



Learn to Code - Python on a device

An introduction to programming a small device

The purpose of this guide is to help you get into the world of digital devices by showing you step by step how to create the software for a tiny electronic device.

The first chapter (1) is a pre-read for those who like to be (over)-prepared.

The second chapter (2) covers the prerequisites and setting up of the gear. No worries if you have a Windows 10 PC. And we will do together setup together when we meet.

After that begins the guide on actually creating some software. The fun stuff.

Contents

1	Background & Concepts.....	2
1.1	The Kit	2
1.2	The Tools.....	3
2	Set It Up.....	4
2.1	Connect	4
2.2	Install the editor and drivers.....	4
2.3	Let's start it up	5
3	Let's Code (almost) - Hello World!.....	6
4	Blink.....	8
5	LightsOn	10
6	ScrollSomeText.....	11
7	ControlTheScroll.....	13
8	MoveTheDotAround	16
9	LimitTheMovement.....	18
10	PaintSomeYellow	19

1 Background & Concepts

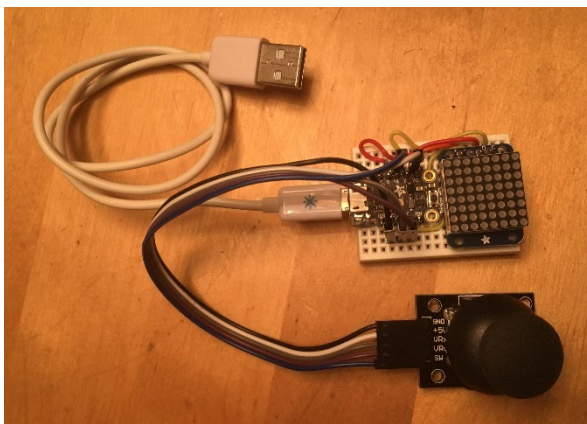
You might know a bit about computers and coding, or you might be a novice. In this guide we assume no previous knowledge at all.

To get started pretty quickly, we won't start from scratch, it will be a bit more like "connect the dots". The aim is to dip your toes, there are always bigger adventures for those who survive the small ones.

Some general ambitions / un-ambitions / prerequisites:

- Nothing is finished, we are always ready. This is work in progress. I learn to teach as you learn.
- This is 100% extra-curricular, although connections to "real work" are very welcome
- Fun fun fun!

1.1 The Kit



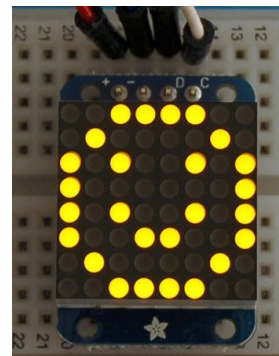
This is what the kit looks like when it is all connected. It fits in the palm of your hand. The two main features, are the LED matrix the tiny joystick.

This is stuff you can order for approximately 40€. It is also something you can put together by yourself. You can also come to the 'Maker Friday' events at Co Create and get some guidance.

In the table below, we take a closer look at the main components.



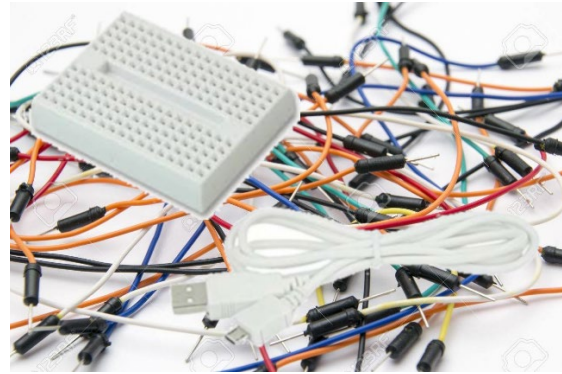
This is the [microcontroller](#), the "brain" of the system. It has a USB connector to connect to our PC and sockets to connect the other components



