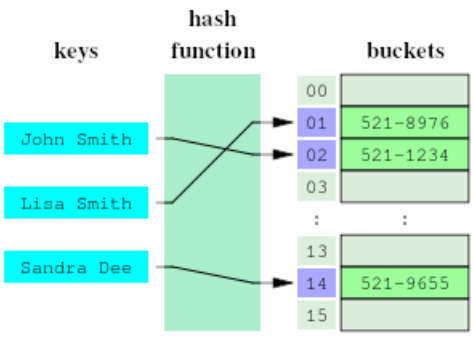
adafruit learning system

CircuitPython 101: Basic Builtin Data Structures

Created by Dave Astels



Last updated on 2019-01-05 08:01:45 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).

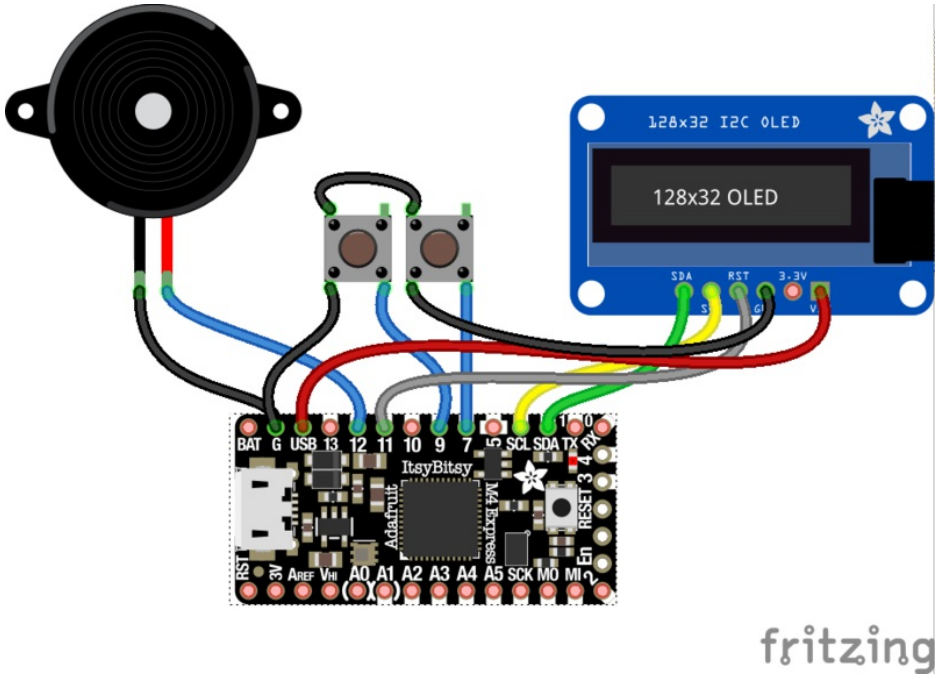
Adafruit suggests using the Mu editor to edit your code and have an interactive REPL in CircuitPython. You can learn about Mu and its installation in this tutorial (https://adafru.it/ANO).

With the introduction of the new SAMD51 (M4) based boards (https://adafru.it/BQG), CircuitPython gets far more interesting. Not only do they have a clock speed almost three times faster than the SAMD21 (M0 boards) and they have six times as much RAM. Not only do CircuitPython programs run significantly faster, 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 this guide, we'll look at several basic ways to organize data in Python: **Tuples**, **Lists**, and **Dictionaries**.

We'll use examples that deal with playing tunes using the Adafruit `pulseio` library. Below is the circuit used featuring an Adafruit ItsyBitsy M4 Express (https://adafru.it/BQC). With a few minimal tweaks to the wiring and code, this will work with any of the M4 Express boards. Note that only the final example makes use of the buttons and an OLED display.



Parts

Any of these M4 boards will do nicely.

1 x ItsyBitsy M4 Express
ATSAMD51 based ItsyBitsy with extra flash.
ADD TO CART

1 x Feather M4 Express
ATSAMD51 based Feather with extra flash.
ADD TO CART

1 x Metro M4 Express
ATSAMD51 based Metro with extra flash.
ADD TO CART

To play the notes, you'll need a buzzer. For the interface you'll need an OLED display and a couple buttons.

