
THE UNIVERSITY OF
WAIKATO
*Te Whare Wānanga o Waikato*

DEPARTMENT OF
COMPUTER SCIENCE
*Te Tari Rorohiko*

# 2024

Contributors

J. Turner

V. Moxham-Bettridge

J. Bowen

# PROGRAMMING MICRO:BITS WITH PYTHON

This week we are going to use a form of Python (MicroPython) to program Micro:bits. If you have done L1 of CSNeT or attended one of our outreach events before, you may have already encountered Micro:bits and might find this session relatively easy (if that's you, don't worry, there are plenty of advanced exercises to complete). If you haven't used Micro:bits before, also don't worry, as the main exercises cover only what is necessary to get a sense for the basics of embedded programming.

## A quick introduction to MicroPython

Don't stress if you haven't programmed much in Python before, Micro:bits actually utilises a language called MicroPython which features a reduced Python instruction set so it's appropriate for beginners and experts alike. Due to this, MicroPython is very popular in the realm of embedded programming (i.e. microcontrollers/small robots) because of its low requirement for resources as it only needs 16KB of RAM to run (whereas Python requires a minimum specified in MBs). As Micro:bits are microcontrollers, it makes sense to use a language that is very power and space efficient to program them. This 'lightweight' version of Python also makes the language – that is already regarded as one of the best languages to learn[3] – slightly easier to grasp because only the necessary libraries are included for someone to learn to program in Python and to program embedded systems.

## Getting started with Micro:bits

To start programming the Micro:bit you will need to open a web browser (e.g. Chrome) and navigate to microbit.org. Select the “Let’s code” on the black bar near the top of the page. Select the “MakeCode editor” button and create a new project called “Getting started”. Once the project has loaded, you will see an `on start` code block and a `forever` code block on the screen. Instead of using this interface to instruct the Micro:bit, we are going to use the Python code editor which can be selected from the dropdown at the top of the screen (see figure 15).

[3] According to the University of Berkley: https://bootcamp.berkeley.edu/blog/most-in-demand-programming-languages/

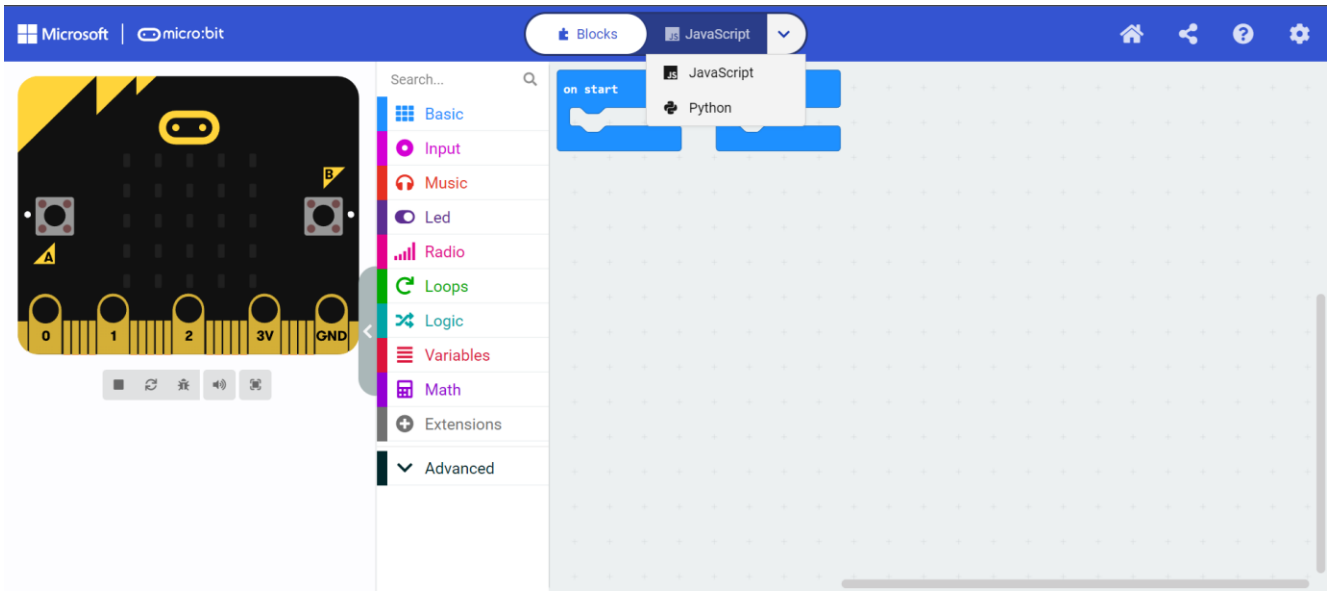


*Figure 15: Selecting the Python code option*

After selecting the Python option, you should see the screen displayed in figure 16.