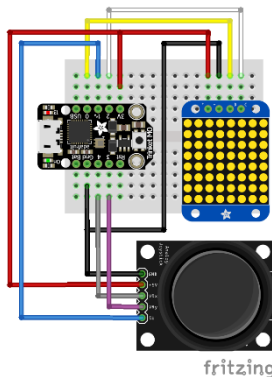
Here we have a [LED matrix](#). Basically 64 LED lights in 8 rows of 8. It is mounted on a small board that has a chip that makes it easier to control



To be able to interact with the kit, we have a tiny [joystick](#). Apart from moving it back and forth, up and down, you can also press it down as a button



The rest are what we call “electromechanics”: A plastic prototyping platform called “[breadboard](#)”, a USB cable to connect to the PC and small wires



The image to the left shows how the signals between the three hardware components are connected. All you need to build this kit is a small soldering iron and some wires. And of course some basic electronics building skills. If you come to the IKEA maker space at IKEA 2, we’ll help you with all that!

All the projects in this document use the setup described so far. And the goal is to get familiar with coding. But of course, there is the chance that you like it so much that you want to do more. Then it’s perhaps nice to know that there is a large community both online and IRL that can help you out. And there are lots of guides on how to build complete projects. [Like Jennifer’s Minecraft glove](#).

1.2 The Tools

Just by looking at the hardware components, you probably have an idea of what kind of behaviour you might want to create. Moving dots around on the screen with a joystick is pretty close to the definition of the first computer games, played on the first personal computers.

Creating that behaviour: moving the joystick controls what LEDs are lit, is what we call programming.

So how do we go about that? What tools do we need? There are several different ways to do this, the methods and tools selected here are chosen to create a “smooth experience” on most IKEA PCs.

1.2.1 The Language

We will be using Python. The design philosophy of Python emphasizes code readability, simplicity and scalability. Readability helps you understand the concepts, simplicity takes away some pitfalls and gives less quirks to learn, scalability means that the skills you get can be useful in for instance cloud apps.

