> led.value = True
>>> led.value = False
```

To implement the blinking demo in CircuitPython we would do somethign like:

```
>>> import time
>>> import digitalio
>>> import board
>>> led = digitalio.DigitalInOut(board.D13)
>>> led.direction = digitalio.Direction.OUTPUT
>>> while True:
...     led.value = True
...     time.sleep(0.5)
...     led.value = False
...     time.sleep(0.5)
```

Analog Input

Quick Reference

Configuring an Analog Input Pin

Arduino

Nothing required

CircuitPython

```
import board
import analogio
adc = analogio.AnalogIn(board.A0)
```

Using an Analog Input Pin

Arduino

```
int val = analogRead(3);
```

CircuitPython

```
adc.value
```

Discussion

Configuring an Analog Input Pin

Arduino

Nothing special is needed to configure an analog input pin. An analog output pin needs to be configured as output in the same way as a digital output pin. Note that only certain pins are able to be used as analog. Check the documentation for your specific board to find which ones.

CircuitPython

Using an analog pin in CircuitPython is similar to using a digital one.

As before, use the `board` module to give access to the correct pins by name. Additionally, the `analogio` modules gives you the analog I/O classes.

To read analog inputs you need to create an instance of `AnalogIn`:

```
>>> import board
>>> import analogio
>>> adc = analogio.AnalogIn(board.A0)
```

To set up an analog output, you create an `AnalogOut` instance.

```
>>> import board
>>> import analogio
>>> led = analogio.AnalogOut(board.A0)
```

Using an Analog Input Pin

Arduino

Analog I/O is done similarly to digital. Different boards can have different numbers and locations of analog pins. Check your documentation for specifics on what your board has.

```
int val = analogRead(3);
```

The returned value is an int between 0 and 1023, inclusive. This range assumes a 10-bit analog to digital converter. In general the number of bits in the analog to digital converter determines the range, which will be 0 to 2^{bits}-1. E.g. a 12-bit converter will result in values between 0 and 4095. Refer to your board's documentation to find the size of the converter.

CircuitPython

Once you have an `AnalogIn` object as shown above, you just need to get its value property:

```
adc.value
```

In CircuitPython, analog input values that you read as shown above will always be in the range 0 to 65535. This does not mean that there is always a 16-bit converter. Instead, the values read from the converter are mapped to the 16-bit range. This frees you from having to be concerned with the details of the converter on your board.

The `AnalogIn` object gives you a handy method for querying the converter's reference voltage, so if you need to know the actual voltage being measured you can do it simply by using the following code. Keep in mind that he result will vary based on the voltage at the pin as well as the reference voltage.

```
>>> adc.value / 65535 * adc.reference_voltage
3.2998
```

Analog & PWM Output

Quick Reference

Configuring an Analog Output Pin

Arduino

Most boards don't have true analog output.

CircuitPython

```
import board
import analogio
dac = analogio.AnalogOut(board.A1)
```

Using an Analog Output Pin

Arduino

Most boards don't have true analog output.

CircuitPython

```
dac.value = 32767
```

Configuring a PWM Output Pin

Arduino

Nothing required

CircuitPython

```
import board
import pulseio
led = pulseio.PWMOut(board.A1)
```

Using a PWM Output Pin

Arduino

```
analogWrite(9, 128);
```

CircuitPython

```
led.duty_cycle = 32767
```

Disscussion

Arduino

Writing to an analog pin is straight forward. This is generally not technically a true analog value, but rather a PWM signal. I.e. the value you are writing sets the duty-cycle of the PWM signal. The range is 0-255, inclusive.
The `analogWrite` is used for this and, like `digitalWrite`, takes the pin and value.

The Arduino DUE has 2 actual analog to Digital converters that output an actual analog voltage rather than a PWM signal.

```
    analogWrite(pin, val);
```

Putting it together, we can read from an analog pin and write to another. The difference is value ranges needs to be accommodated, and that's what the division by 4 accomplishes.

```
int ledPin = 9;      // LED connected to digital pin 9
int analogPin = 3;   // potentiometer connected to analog pin 3
int val = 0;         // variable to store the read value

void setup()
{
  pinMode(ledPin, OUTPUT);   // sets the pin as output
}

void loop()
{
  val = analogRead(analogPin);   // read the input pin
  analogWrite(ledPin, val / 4);  // analogRead values go from 0 to 1023, analogWrite values from 0 to 255
}
```

CircuitPython

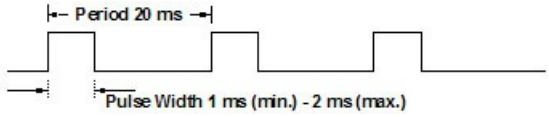
There are two types of analog output available on CircuitPython hardware: true analog and PWM (as on Arduino).

For true analog output, the value parameter of the `AnalogOut` object is set to a value between 0 and 65535, the same range as seen in `AnalogInput`'s value range: 0 sets the output to 0v and 65535 sets it to the reference voltage.