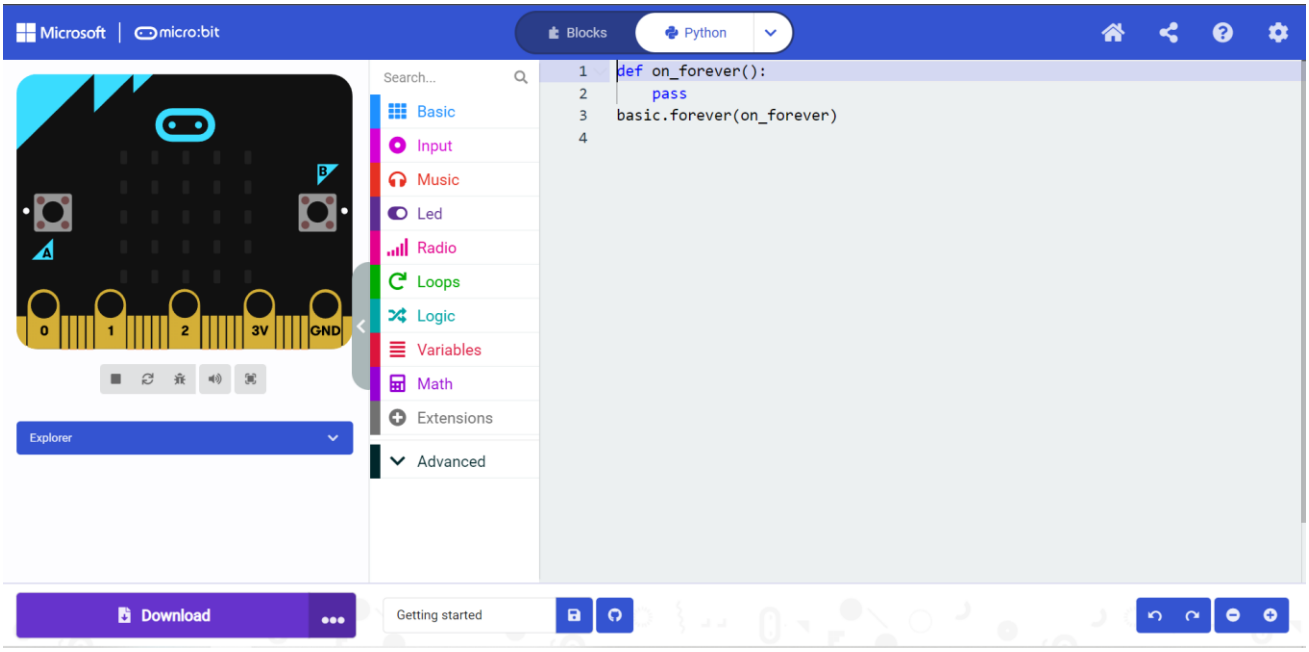


*Figure 16: The Python code initially shown*

Firstly, delete the current code (as we won't be needing it yet) and replace it with '`basic.show_string("Hello world!")`'. You should now see the text, "Hello World"

being displayed on the Micro:bit simulator's LEDs, however you should also notice that none of the LEDs on the physical device are lighting up. This is because we haven't actually instructed the Micro:bit to do so. Download the program to the device by clicking on the 'Download' button and following the prompts (feel free to seek assistance from staff about this).

## Learning the basics

We know how to show a string on the LEDs, but what about an icon? To do this, type 'basic', and as you're typing you should see a popup showing the different statements available (see figure 17).

