Python Pocket Morse

Setup and user guide

Version 1

Contents

Python Pocket Morse 2
A brief Python primer 2
Step 1 - Connecting 3
Step 2 - Light it up 4
Explanation 4
Result 5
Step 3 - Make the noise 5
Explanation 5
Result 6
Step 4 - Dits and Dahs 6
Explanation 7
Result 7
Step 5 - Make up repetitive beats 7
Explanation 9
Step 6 - Modely going 10
Explanation 12
Conclusion 13

Python Pocket Morse

Why should I try this?

- Maybe you've tried the previous **RSGB Morse on a BBC Micro:bit instructions** and are now wondering how you take the concept further?
- Maybe you're learning Morse and want to practice away from your rig with something that fits in your pocket?
- Or maybe you'd like to do an activity as a club?

The RSGB has appointed Laura Robertson, MM7BFL as its CW Champion and has a renewed focus on bringing attention to the mode. This includes its continuing popularity for SOTA and POTA. Take advantage of this low-cost opportunity to learn some Morse and programming together – including a 'paddle mode', not just a straight key equivalent.

Even if Morse isn't your thing, this is a great way to learn some Python if you don't really know where to start. It's a useful and common programming language within the amateur radio community and this activity is a great first step.

This activity does not expect or require Python or micro:bit knowledge. It isn't intended as an in-depth explanation of the topics included, but more as an example of the capabilities they offer and as a jumping-off point for further experimentation and learning.

What will I do?

This exercise will cover using a micro:bit V2 and the 'more advanced' Python programming mode instead of the previous block programming mode. It is intended as gentle introduction to both Python and small computer programming and can be done without any previous knowledge of the micro:bit.

It will cover setting up your environment, and then basic loops, branches, conditions, methods, and interacting with the micro:bit hardware.

You'll start from a base of 'make a light turn on when I press this button' and continue up to 'paddle' mode for dit-dah input for controlling both speakers and lights at the same time, switching between them as required.

It will take you a couple of hours to follow the guide all the way through. At the end you should be able to continue on, perhaps using the built-in radio to build an actual CW transceiver, or wiring up your own Morse Key or Paddle to get that correct feeling.

What will I need?

You will need a computer that can run a Chrome-based browser (Chrome, Chromium, Edge, and related family), a Micro USB cable, and a micro:bit V2 (you can buy this online for around £16 for a single board, or £18 for a pack with the required cables and optional battery pack).

A brief Python primer

Python is a very common and popular beginners programming language and is used worldwide to teach programming principles. However, do not let that fool you into thinking that it is just for beginners. Python has driven major software projects, space probes, AI and data analysis, and powers some of the largest websites in the world.

For the purposes of this article, we will not be delving too deep into the language itself, but it is useful to know some things before we get started.

1. Python is whitespace sensitive. The indentation really does matter. And you should be consistent if you are using ‘tabs’ or ‘spaces’ to move code around, and how many you use. This is a common cause of frustration for beginners (and experts). A good rule of thumb is that the lines after : should be indented from the line above it
2. Python is case sensitive. While you get a reasonable amount of freedom in what you want to name things, you need to be consistent. ‘Thing’ and ‘thing’ are not the same
3. Python is an object-orientated language. This may not mean anything to you now and object hierarchy is outside the scope of this article, but it is a useful term to know and to investigate in future. Here it is represented by the use of . to call methods on objects
4. The micro:bit uses a reduced form of Python. While everything that we will cover here in terms of the language itself applies outside of the micro:bit environment, if you go further and try more code you have found elsewhere, you may run into some of the limitations

Step 1 - Connecting

Open your browser and go to https://python.microbit.org/v/3, it will open with a default initial program to display a heart.