```
dac.value = 32767
```

A related output capability is PWM (Pulse Width Modulation). It's done in a completely different way. Instead of using the `analogio` module, you'll need to use `pulseio` and create a `PWMOut` object using the Pin on which you want to generate the signal.

```
>>> import board
>>> import pulseio
>>> led = pulseio.PWMOut(board.A1)
```



Once you have a PWMOut object, you can set it's duty cycle (the percent of time the output is high) as a 16 bit interger (0-65535). For example:

```
# set the output to 100% (always high)
>>> led.duty_cycle = 65535
# set the output to 0% (always low)
>>> led.duty_cycle = 0
# set the output to 50% (high half the time)
>>> led.duty_cycle = 32767
```

Time

Quick Reference

Delays

Arduino

```
delay(1500);
```

CircuitPython

```
import time
time.sleep(1.5)
```

System Time

Arduino

```
unsigned long now = millis();
```

CircuitPython

```
import time
now = time.monotonic()
```

Discussion

Being able to wait a specific amount of time is important in many projects.

The methods Arduino and CircuitPython handle this are very similar.

Delays

Arduino

The function used to wait a specific time, `delay`, is built into the Arduino framework. No `#include` is required. The single argument is the time to delay in **milliseconds**.

An example is the simple blink LED code shown below:

```
int ledPin = 13;                 // LED connected to digital pin 13

void setup()
{
  pinMode(ledPin, OUTPUT);      // sets the digital pin as output
}

void loop()
{
  digitalWrite(ledPin, HIGH);   // sets the LED on
  delay(1000);                  // waits for a second
  digitalWrite(ledPin, LOW);    // sets the LED off
  delay(1000);                  // waits for a second
}
```

The `delay` function waits one second (1000 milliseconds) each time the `ledPin` is toggled on or off to create a blinking light.

CircuitPython

The equivalent function to `delay` in CircuitPython is the `time.sleep` function. The `time` module is not included by default, it must be imported into the program.

The argument passed into `time.sleep` is the time to delay in **seconds** (not milliseconds). The value can be a decimal so if you wanted to wait one second you'd put in `1.0`, if you wanted 50 milliseconds, it would be `0.050`.

The code below prints the word Hello repeatedly. The prints are spaced by a delay of `0.1` seconds, which is 100 milliseconds.

```
import time

while True:
    print("Hello")
    time.sleep(0.1)
```

Get System Time

In general, microcontrollers and single board computers often lack a Real-Time Clock (RTC) module to know what the time is. What each system generally does have is a count of the time since it was powered on. This is still useful to do certain work that spans a period of time.

Arduino

To get the time since a board has been on in Arduino, the common function is `millis`. It has no arguments and returns the number of milliseconds since the board was last powered on.

The following code sketch prints the time since power on to the serial port every second (1000 milliseconds).

```
unsigned long initial;

void setup(){
   Serial.begin(9600);
   initial = millis();
}

void loop(){
   unsigned long now;
   now = millis();
   if( now - initial > 3 ) { // Print done after 3 milliseconds elapses
      Serial.print("done");
   }
   else {
      initial = now;
   }
}
```

CircuitPython

Python has a number of time functions. But many boards suffer the same issue as Arduino finds - there is no RTC.

CircuitPython has a similar function to the Arduino `delay` called `time.monotonic`. But it returns **seconds**, not milliseconds like `delay`.

Note, it is not recommended to compare times greater than an hour or so as the value will start rounding and could lose time and thus accuracy at that point.

```
import time

initial = time.monotonic()  # Time in seconds since power on

while True:
    now = time.monotonic()
    if now - initial > 0.003:  # If 3 milliseconds elapses
        print("done")
    else:
        initial = now
```

Reference

Arduino

Arduino Reference

- delay (https://adafru.it/COV)
- millis (https://adafru.it/COW)

CircuitPython

- time - time and timing related functions (https://adafru.it/COX)

Libraries and Modules

Quick reference

Arduino

```
include <time.h>
```

CircuitPython

```
import time
```

Discussion

Both Arduino and CircuitPython provide a way to pull in code from outside of the file you are working on. This could be another file you've written, something that's part of the framework, or a sensor support module.

Arduino

C and C++ have code split into code files (that end in .c or .cpp, respectively) and header files (that end in .h). The Arduino environment makes a slight charge to that for the main file of your sketch/program. It ends in .ino.

Header files traditionally contain function and global variable declarations, as well as macro and type definitions. In C++, class definitions go here as well. Code files contain function definitions. Header files provide the interface to libraries, telling your code how to access/call the library's facilities.

To make use of a header file (and the library it is part of) you *include* it:

```
#include <string.h>
```

Now your code can use the functions and types defined in `string.h` which is a collection of string manipulation functions, e.g. `strcpy()` that is used to copy strings.

CircuitPython

All CircuitPython code files have a `.py` extension (ending). There are no separate files defining interfaces.

CircuitPython has a similar mechanism to the Arduino/C include called **modules**. Code creators collect a group of functions together for a specific purpose and create a module.

Precompiled modules have a `.mpy` file extension. Not all modules must be precompiled, it just saves space.

> To use a CircuitPython module, you should place the appropriate files, most often .mpy files, in the /lib folder on the board's flash drive.

Earlier on the Time page, the `time` module was imported and provided two functions: `time.sleep` and `time.monotonic`.

Note that when you import a module as above, you can't just refer to the things in it, you have to prefix them with the module name. I.e. `time.monotonic()`, not just `monotonic()`. There are ways to avoid this, but that's beyond the scope of this guide.