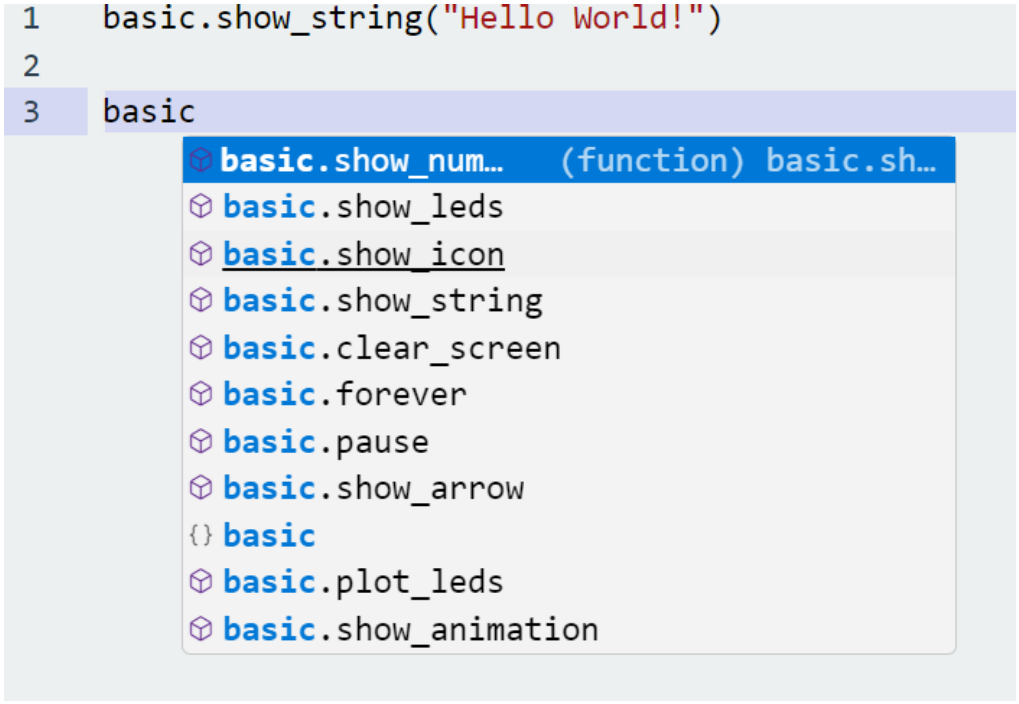


*Figure 17: The popup that displays all available 'basic' methods*

As we want to show an icon, click on the '`basic.show_icon`' option and you should now see the `show_icon` function appear in your code with a heart selected as default. Before you download this to the device, comment out the code on line 1 (i.e. place a '#' symbol *before* the code) as this will stop the 'Hello World!' text from appearing on the device.

Showing a singular heart is pretty simple (and boring), so let's create a spinning arrow instead that will utilise the built-in function, `forever`, and a function we will create ourselves.

Firstly, delete all the code you currently have, then find (and *select*) the `basic.forever` command like what was shown above for `show_icon`. You should now be presented with what is shown in figure 18.

```
1   def on_forever():
2       pass
3   basic.forever(on_forever)
4
```

*Figure 18: The code provided when the 'basic.forever' function is selected*

The first word on line 1, `def`, is a *keyword* that means to define a function with the name of `on_forever`. The next line (line 2) shows the *body* of the function which is what houses the code you want to execute when the function is used (the code must be indented to be valid Python syntax). Line 3 shows the *calling* (or use) of the `on_forever` function in a `basic.forever` statement which essentially is an infinite loop (i.e. it loops forever). It is important to note that line 3 is **not** a part of the `on_forever` function (as that is defined on lines 1 & 2).

Currently, these code statements do nothing as the *pass* instruction is just a placeholder used to allow empty functions in the code (as without it would cause an error), so let's change this.

In the function, put '`basic.show_arrow(ArrowNames.NORTH)`' in place of the `pass` statement on line 2 (you should now see an arrow pointing upwards on the LEDs). To create an illusion that an arrow is spinning in a clockwise direction, add more arrow statements to your function. Do this now.