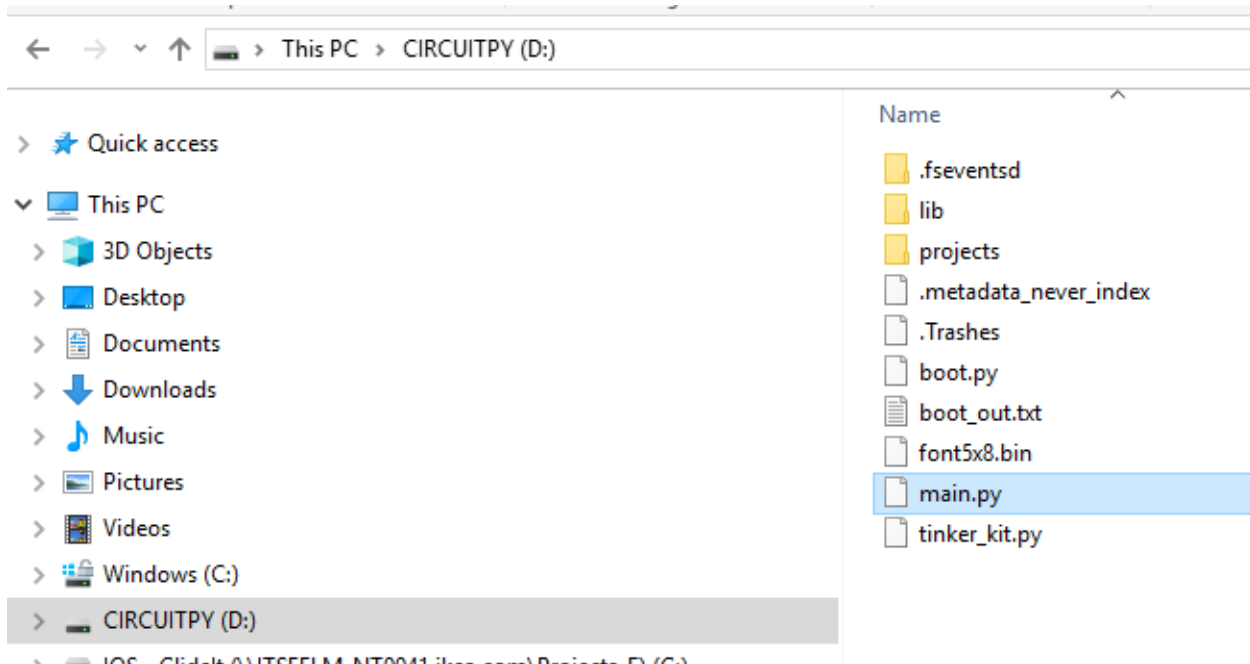
1.2.2 The Editor

Just like using special software for writing texts, we use special software to create software. The selection of software for coding is somewhat connected to which language we use. But there are often several options. We will be using an editor “Mu” which is good in at least three ways: it gives us guidance when writing Python, it comes together with tools to get the code onto our device, it also can be run without having to install anything (if you have a Windows 10 PC)

2 Set It Up

2.1 Connect

We start off by connecting the micro USB cable from the microcontroller to your PC. The microcontroller should act like a USB memory stick or a tiny external hard drive. On windows it looks something like this:



If you can see this, or something similar, you have a connection with the kit and you are able to move files from your computer over to the microcontroller to be able to change its behaviour. To make our lives easier and this learning a bit more fun, we will also setup an editor that helps us interact with the microcontroller and see a little bit more of what's going on "underneath the hood". We'll do that next.

2.2 Install the editor and drivers

2.2.1 IKEA PC with Windows 10

You can just bring your computer.

If you want to be extra prepared, you can download a [portable version](#) of the editor that we will use to code. This version can be used without installing anything on your computer. All we do is transfer a zip-file to your computer and unzip it to a folder of your choice.

2.2.2 IKEA Mac

We haven't tested this yet.

For the best experience, you probably would like to get [the Mac version](#) of the editor installed. You might need help from IT Helpdesk to get that done.

2.2.3 IKEA PC with Windows 7

For the best experience you need to install the [PC version](#) of the editor and [these drivers](#) you might need help from IT Helpdesk to get that done.

2.3 Let's start it up

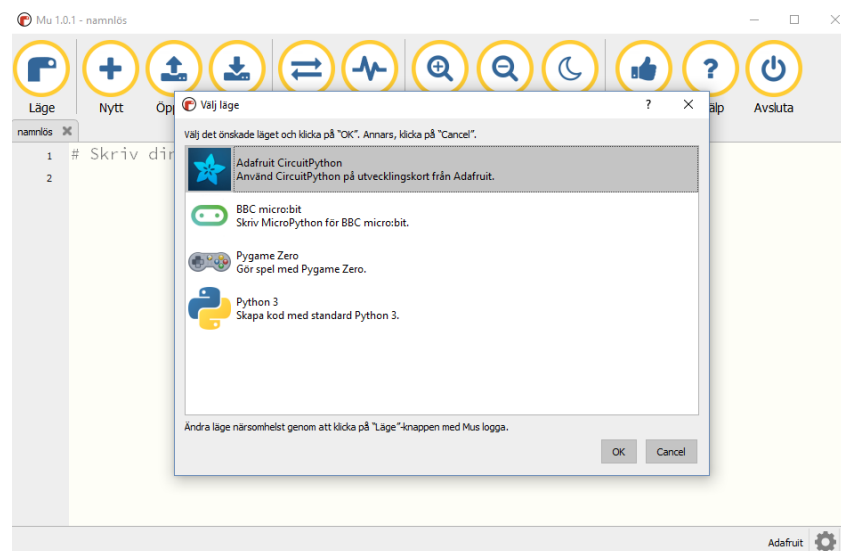
Now that we have the editor in place, we can launch it. Depending on what kind of computer you have, there are a few different ways to start it up. If you are using the portable version from 2.2.2, you simply browse to where you unzipped the files and locate the file “Launch Mu” or “Launch Mu.bat” and double click it.