1 x Piezo buzzer
Piezo buzzer that you can drive with any frequency square wave.
ADD TO CART

1 x Breadboardable tactile button
Little clicky switches are standard input "buttons" on electronic projects.
ADD TO CART

1 x 128x32 I2C OLED display
Small, but very readable OLED display with an I2C interface.
ADD TO CART

Tools and Materials

- A small solderless breakboard
- wires for use of the breadboard
- microUSB data cable fro connecting the M4 board to your computer

Key image by Jorge Stolfi (https://adafru.it/BQ9) an is CC BY-SA 3.0

Tuple

Tuples are a lightweight way to group information together. They are created using a comma separated sequence of items in parentheses:

```
>>> t = (1, 2, 'three')
```

Notice that there is no need for the parts of a tuple to be the same type of thing. E.g. in the example above we have a tuple with two integers and a string.

Once you have a tuple, you can access the parts of it using an indexing notation:

```
>>> t[0]
1
>>> t[1]
2
>>> t[2]
'three'
```

Notice that tuples (and lists) in Python are 0 based. That is, the index of the first item is 0, the index of the second item is 1, and so on. You can think of this as being how far from the first item you want to access.

Tuples are immutable. That means that once one is created, it can't be changed: you can't add or delete items, or change the values in the tuple.

Let's use tuples to represent notes in a song. Each note has a frequency and a duration. Frequency is in hertz and duration is in seconds. A C4 for quarter of a second would be

```
(261, 0.25)
```

We can use this to write a `play_note` function using `pulseio`:

```
def play_note(note):
    pwm = pulseio.PWMOut(board.D12, duty_cycle = 0, frequency=note[0])
    pwm.duty_cycle = 0x7FFF
    time.sleep(note[1])
    pwm.deinit()
```

From this, we can write some simple code to play notes.

```
import time
import board
import pulseio

C4     = 261
C_SH_4 = 277
D4     = 293
D_SH_4 = 311
E4     = 329
F4     = 349
F_SH_4 = 369
G4     = 392
G_SH_4 = 415
A4     = 440
A_SH_4 = 466
B4     = 493

def play_note(note):
    if note[0] != 0:
        pwm = pulseio.PWMOut(board.D12, duty_cycle = 0, frequency=note[0])
        # Hex 7FFF (binary 0111111111111111) is half of the largest value for a 16-bit int,
        # i.e. 50%
        pwm.duty_cycle = 0x7FFF
    time.sleep(note[1])
    if note[0] != 0:
        pwm.deinit()


a4_quarter = (A4, 0.25)
c4_half = (C4, 0.5)

play_note(a4_quarter)
play_note(c4_half)
```

List

A list in Python is just that: a list of things. We use lists all the time: shopping lists, TO-DO lists. Even Santa uses lists.

Lists in Python have features that jive with our general idea of lists:

- They can be empty.
- They can have any number of things in them.
- They can have different kinds of things in them.
- You can append (add to the end) new things to them.
- You can insert new things anywhere in them (this is easier with lists that aren't written on paper)
- You can sort them to put them in some particular order (if everything in them can be compared).
- You can see how long they are.
- You can remove things from them.
- You can replace things in them.
- You can check if something is in them.
- You can combine them.
- You can throw them out when you're done with them.

Making a list looks a lot like making a tuple, except that square brackets are used:

```
>>> my_list = [1, 2, "three"]
```

In Python, lists and tuples are both a kind of sequence, which means that they have many capabilities in common. For example, you can find their length:

```
>>> len(my_list)
3
```

Accessing items is the same:

```
>>> my_list[0]
1
>>> my_list[1]
2
```

If you use negative indices, they are from the end rather than the start. That makes sense, but -1 is the last item, -2 is the second to last, etc. There really isn't a way around this since -0 isn't really a thing. You can use negative indices with tuples as well as with lists, but it isn't generally as useful. Tuples tend to be small, and for a specific purpose they tend to be the same size with the same type of information in each position. In our tuple example, the frequency was always at position 0, and the duration at position 1.