To make this slightly more interesting, let's make it so the arrow randomly stops spinning. To do this, we will need to utilise the built-in Python library, `Math` (see figure 19).

```
1   def on_forever():
2       random_number = randint(1, 4)
3       basic.show_arrow(ArrowNames.NORTH)
4       basic.show_arrow(ArrowNames.EAST)
5       basic.show_arrow(ArrowNames.SOUTH)
6       basic.show_arrow(ArrowNames.WEST)
7
8   basic.forever(on_forever)
```

*Figure 19: Using the Math.randint() function*

What happens on line 2 is, the `randint` function generates a random integer (whole number) between 1 and 4 (including both) and stores that in the variable, `random_number`.

To stop the arrow from spinning, we need to use `if` statements to control the flow of execution (see figure 20).

```
1   def on_forever():
2       random_number = randint(1, 4)
3       if random_number == 1:
4           basic.show_arrow(ArrowNames.NORTH)
5       elif random_number == 2:
6           basic.show_arrow(ArrowNames.EAST)
7       elif random_number == 3:
8           basic.show_arrow(ArrowNames.SOUTH)
9       else:
10          basic.show_arrow(ArrowNames.WEST)
11
12  basic.forever(on_forever)
```

*Figure 20: Using if statements to control execution flow*

Have a look at the LEDs, what is happening?

Hopefully you noticed the arrow pointed in random directions and didn't actually come to a stop. To fix this, we can use a `for` loop instead of the `basic.forever` statement (see figure 21).

```
# Our function that displays an arrow
# based on a random number
def on_forever():
    random_number = randint(1, 4)
    if random_number == 1:
        basic.show_arrow(ArrowNames.NORTH)
    elif random_number == 2:
        basic.show_arrow(ArrowNames.EAST)
    elif random_number == 3:
        basic.show_arrow(ArrowNames.SOUTH)
    else:
        basic.show_arrow(ArrowNames.WEST)

# This is a for loop that will loop twice
# (meaning the arrows should spin clockwise
# twice)
for loop_counter in range(2):
    basic.show_arrow(ArrowNames.NORTH)
    basic.show_arrow(ArrowNames.EAST)
    basic.show_arrow(ArrowNames.SOUTH)
    basic.show_arrow(ArrowNames.WEST)

# Calling our function so the final arrow
# is displayed (i.e. what the spinning
# arrow would stop at)
on_forever()
```

*Figure 21: The code required to make the arrow spin twice before stopping in a random direction*

The first part of this code is the function we made earlier (this displays an arrow in a random direction). The second part makes the arrow spin, and the final part makes it stop by calling the function you made (i.e. part 1). You should now see the arrow do just this, spin twice and then stop in a random direction (feel free to ask for help if you encounter any difficulty).

## Guess the Dice Roll game

Now that you know the basics, have a go at completing the simple game, 'Guess the Dice Roll'. How the game works is the player must select a number from 1 to 6 (inclusive) and then roll the dice. If they guess correctly (i.e. the result of the dice roll is the same as the number they chose), they have won the game and are presented with a smiley face, else, they have lost the game and are presented with a sad face. The code file (has a `.hex` extension) is on the Slack channel; you will need to download this and then use the 'Import' function on the MakeCode editor home page (feel free to ask staff for assistance). Have a read through of the code to familiarise yourself with it before continuing.

If you have had a quick play with the Micro:bit, you should have noticed that the buttons (when pressed) display a letter ('A' for left, 'B' for right), and when you shake it, dice faces appear but nothing really happens. This is what you're going to fix.

To allow the player to be able to choose a number with the left and right buttons, we need to add functionality to the `on_button_pressed_a` function as currently it just displays an "A" when pressed (see figure 22).

```
1   # this is the function that is called when
2   # the left button is pressed
3   def on_button_pressed_a():
4       basic.show_string("A")
5
```

*Figure 22: The on_button_pressed_a function*

As this button is on the left, it makes sense for it to *decrement* the player's guess by 1 each time it is pressed. To do this we need to replace line 4 with '`guess = guess - 1`'. Although this does work, it's technically not correct as the variable, `guess`, is actually a *global* variable (meaning it can be accessed/used from anywhere within the file) so we need to state that first (see figure 23). Feel free to ask staff for clarification and/or further explanation.

```
1   # this is the function that is called when
2   # the left button is pressed
3   def on_button_pressed_a():
4       global guess
5       guess = guess - 1
```

*Figure 23: Declaring guess as a global variable*

It would be a good idea to display the guess on the LEDs so the player knows what they have selected. This can be done using a function from the basic library (hint: you've used the string, icon and arrow variants previously). Once done, do this for the other button but *increment* the value instead (i.e. `guess = guess + 1`).