SW > portamu_1.0.1_win64

Name	Date modified	Type	Size
bin	2018-11-21 16:21	File folder	
lib	2018-11-21 16:21	File folder	
pkgs	2018-11-21 16:21	File folder	
Python	2018-11-21 16:21	File folder	
_rewrite_shebangs.py	2018-09-30 01:07	PY File	1 KB
_system_path.py	2018-09-30 01:07	PY File	8 KB
Launch Mu.bat	2018-09-28 18:03	Windows Batch File	1 KB
LICENSE	2018-09-30 01:06	File	35 KB
Mu.launch.pyw	2018-09-30 01:09	Python File (no co...	2 KB
uninstall.exe	2018-10-04 21:23	Application	409 KB
win_icon.ico	2018-09-30 01:06	Icon	362 KB

For other types of computers, the application named “Mu” is launched in the same way as other installed applications.

In any case, when the editor application “Mu” is started, it should look something like this:



Your text might be in another language, but you should select the “Adafuit CircuitPython” mode and press OK. And if your kit is connected properly, you will see a brief message in the bottom part of the window that an “Adafuit CircuitPython Device” is found. If that is what you see, then we’re good to go.

3 Let's Code (almost) - Hello World!

Explorative Programming

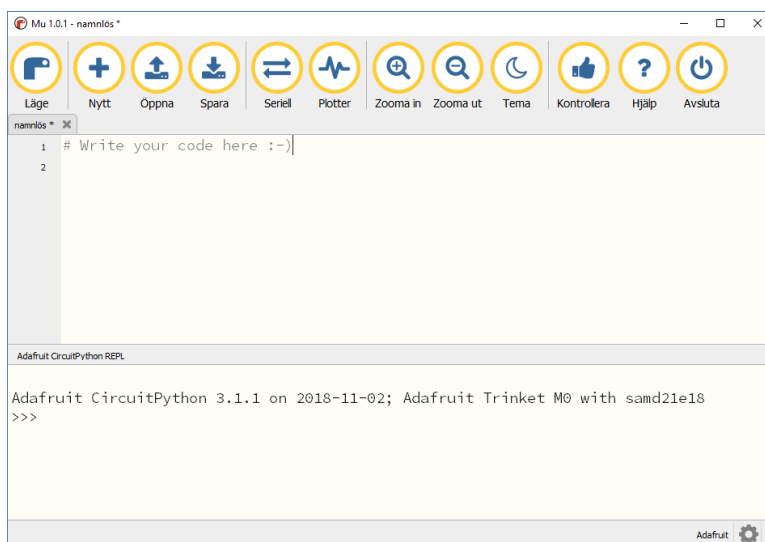
“Hello World” is a concept within coding. It typically refers to the first successful operation of a development environment: Getting the computer to just print a string of letters. Of course it could be any text. But “Hello World!” is a thing.

The title said “almost” coding. In this step we will not store any behaviour on our device, we will just interact with it and give it some commands that it will immediately perform (this is called [REPL](#)).

The first thing to do is to click on the button that looks like this:



The lower part of the window should show something like this:

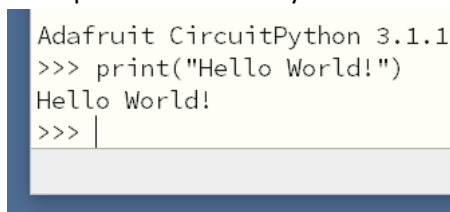


For the text and “>>>” to show, you might have to place your cursor in the window, and press a key. You shouldn’t need to press the combination of the “Ctrl”-key and “C” (CTRL+C) but if all else fails, try it.

Now comes your assignment: place the cursor at “>>>” (by clicking right beside it, to the right) and type the following text:

```
>>> print("Hello World!")
```

Then press the enter key. You should see something like this:





To get the microcontroller to print “Hello World!” we used what is called a *function*. In this case the name of the function was “print()”. The things we write between the parentheses are called the functions *arguments*. In this case there was only one argument and it was a *string*. That is what we call a string of characters, and that is a type of data (*datatype*) we will use later.