Plug in your micro:bit and press the ‘Send to micro:bit’ button

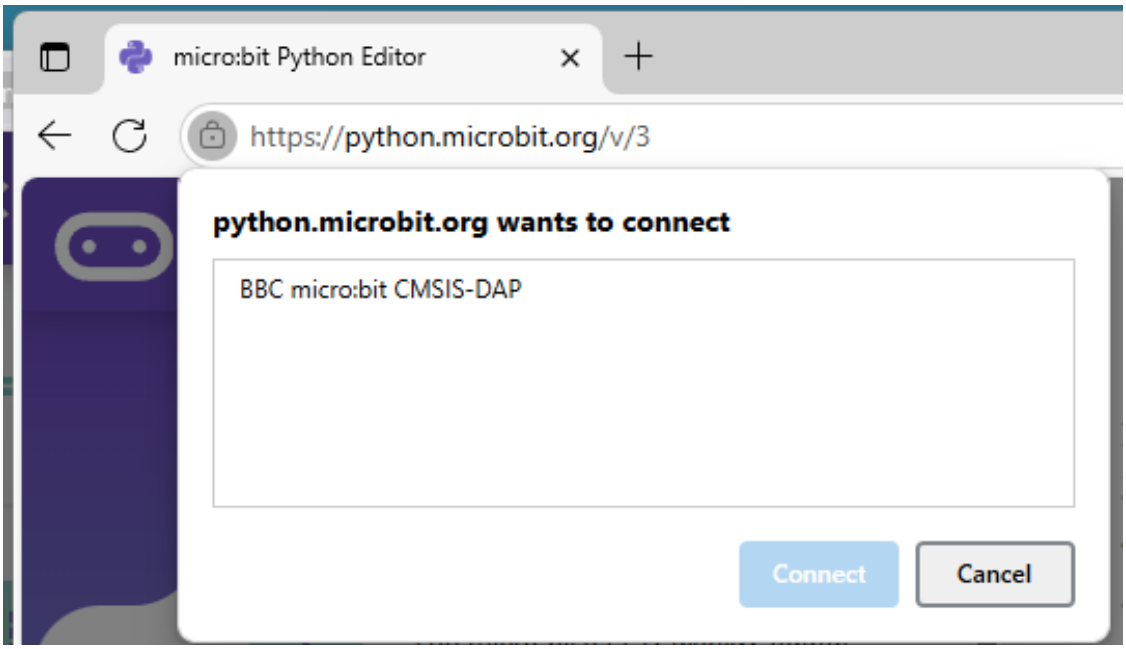

image.png

Select the ‘BBC micro:bit’ option and press ‘Connect’. It can take a couple of minutes the first time you do this while the internal software is updated to the latest version.

Once that’s complete, you should have a heart image, and a scrolling ‘hello’.

And now you’re ready to start writing your own code.

Step 2 - Light it up

So now we've got the sample program working, it's time to do something more useful.

Let's start with 'when I press this button, a light will light up'. Which is the starting point for so many of these types of tutorials for a reason.

Delete the sample code and add this code:

```
# Import from the microbit module
from microbit import display, button_a, Image

# Clear the display
display.clear()

# Code in a 'while True:' loop repeats forever
while True:
    if button_a.is_pressed():
        display.show(Image(
            "00000:"
            "05550:"
            "05550:"
            "05550:"
            "00000:"
        ))
    else:
        display.clear()
```

Explanation

A quick rundown on what this does. Lines that start with a # are comments. They are not part of the code and contain plain text descriptions and notes for yourself:

1. `from microbit import display, button_a, Image` - this adds the code that is in the micro:bit module (or library) to our code so that we can use code that someone has written for us (which is quite handy and saves a lot of time)
2. `display.clear()` - display is one of those imports. We want to make sure it's blank before we draw anything on it. So call the clear() method on it. 3.while True: - We want this to run forever. A 'while' statement takes a condition. In this case we never want to stop, so the condition is always True. The code will loop around this constantly every time it reaches the bottom
3. `if button_a.is_pressed():` - An if statement is a question which uses a condition to establish the answer (just like the while in (3)). button_a is provided by the microbit module, just like display and represents the A button on the board. is_pressed() is a method that is provided that will return True or False depending on the state of the button.
4. `display.show(Image(<numbers>))` - Use the display to show an Image. The numbers here are the representation of the brightness of a particular LED in the 6x6 array on the front of the board. For reference, here Image is a Class. We will not directly cover Class creation in this article, but you can look it up if you're curious.

5. `else` - in (4) we asked a question: 'Is button_a pressed?', and if it is, we display an image. An else is 'what do we do if the answer isn't True?'.
6. `display.clear()` ` - Hopefully you can see what this does from the previous explanations: Use the display provided from the microbit library

Result

And once you've pressed the 'Send to micro:bit' button, if you press button a on the front of the board (the one on the left), you should get a small red square light up on the LEDs.

Now you've got a straight-key equivalent silent Morse Key!

Admittedly, not a very exciting one, but it's more than you had before you started this…

Step 3 - Make the noise

Lights are all well and good, but we all know it's not real until it beeps at you. Dits and Dahs are named that because of the noise, after all.