```
import time

while True:
    print("Hello")
    time.sleep(0.1)
```

CircuitPython does not have the capability to import large modules available for full Python/CPython as the memory on microcontrollers is very limited.

Adafruit has committed to making a large number of modules available to support a broad range of hardware. This includes sensors, displays, smart LEDs (NeoPixel, etc.) and much more. Just as Adafruit is a leader in providing Open Source Arduino libraries, Adafruit is striving to do the same in the Python world.

There are more features to Python Modules but the above covers the basics. See a Python reference such as this one (https://adafru.it/COY) to learn more.

Refer to the Adafruit GitHub repository (https://adafru.it/aYH) for the latest CircuitPython modules available.

The board Module

You are writing embedded code on a microcontroller board. That almost always means that you're interacting with the outside world through some of the board's pins. How do you know which ones do what, and which ones to use in your code?

Arduino

Before compiling in the Arduino IDE, you select which board you're using. That basically tells the compiler information it needs to compile specifically for the MCU and configuration of that specific board. That doesn't help with knowing what pins to use, however. You have to look at the documentation for the board and figure it out. The Arduino hardware platform provides some standards that help. You'll always have some pins prefixed with "D". These are for digital I/O. Some will be prefixed with "A". Those can be used for analog signals, but not all support real analog output.

There will be I2C (SCL and SDA) and SPI (MOSI, MISO, and SCK) pins. In your code you'll use the appropriate pin numbers although some boards have predefined values for those pins available when the board is selected in the Arduino IDE.

CircuitPython

CircuitPython takes a very different approach to dealing with the various supported boards. First of all, CircuitPython lives on the board whereas Arduino is a set of tool on your computer that generate a binary file that then lives on the board.

Part of making this work is making versions of CircuitPython that are specific to each supported board. For example, if you want to use CircuitPython with a Feather M4 Express, you get the Feather M4 Express build of CircuitPython and put that onto your Feather M4 Express board. See this guide (https://adafru.it/cpy-welcome) for the details.

The reason this is significant is that CircuitPython knows what board it's running on, and it knows what the capabilities of that board are, and it knows what the pins on that board are and what they can do.

To make this available, there is the `board` module that you can import. This module contains constants for the pins on the specific board. The `board` module in CircuitPython for a different board will have different constants specific to that board. The user does not have to tell CircuitPython what board it is running on, it knows.

All this makes using the board module the safest, most reliable way to use your board's pins. It allows you to not worry about what MCU pins the board pins are connected to. This is more of an issue with more complex MCUs built around ARM cores (such as the SAMD or nRF52 families of MCUs).

For example, I2C signals could be connected to the MCU in various ways (which we will not go into) but using the board module you can simply do:

```
import board
import busio

i2c = busio.I2C(board.SCL, board.SDA)
```

This will work in CircuitPython on any supported board.

So how do you know what pins are available on your board? And what they're called? You can use CircuitPython's `dir` function. This is on a Circuit Playground Express:

```
>>> import board
>>> dir(board)
['A0', 'A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7', 'A8', 'A9',
 'ACCELEROMETER_INTERRUPT', 'ACCELEROMETER_SCL',
 'ACCELEROMETER_SDA', 'BUTTON_A', 'BUTTON_B', 'D0', 'D1',
 'D10', 'D12', 'D13', 'D2', 'D3', 'D4', 'D5', 'D6', 'D7',
 'D8', 'D9', 'I2C', 'IR_PROXIMITY', 'IR_RX', 'IR_TX', 'LIGHT',
 'MICROPHONE_CLOCK', 'MICROPHONE_DATA', 'MISO', 'MOSI',
 'NEOPIXEL', 'REMOTEIN', 'REMOTEOUT', 'RX', 'SCK', 'SCL',
 'SDA', 'SLIDE_SWITCH', 'SPEAKER', 'SPEAKER_ENABLE', 'SPI',
 'TEMPERATURE', 'TX', 'UART']
>>>
```

And this is on a Gemma M0:

```
>>> import board
>>> dir(board)
['A0', 'A1', 'A2', 'APA102_MOSI', 'APA102_SCK', 'D0',
 'D1', 'D13', 'D2', 'I2C', 'L', 'RX', 'SCL', 'SDA', 'SPI',
 'TX', 'UART']
```

You can see that there are some pins in common, but also ones that are board specific.