Now we will use the other most common datatypes; numbers. When it comes to numbers, python has two flavours: *integers* like: 0 1 2 42 64 and *floating point numbers* like: 0.0 3.14 6.67

Note the subtle difference between the two first numbers in each row: 0 vs 0.0 we’ll come back to that.

Using some of the *operators* that we know from maths, we’ll get acquainted with using our little device as a calculator. Type in the following, and press enter:

```
>>> 7 + 0.0
```

Notice anything interesting in the result?

Now try out some other operators: + - / * and the overachievers could try // and %

When entering commands at the “>>>”-prompt, you cannot use the mouse to navigate within the line. But you can use the right and left arrow keys. And you can recall previous lines with the up key.

One specific that I’d like you to try is a division of two integers that does not leave a remainder, such as:

```
>>> 12 / 4
```

Now the next step to explore is the basic concept of *variables*. A variable is a kind of named container of data. In Python you *define* a variable by *assigning* it a value. Try this out by creating a variable “name” and give it the value of your name, and then print it by sending it as an argument to the print() function:

```
>>> name = “Edith”
>>> print(name)
```

One thing to note is how we use quotes to make the distinction between the variable and function names and what is called a *literal string*. To check your understanding, try to predict the output of:

```
>>> print(“name”)
```

Now it’s time to go crazy! Play around with defining variables, performing some operations and printing.

```
>>> str1 = “good”
>>> str2 = “bye”
>>> print(str1 + str2)
```

What we have done in this chapter 3 is an interactive exploration of some of the basic concepts of coding: *variables*, *literal strings*, *datatypes* such as *integers* and *floats* and *strings*, *functions* and their *arguments*, how to perform some *operations* such as addition and subtraction and how to *define variables* by *assigning* them a value.

Wow! That was all for chapter 3

4 Blink

The equivalent of “Hello World!” when coding for small electronic devices (coding for *embedded*) is to blink a LED lamp. This will also be our first real programming.

The task is to make one of the LEDs in the led matrix blink as soon as the device is connected to power.

But first we have to setup the device to load the project file for this step 4. We do that by modifying a file on the device named main.py. So in the Mu editor, click the “open” button:



Then navigate to the file **main.py**, and select open. If the file main.py is hard to find, flip back to 2.1.

```
1 #####
2 #
3 # Select which project to load
4 #
5 #####
6
7 project = 4
8
9 #####
10 #
11 # Nothing to see below :-)
```

The file main.py has one single purpose; to let us select which project file to load. That is done by changing the value assigned to the variable “project” on line 7. We want to load the file for project 4, so we set it to 4.

When we’re done it should look like the picture to the left.

You can now save the file. The green LED located to the center of the microcontroller board should blink very briefly as the device restarts.

You can leave the “main.py” open. The next thing to do is to open the file we are going to do the actual coding in. All the files for the projects 4 through 10 are located in the folder “projects” shown in 2.1. So go ahead and open the file **Blink_4.py** in the Mu editor. It should contain the code below:

```
01 import board
02 import tinker_kit
03 import time
04
05 kit = tinker_kit.kit(board)
```

The lines 1, 2 & 3 all start with “import”. Those are references to pieces of code, or *modules*, stored elsewhere on the device. The code we are about to write depends on those modules to be able to work. You can think of them as drivers or concepts that the behaviors we want to code need. The first one; “board” supplies the basic concepts for all hardware, sending and receiving different kinds of signals. The next one “tinker_kit” handles the specific hardware we have, like the LED matrix and the joystick. The last one: “time” supplies methods to make the microcontroller pause for a certain amount of time.

The first line of code that actually does stuff, is line 5. It is used in all projects, so let’s look at it:

module
name
arguments

↓
↓
↓

kit =
tinker_kit.kit(board)

↑
↓
↓

return value
function call

This line calls the kit function in the tinker_kit module and returns an *object* by the name kit.