Let's see about fixing that by adding some sound output as well, using the provided music library.

```
from microbit import display, button_a, Image
# Add the music library
import music

display.clear()

while True:
    if button_a.is_pressed():
        display.show(Image(
            "00000:"
            "05550:"
            "05550:"
            "05550:"
            "00000:"
        ))
        # Play a noise at 600Hz
        music.pitch(600)
    else:
        # Stop the noise
        music.stop()
        display.clear()
```

Explanation

This is very similar to what we had before, as you'd expect. We want all the code that checks if a button has been pressed and does something. But this time we want both 'light up the display' and 'make some noise'.

Let's cover the new bits:

1. `import music` - this is another provided module / library. This time it's the one specifically for playing music or making noises. `2.music.pitch(600)` - use the music library to play a pitch. This pitch method is being given a variable. A variable is literally 'a thing that can change'. We give the variable of 600 in this case, which the pitch method used as the frequency of the pitch to generate. So, we'll get a 600Hz tone. More on variables and methods later
2. `music.stop()` - the same way we want the display to be blank when there isn't a button pressed, we also want the tone to stop

Result

Send that code to the micro:bit and press the button. Don't do this anywhere where a buzzing noise won't be appreciated!

You should get both a square light up and a tone played whenever the button is pressed. Now you can really do some Morse code.

Perhaps you don't like the 600Hz and want a nice Concert A 440Hz. Hopefully you can see what you would change in the code.

Step 4 - Dits and Dahs

Great, so now we've got some noise and we've got some lights. But we don't really have anything that can't be achieved with a battery, and LED, and a buzzer, so in the next few steps we're going to make things more complicated and work towards something that works more like a paddle than a straight key.

Firstly, let's introduce the concept of TWO BUTTONS and how to make them do different things.

```
# Note the new import here
from microbit import display, button_a, button_b, Image
import music

display.clear()

# Code in a 'while True:' loop repeats forever
while True:
    if button_a.is_pressed():
        display.show(Image(
            "00000:"
            "05550:"
            "05550:"
            "05550:"
            "00000:"
        ))
        music.pitch(600)
    # A new construction, ask multiple questions!
    # Also using button_b for the first time
    elif button_b.is_pressed():
        display.show(Image(
            "00000:"
            "00000:"
```

```
            "99999:"
            "00000:"
            "00000:"
        ))
        music.pitch(600)
    else:
        music.stop()
        display.clear()
```

Explanation

This is starting to show how you can build complex behaviour out of simple building blocks of code. Maybe you can start to work out what this does even before the explanation?

Let's go over the new parts

1. `from microbit import display, button_a, button_b, Image` - we need to be able to use the other button, so let's get that from the micro:bit library like we do the other things
2. `elif button_b.is_pressed():` - elif is part of the if/else tree. If asks a question of the condition that follows it. elif does the same, but only when the if statement before it is False. So if button_a is pressed, we'll get the same behaviour as before. But if it is not, and button_b is pressed instead, we'll get the new behaviour. You can add as many elif as you want, but be aware if can get pretty hard to follow the code pretty quickly
3. Note we have a new `Image` declaration in the button_b block. This one looks like a line

Result

Again, let's send this to the micro:bit and see what we've got.

If this has all worked, if you press the A button you get a tone and a square on the display. If you press the B button you get the same tone, and a line on the display.

Maybe you have noticed there's some ambiguity about what you do if both buttons are pressed at once. In this code, the first if statement will be True, the dot will be displayed and button B will basically be ignored. That's probably okay for this for now.