Lists are more dynamic, both in size and content, unlike tuples.

> You can change the contents/size of lists but not tuples! In exchange, tuples use less memory and are great when you want 'immutable' data

Changing the thing at a specific location of a list is much like accessing whatever is there: you simply give it a new

value.

```
>>> my_list[2] = 42
>>> my_list
[1, 2, 42]
```

Appending new items to a list is easy:

```
>>> my_list.append(3)
>>> my_list
[1, 2, 42, 3]
```

As is inserting something anywhere in the list:

```
>>> my_list.insert(1, 99)
>>> my_list
[1, 99, 2, 42, 3]
```

The first argument to `insert` is where in the list to put the new item, at index 1 in the above example (i.e. the second position). Everything else will get moved to one position larger to make room.

Lists have many more capabilities that we won't consider here. See the python documentation (https://adafru.it/BNw) for more information.

Making songs

We used tuples to form notes that can be played by combining a frequency and a duration, and we wrote a function to pull that information out of a tuple and play the tone on the CircuitPlayground Express' built-in speaker. The next step is to put those individual notes together into songs. Before we can do that we need to make a slight adjustment to the `play_note` function to add rests. We can make the decision to use a frequency of 0 Hertz indicate a rest, or silence:

```
def play_note(note):
    if note[0] != 0:
        pwm = pulseio.PWMOut(board.D12, duty_cycle = 0, frequency=note[0])
        pwm.duty_cycle = 0x7FFF
    time.sleep(note[1])
    if note[0] != 0:
        pwm.deinit()
```

With that we can now construct a list of note tuples.

```
import time
import board
import pulseio

C4     = 261
C_SH_4 = 277
D4     = 293
D_SH_4 = 311
E4     = 329
F4     = 349
F_SH_4 = 369
G4     = 392
G_SH_4 = 415
A4     = 440
A_SH_4 = 466
B4     = 493

twinkle = [(C4, 0.5), (C4, 0.5), (G4, 0.5), (G4, 0.5), (A4, 0.5), (A4, 0.5), (G4, 0.5), (0, 0.5),
           (F4, 0.5), (F4, 0.5), (E4, 0.5), (E4, 0.5), (D4, 0.5), (D4, 0.5), (C4, 0.5)]

def play_note(note):
    if note[0] != 0:
        pwm = pulseio.PWMOut(board.D12, duty_cycle = 0, frequency=note[0])
        # Hex 7FFF (binary 0111111111111111) is half of the largest value for a 16-bit int,
        # i.e. 50%
        pwm.duty_cycle = 0x7FFF
    time.sleep(note[1])
    if note[0] != 0:
        pwm.deinit()

def play_song(song):
    for note in song:
        play_note(note)

play_song(twinkle)
```

Dictionary

Dictionaries allow us to associate a value with a name (generally called a key). It's more general than that, but that's probably the most common use.

A common way to construct a dictionary to use the brace notation:

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d
{'one': 1, 'three': 3, 'two': 2}
```

Notice that dictionaries are not ordered. That's fine, because they don't use numeric indices, they use keys.

As with lists, we can find out how many key-value pairs are in a dictionary using:

```
>>> len(d)
3
```

Accessing data in a dictionary is done similarly to lists:

```
>>> d['three']
3
```

as is changing it:

```
>>> d['three'] = 'not3'
>>> d
{'one': 1, 'three': 'not3', 'two': 2}
```

Adding a key/value pair to a dictionary is the same as modifying one:

```
>>> d['four'] = 4
>>> d
{'four': 4, 'one': 1, 'two': 2, 'three': 'not3'}
```

To remove from a dictionary we use the `del` function:

```
>>> del(d['three'])
>>> d
{'four': 4, 'one': 1, 'two': 2}
```

Finally, we can check if a dictionary contains a specific key:

```
>>> 'one' in d
True
>>> 'three' not in d
True
>>> 'three' in d
False
```

Formats - Songbook

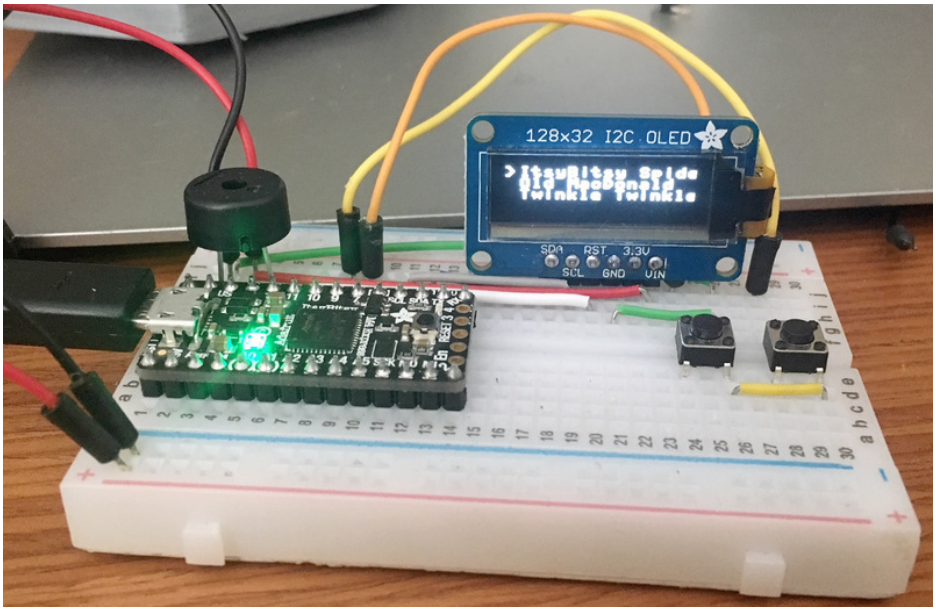
So we have notes represented by tuples that we can put together in a list to make a song. What if we want multiple songs? We could have each in a separate variable. Let's add the additional requirement that we want to connect an I2C OLED display to show a list of songs that we can select from. Having songs in separate variables means that everything has to be hardcoded. That's seldom a good idea. What we'd like is to have things stored in such a way that we just have to add a song and have the menu automatically be updated to reflect it. We can do this by storing them in a dictionary, keyed by name (edited for brevity):

```
songbook = {'Twinkle Twinkle': [(C4, 0.5), (C4, 0.5), (G4, 0.5), (G4, 0.5), (A4, 0.5), ...],

            'ItsyBitsy Spider': [(G4, 0.5), (C4, 0.5), (C4, 0.5), (C4, 0.5), (D4, 0.5), ...],

            'Old MacDonald': [(G4, 0.5), (G4, 0.5), (G4, 0.5), (D4, 0.5), (E4, 0.5), ...]
           }
```

With that done, we can get the names of the songs by `songbook.keys()`. This isn't a list, although it can be used for some list-like things. It can't be indexed, however. We need to convert it to a list in order to be able to do that: `list(songbook.keys())`. While we're at it, we should go ahead and sort it so that it will display in alphabetical order: `sorted(list(songbook.keys()))`.



Here is the complete code:

```
import time
import board
import debouncer
```

```
import busio as io
import digitalio
import pulseio
import adafruit_ssd1306

i2c = io.I2C(board.SCL, board.SDA)
reset_pin = digitalio.DigitalInOut(board.D11)
oled = adafruit_ssd1306.SSD1306_I2C(128, 32, i2c, reset=reset_pin)
button_select = debouncer.Debouncer(board.D7, mode=digitalio.Pull.UP)
button_play = debouncer.Debouncer(board.D9, mode=digitalio.Pull.UP)

C4     = 261
C_SH_4 = 277
D4     = 293
D_SH_4 = 311
E4     = 329
F4     = 349
F_SH_4 = 369
G4     = 392
G_SH_4 = 415
A4     = 440
A_SH_4 = 466
B4     = 493