That’s a lot of “kit”. The only thing we need to know is that after this line is run, we have an object “kit” which supplies us with all the functionality that our little kit of hardware has. In this first project we will only use the LED matrix. The matrix is “inside” our kit object so we access it with kit.matrix .

I bet you are eager to see something happen, so it’s time to code!



The first thing to do is remove these two lines:

```
#remove this line and the one below before you start coding
print("*** Open Blink_4.py and start coding")
```

Then you can go ahead and type in lines 6 to 13 so that your file Blink_4.py looks like:

```
01 import board
02 import tinker_kit
03 import time
04
05 kit = tinker_kit.kit(board)
06
07 while True:
08     kit.matrix[0,0] = 0
09     kit.matrix.show()
10     time.sleep(0.5)
11     kit.matrix[0,0] = 1
12     kit.matrix.show()
13     time.sleep(0.5)
```

When you save the file, the microcontroller should reload and run the program.

Does it do what you expected it to? Does it seem to work? If not, there's help. Learning to code is actually 50% comprehending new abstract concepts, and 50% learning how to google error messages. Some code editors even present links to stackoverflow.com together with the error messages.

One thing that might give some issues, is the fact that not only the text, but also the whitespace is important. All the rows after "while True:" should be equally indented four steps. The indentation indicates that all these rows (row 8 to 13) should be seen as one block of code following the *loop* command "while True:". A loop command is used when you want to run some part of the code repeatedly. Like: "while the tank level sensor is below max, run the filling pump". In our blink code, we want to loop forever, so we use a condition that is always met: True.

Now let's look at the code that is repeated to make do the blinking. Lines 8 and 9 turns one LED on, line 10 pauses the microcontroller for 0.5 seconds, line 11-13 turns the same LED back on and pauses again.

To manipulate the matrix of LEDs we use two steps: we assign either 1 or 0 to matrix[0, 0], then we call a matrix.show(). One concept that we get along with the object "matrix" is what is called a *pixel map* or *pixmap*. It is a representation of all the LEDs in the display. To address the top left LED (or *pixel*) we call matrix[0, 0]. To address the pixel to the right of that, we call matrix[1, 0], and as you might have guessed, the one below that would be matrix[1, 1]. And the pixel in the bottom right is matrix[7, 7]. The two values within the square brackets can be seen as coordinates on the pixmap.

So if line 8 and 11 sets the pixel on and off respectively, why do we need to call matrix.show() ? The answer is that performing matrix[0, 0] = 1 only modifies the internal pixmap, while the call to matrix.show() transfers the modified pixmap from the microcontroller to the LED matrix. So that's all for step 4. Good job!

5 LightsOn

The next step after blinking a LED is reacting to a push-button. The task is to turn on one LED in the matrix by pushing the joystick button. (Check so you understand the button functionality)

One other feature of our kit is that the joystick can be used as a push-button. The way we access this functionality is by reading an *attribute* of our kit object. That is done on line 8 where you can see `push.joystick.push`. That means we are accessing the “push”-attribute of the joystick object. This attribute is 1 when the button is pressed and 0 otherwise.

So, just as before, edit the file `main.py` to select project number 5, then open the file `LightsOn_5.py`. You can complete the file with the code listed below.

```
01 import board
02 import tinker_kit
03 import time
04
05 kit = tinker_kit.kit(board)
06
07 while True:
08     kit.matrix[0,0] = kit.joystick.push
09     kit.matrix.show()
10     time.sleep(0.05)
```

Does it do what you expected?

If you get it to work, try to use your knowledge about the coordinates for the pixmap. Set some other pixels.

Could you invert the behavior? (There at least two types of possible inversions)

What if I tell you there is another function for the matrix object: `kit.matrix.fill()`, it sets all the pixels to a certain value. `kit.matrix.fill(0)` sets all pixels to 0, `kit.matrix.fill(1)` sets all to 1.

Knock yourself out! Then head on to project 6.