Step 5 - Make up repetitive beats

Okay, hold on to your hats. If you're wearing a hat. If you're not, then just get ready. This is where it gets a bit complicated.

```
# Import the time library
import time

from microbit import display, button_a, button_b, Image
import music

# Variables for how long a dit is
dit_timer = 0
dit_duration = 250

# Variables for how long a dah is
```

```
dah_timer = 0
dah_duration = dit_duration * 3

display.clear()

# Decide which image to display
def display_mode(dit=None, dah=None):
    if dit:
        display.show(Image(
            "00000:"
            "05550:"
            "05550:"
            "05550:"
            "00000:"
        ))
    if dah:
        display.show(Image(
            "00000:"
            "00000:"
            "99999:"
            "00000:"
            "00000:"
        ))


while True:
    # Do nothing if both buttons are pressed
    if button_a.is_pressed() and button_b.is_pressed():
        continue
    elif button_a.is_pressed():
        # Display the correct image
        display_mode(dit=True)
        # Work out how long we've been making a noise
        play_duration = time.ticks_diff(time.ticks_ms(), dit_timer)
        # If we've been playing for less than the duration of a dit
        # then continue playing
        if play_duration // dit_duration == 0:
            music.pitch(440)
        # If we've been playing for longer than a dit, but less than
        # a dit + space length, 'play' the space (nothing)
        elif play_duration // dit_duration < 2:
            music.stop()
        # If we've finished a space, reset the counter so we start
        # playing again
        else:
            dit_timer = time.ticks_ms()

    elif button_b.is_pressed():
        # Display the correct image
        display_mode(dah=True)
```

```
        # Work out how long we've been making a noise
        play_duration = time.ticks_diff(time.ticks_ms(), dah_timer)
        # If we've been playing for less than the duration of a dah
        # then continue playing
        if play_duration // dah_duration == 0:
            music.pitch(600)
        # If we've been playing for longer than a dah, but less than
        # a dah + space length, 'play' the space (nothing)
        elif play_duration // dit_duration < 4:
            music.stop()
        # If we've finished a space, reset the counter so we start
        # playing again
        else:
            dah_timer = time.ticks_ms()
    # If no buttons are pressed, stop all the things
    else:
        dit_timer = time.ticks_ms()
        dah_timer = time.ticks_ms()
        music.stop()
        display.clear()
```

Explanation

Now we have a lot of new code, and it all looks quite complicated. Once you start going through it, however, it's actually not that complex and is just an extension of what we have seen before.

1. `import time` - we need to know how long a dit or a dah should be, so we're going to need some time methods, so get the time library
2. `dit_timer = 0` - A variable to store the time when we started playing a dit
3. `dit_duration = 250` - A variable to store how long a dit should be, in microseconds
4. `dah_timer = 0` - A variable to store the time when we started playing a dah
   `5.dah_duration = dit_duration * 3` - A variable to store how long a dah should be in microseconds, referencing the `dit_duration` but 3 times as long
5. `def display_mode(dit=None, dah=None):` - A new thing! This is declaring our own method. A method is a block of reusable code. When you call a method, you execute this block, as if it was copy and pasted into where you called it from. This method is called `display_mode` and takes two arguments, dit and dah
6. The `display_mode` method simply uses the dit or dah variable to decide if it needs to update the display to the square or line for a dit or dah
7. `if button_a.is_pressed() and button_b.is_pressed():` - Lets solve that problem of what we do if both. continue is a special keyword for if you're inside a loop (remember, we're in a while loop), that just says 'ignore everything else and move to the next iteration
8. `display_mode(dit=True)` - This is where we call our method from (6), and we want a dit to be displayed
9. `play_duration = time.ticks_diff(time.ticks_ms(), dit_timer)` - This looks complicated, so lets break this down. Use the ticks_diff method from the time library, which takes two arguments. time.ticks_ms() gets us the time since we powered on in milliseconds (importantly, this is not the time of day), and then we get the difference

between that and the time we saved earlier. The end result is ‘how long is it since we last ran this line of code?’

10. `if play_duration // dit_duration == 0:` - Now we’re using some Fancy Maths. // is the mod function. Its result is a little strange, but is ‘what is left over if you divide by dit_duration ?’. We then check if that is 0, or play_duration is not divisible by dit_duration entirely. In this case, we play a noise.
11. `elif play_duration // dit_duration < 2:` - More fancy maths. A dit is evenly spaced. So if the mod is less than 2, we are currently in the ‘space’. Remember that this is an elif so we will not be in this code if (11) has already happened
12. `dit_timer = time.ticks_ms()` - If we’ve exceeded the length of the space then reset the counter. This means that on our next while loop, we will match the first condition again. We’re running fast enough here that the extra loop will be unnoticeable on a human level
13. Then do all that again, but with a dah instead
14. `dit_timer = time.ticks_ms() dah_timer = time.ticks_ms()` - If no buttons are pressed, update the start timer to the current time so when a button is pressed we know when. This also sets our ‘initial’ values on the first iteration of the loop

I hope all that makes sense. There are definitely other ways that this could be written to achieve the same end, this particular approach was chosen to showcase some of the more advanced logic you can achieve with just the in-built maths and conditional code.

The end result of this is that you can hold down either A or B and get a repeated dit or dah, emulating a basic paddle key mode.

Step 6 - Modely going

But now we have a problem. You might want to use a straight key, but we’ve made a paddle. You could use the previous code and reprogram the micro:bit every time you want to change mode. But that’s going to get annoying quickly.