Now you should be able to select an integer, but when you shake the Micro:bit, it always ends on a dice face of 6. This is because it is hardcoded in the `on_gesture_shake` function. To fix this, follow the exercises listed below.

**Exercises:**

1. Generate a random number and store it in a *global* variable called `dice_value`
2. Once the dice has rolled, display `dice_value` as a dice face on the LEDs (see figure 24). Hint: you may need to use a series of `if` statements and a delay

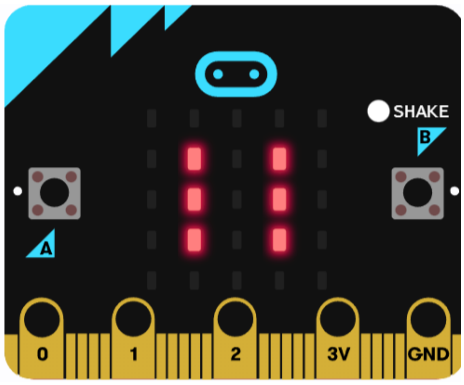


*Figure 24: Displaying a dice face of 6 on the LEDs*

3. Display a smiley face icon if the player's guess was correct (i.e. `guess == dice_value`) else a sad face icon if the player's guess was incorrect (i.e. `guess != dice_value`).

## Summary

Today we have covered the basics of programming Micro:bits with MicroPython. Hopefully through the simple exercises you have developed your understanding of MicroPython (and Python!) along with Micro:bits. Next week we'll use Micro:bits in conjunction with the Maqueen robots to design, and develop driving cars.

## Advanced exercises

Have you already finished the previous exercises and want to tackle a challenge? Currently the game allows any integer to be selected, although the only numbers on a 6 sided dice are 1, 2, 3, 4, 5 & 6 so you should implement input checking to prevent a player selecting an invalid number. When the dice is rolled, it only rolls once before displaying the resulting face which isn't really realistic as it is unlikely to only roll once every time, therefore you should make it roll a different amount each time (hint: random number generation).

Those are a few tasks to get you started, but once you have finished them also, have a look at doing these:

1. Implement a scoreboard system using the **serial port** where the score is formatted nicely (or as nicely as you can with characters, see figure 25 for an example).

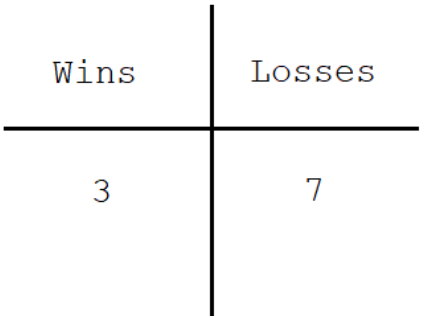


*Figure 25: Example scoreboard*

2. Implement a second player (this is a substantial task).The game would still play as normal but with an additional player, so they would both select a number and then one would roll the dice (up to you who does this) with the results of that being added to the scoreboard.

## Useful Resources

- BBC Micro:bits Introduction page: https://microbit.org/get-started/first-steps/introduction/
- BBC Micro:bits lesson:

https://www.bbc.co.uk/programmes/articles/2M3H2YpKLsw2W8fC2ycHYSR/welcome-to-the-micro-bit-live-lesson

- MicroPython Home page: https://micropython.org/
- MicroPython basics: What is MicroPython?: https://www.digikey.co.nz/en/maker/projects/micropython-basics-what-is-micropython/1f60afd88e6b44c0beb0784063f664fc
- What is MicroPython? (a short but useful blog post): https://www.kevsrobots.com/blog/what-is-micropython.html
- Embedded system facts for kids (it sounds lame, but it provides good explanations and examples): https://kids.kiddle.co/Embedded_system#:~:text=An%20embedded%20system%20is%20a,keyboard%20or%20monitor%20or%20mouse.
- The Importance of Error Checking in Software Development (we only talked about a small facet of error checking, this is more comprehensive): https://blog.intertecintl.com/the-importance-of-error-prevention-in-software-development