Or if you want, you could try out what this code does:

```
05 kit = tinker_kit.kit(board)
06 last_push = 0
07
08 while True:
09     if kit.joystick.push != last_push and kit.joystick.push:
10         kit.matrix.fill(not kit.matrix[0,0])
11         kit.matrix.show()
12     last_push = kit.joystick.push
13     time.sleep(0.05)
```

6 ScrollSomeText

In this next step we will use one more concept supplied by kit object. The new concept is `BitmapFont`. The purpose of this concept or *abstraction* is that we can call a function with some text as an argument, and have that text shown on a display like our LED matrix. So it handles the design of each character and placing them after each other and generating a pixel map.

The `BitmapFont` object is accessed by `kit.font`.

We use that object and call the `text(string, x, y, pixel)` function. It takes four arguments:

1. The text string to display
2. The X coordinate to place the text (0 = leftmost column of the matrix)
3. The Y coordinate to place the text (0 = top row the matrix)
4. The colour to draw the characters with, in our case pixels can only be on or off (1 or 0)

Let's try this out and edit the project file **ScrollSomeText 6.py** so it looks like:

```
01 import board
02 import tinker_kit
03 import time
04
05 kit = tinker_kit.kit(board)
06
07 kit.font.text('hej', 0, 0, 1)
08 kit.matrix.show()
```

You can play around with all four arguments for the `kit.font.text()` call. Not much is shown on the 8x8 pixels we have. To be able to display texts of any useful length we need to scroll. That means moving the text position gradually to the left. To do that we will use a new kind of loop. A *for loop*.

So for a while, we will leave the LED matrix, and look at loops and *lists*. A list variable holds any number of items of the datatypes we looked at before. The following code creates a list of numbers and then loops through that list and prints each number. (*The print function prints to the terminal window*)

```
08 list = [0, 1, 2, 3, 4]
09 for number in list:
10     print(number)
```

You can add this code to the file and run it. Do you get a print of all the numbers?

A for loop is used to perform an action a certain number of times, and the *iteration* variable (in the example above it's the variable "number") is only used to index each element in a list of some sort.

This can be simplified using the function `range(start, stop, step)` as shown in the following code:

```
11 for number in range(0, 5, 1):
12     print(number)
```

You can add this to your code and see if it prints the same thing.



If we want to scroll some text across our tiny display, we want to print the same string to the pixel map over and over again but with an offset that starts at zero and goes gradually downwards until we have shown the last letter in the string.

Now it's time to try out this new concept the "for loop". Remove the lines 7-12 you used to test the list and loop concepts, then edit the file so it looks like the listing below:

```
01 import board
02 import tinker_kit
03 import time
04
05 kit = tinker_kit.kit(board)
06
07 while True:
08     for i in range(20, -25, -1):
09         kit.matrix.fill(0)
10         kit.font.text('IKEA', i, 0, 1)
11         kit.matrix.show()
12         time.sleep(0.05)
```

Note that line 8 is indented 4 spaces, and lines 9-12 are indented 8 spaces. We actually have two loops here, one while loop we recognize from the previous project. Line 7's "while True:" makes everything below that go on forever. And our new loop concept on line 8: "for i in range (20, -25, -1):". That for loop can be read like this: do the lines below 45 times, the first time around the variable i will be 20, and for each time round it will be decreased by 1.

Does the code run? Does it look nice?


What happens if you replace 'IKEA' with a much longer string?

Apart from grasping the loop concept and avoiding typing errors, getting this to work requires us to find a good setting for the loop variable i. Try to get your longer string working by changing the range() arguments.

Can you describe to your buddy or instructor how the execution flows in this code?