So let’s make it so you can change mode whenever you want. And fortunately, the micro:bit V2 has an extra button! The logo is actually a capacitive button and it’s nice and obvious so we can use that to change the mode between ‘straight key’ and ‘paddle’ as we wish.

```
import time

from microbit import *
import music

dit_timer = 0
dit_duration = 250

dah_timer = 0
dah_duration = 250 * 3

# Variables for holding what mode we're in
# and whether we should change mode
mode_pressed = False
single_mode = True
```

```
display.clear()


def display_mode(dit=None, dah=None):
    if dit:
        display.show(Image(
            "00000:"
            "05550:"
            "05550:"
            "05550:"
            "00000:"
        ))
    if dah:
        display.show(Image(
            "00000:"
            "00000:"
            "99999:"
            "00000:"
            "00000:"
        ))


# Handle displaying the mode change
def change_mode(current_mode):
    if current_mode:
        display.show(Image(
            "90009:"
            "99099:"
            "90909:"
            "90009:"
            "90009:"
        ))
    else:
        display.show(Image(
            "99999:"
            "90000:"
            "99999:"
            "00009:"
            "99999:"
        ))
    # Do nothing for a second so we can read the display
    sleep(1000)
    # Invert the mode
    return not current_mode

# Code in a 'while True:' loop repeats forever
while True:
    if button_a.is_pressed() and button_b.is_pressed():
        continue
    elif button_a.is_pressed():
```

```
        display_mode(dit=True)
        if single_mode:
            music.pitch(440)
        else:
            play_duration = time.ticks_diff(time.ticks_ms(), dit_timer)
            if play_duration // dit_duration == 0:
                music.pitch(440)
            elif play_duration // dit_duration < 2:
                music.stop()
            else:
                dit_timer = time.ticks_ms()

    elif button_b.is_pressed():
        if single_mode:
            continue
        display_mode(dah=True)
        play_duration = time.ticks_diff(time.ticks_ms(), dah_timer)
        if play_duration // dah_duration == 0:
            music.pitch(600)
        elif play_duration // dit_duration < 4:
            music.stop()
        else:
            dah_timer = time.ticks_ms()
    else:
        dit_timer = time.ticks_ms()
        dah_timer = time.ticks_ms()
        music.stop()
        display.clear()

    # Record that we've pressed the button if we have
    if pin_logo.is_touched():
        mode_pressed = True
    # Check if we have let go of the button
    if mode_pressed and not pin_logo.is_touched():
        # Record that we've stopped pressing the button
        mode_pressed = False
        # Change the mode
        single_mode = change_mode(single_mode)
```

Explanation

This is basically the same as before, but with some special handling for the middle button. We need to handle this differently as we're not doing something while it is pressed, we need to do something when it has been pressed. So, we need to track the press/not pressed state and do something when it has been released:

1. `mode_pressed = False` - a variable to hold whether we're currently pressing the logo button
2. `single_mode = True` - a variable to hold what mode we're currently in

3. def change_mode(current_mode): - another method to display an Image to tell us what mode we're entering. This method should be straightforward to understand now, with two new parts
4. `sleep(1000)` - this is another inbuilt function. The name is descriptive, we stop operating in the current state for however long we pass as an argument. It's measured in milliseconds, so 1000 will pause execution for 1 second, so we can see what has been displayed
5. `return not current_mode` - sometimes you want to know what the result of a method is. If you've passed in some variables and performed some operations on them, you might want the result in the original code. In our case, we want to update the single_mode variable with the new mode that we are in. not in this case inverts the result, see (6) for more information
6. `if pin_logo.is_touched():` - if the logo is pressed, then record that we have pressed it so we know when we have finished pressing it
7. `if mode_pressed and not pin_logo.is_touched():` - now we have two questions in the if statement. Have we previously pressed the button down and is the button currently not pressed. In this case, this means that the button was pressed last iteration, but now is not pressed, so we should change mode, and update our state record to match.

Once all that is together we've got a way to switch between straight key mode using just Button A and paddle mode using both Button A and B.

Conclusion

There are a lot of details that have been omitted in this tutorial with the aim of getting something done and explaining just enough that you can search for further information on a particular topic if you want to, or just make something work if you don't.

There's a lot of potential for expansion here. The micro:bit even has a transceiver in it, you could potentially turn it into an actual CW transmitter. Or build even more advanced input modes.

Hopefully this is useful to you on your coding journey - making something from a blank page is always exciting.

Get in touch and let us know how you get on, email maker.champion@rsgb.org.uk and tell us!