# pylint: disable=line-too-long
songbook = {'Twinkle Twinkle': [(C4, 0.5), (C4, 0.5), (G4, 0.5), (G4, 0.5), (A4, 0.5), (A4, 0.5), (G4, 1.
                                (F4, 0.5), (F4, 0.5), (E4, 0.5), (E4, 0.5), (D4, 0.5), (D4, 0.5), (C4, 0.
                                (G4, 0.5), (G4, 0.5), (F4, 0.5), (F4, 0.5), (E4, 0.5), (E4, 0.5), (D4, 0.
                                (G4, 0.5), (G4, 0.5), (F4, 0.5), (F4, 0.5), (E4, 0.5), (E4, 0.5), (D4, 0.
                                (C4, 0.5), (C4, 0.5), (G4, 0.5), (G4, 0.5), (A4, 0.5), (A4, 0.5), (G4, 1.
                                (F4, 0.5), (F4, 0.5), (E4, 0.5), (E4, 0.5), (D4, 0.5), (D4, 0.5), (C4, 0.

           'ItsyBitsy Spider': [(G4, 0.5), (C4, 0.5), (C4, 0.5), (C4, 0.5), (D4, 0.5), (E4, 0.5), (E4, 0
                                (E4, 0.5), (E4, 0.5), (F4, 0.5), (G4, 0.5), (G4, 0.5), (F4, 0.5), (E4, 0

           'Old MacDonald': [(G4, 0.5), (G4, 0.5), (G4, 0.5), (D4, 0.5), (E4, 0.5), (E4, 0.5), (D4, 0.5)
                             (B4, 0.5), (B4, 0.5), (A4, 0.5), (A4, 0.5), (G4, 0.5), (0, 0.5),
                             (D4, 0.5), (G4, 0.5), (G4, 0.5), (G4, 0.5), (D4, 0.5), (E4, 0.5), (E4, 0.5)
                             (B4, 0.5), (B4, 0.5), (A4, 0.5), (A4, 0.5), (G4, 0.5), (0, 0.5),
                             (D4, 0.5), (D4, 0.5), (G4, 0.5), (G4, 0.5), (G4, 0.5), (D4, 0.5), (D4, 0.5)
                             (G4, 0.5), (G4, 0.5), (G4, 0.5), (G4, 0.5), (G4, 0.5), (G4, 0.5), (0, 0.5),
                             (G4, 0.5), (G4, 0.5), (G4, 0.5), (G4, 0.5), (G4, 0.5), (G4, 0.5), (0, 0.5),
                             (G4, 0.5), (G4, 0.5), (G4, 0.5), (D4, 0.5), (E4, 0.5), (E4, 0.5), (D4, 0.5)
                             (B4, 0.5), (B4, 0.5), (A4, 0.5), (A4, 0.5), (G4, 0.5), (0, 0.5)]
           }
# pylint: enable=line-too-long

def play_note(note):
    if note[0] != 0:
        pwm = pulseio.PWMOut(board.D12, duty_cycle = 0, frequency=note[0])
        # Hex 7FFF (binary 0111111111111111) is half of the largest value for a 16-bit int,
        # i.e. 50%
        pwm.duty_cycle = 0x7FFF
    time.sleep(note[1])
    if note[0] != 0:
        pwm.deinit()


def play_song(songname):
    for note in songbook[songname]:
```

```
        play_note(note)


def update(songnames, selected):
    oled.fill(0)
    line = 0
    for songname in songnames:
        if line == selected:
            oled.text(">", 0, line * 8)
        oled.text(songname, 10, line * 8)
        line += 1
    oled.show()


selected_song = 0
song_names = sorted(list(songbook.keys()))
while True:
    button_select.update()
    button_play.update()
    update(song_names, selected_song)
    if button_select.fell:
        print("select")
        selected_song = (selected_song + 1) % len(songbook)
    elif button_play.fell:
        print("play")
        play_song(song_names[selected_song])
```

Closing

Tuples, Lists, and Dictionaries provide three different ways to organize information. Choosing the best way to do this can make the difference between a program that is awkward to change and one that is a pleasure to work with.

Next time we'll have a look at functions and some guidelines for writing good ones.