7 ControlTheScroll

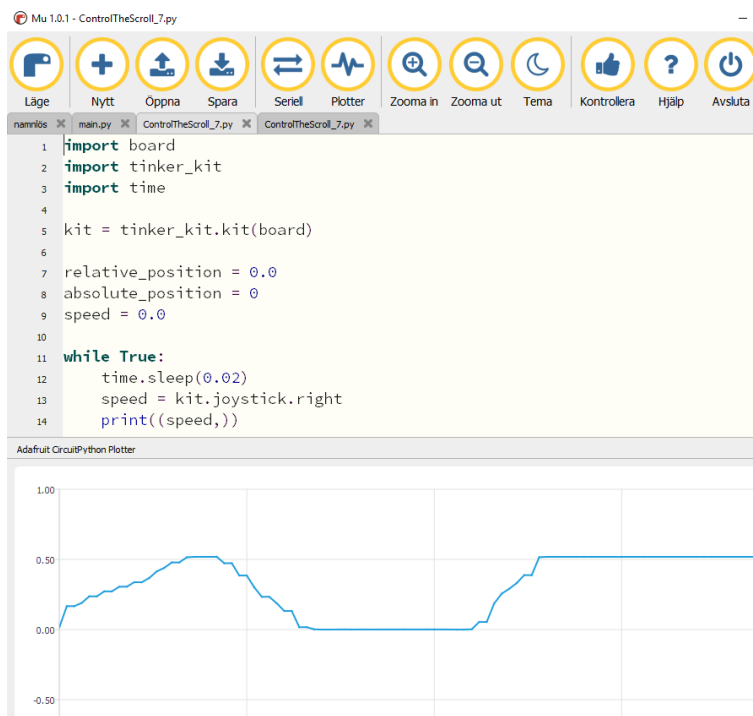
Our goal for this project is to control the scrolling of the text by moving the joystick from side to side. To do that we use the attribute `kit.joystick.right`. When we hold the joystick to the far right, this attribute will be 0.5 and when the joystick is the far left it will be -0.5.

To check that this indeed is so, we will use a tool in the editor called “plotter”. Click this: 

Then type in the listing shown below:

```
01 import board
02 import tinker_kit
03 import time
04
05 kit = tinker_kit.kit(board)
06
07 relative_position = 0.0
08 absolute_position = 0
09 speed = 0.0
10
11 while True:
12     time.sleep(0.02)
13     speed = kit.joystick.right
14     print((speed,))
```

Now run the code and move the joystick to the left and right, you should see something like this:





OK, so we have a variable by the name “speed” that we can adjust between -0.5 and 0.5 by moving the joystick. Looking back at the previous project we see that when we were scrolling the string “IKEA”, we had the position vary between -30 and 15. So our goal is to use the input signal from the joystick to set the position within that range.

The first step is to convert our speed into some kind of position, we add it to the variable named `relative_position`. Update the while loop to look like below:

```
11 while True:
12     time.sleep(0.02)
13     speed = kit.joystick.right
14     relative_position = relative_position + speed
15     print((relative_position,))
```

When this code runs, you should be able to use the joystick to make the relative position go up or down to any number. Only your patience sets the limit.

OK, it’s good that we now can go further than +/-0.5, but we would like to stay within the 45 pixel range we had in the last project. To achieve that, we use a mathematical function called the modulus operator. In Python written `%`. It gives the remainder of the division of two numbers. Some examples:

```
5 % 45 = 5
35 % 45 = 35
45 % 45 = 0
55 % 45 = 10
```

So with this knowledge, let’s see if we can limit `relative_position` to the range 0-45. Update the code to:

```
11 while True:
12     time.sleep(0.02)
13     speed = kit.joystick.right
14     relative_position = (relative_position + speed) % 45
15     print((relative_position,))
```

Save it and run it and see what happens when you move the joystick. Are you satisfied? Can you make `relative_position` go outside 0-45?

So now we have a variable that we can steer with a range of 45 steps. The only problem left to solve is that the range should be between -25 and 20 instead of 0 and 45. But that is easily fixed. Update to:

```
11 while True:
12     time.sleep(0.02)
13     speed = kit.joystick.right
14     relative_position = (relative_position + speed) % 45
15     absolute_position = int(relative_position) - 25
16     print((absolute_position,))
```

Save it and see if the plot behaves the way you want when you move the joystick.



If it all looks good, you can replace the `print()` function with the three lines that clear the pixmap, draws the text at the specified position on the pixmap and shows the pixmap on the led matrix. Your complete listing should look like this:

```
01 import board
02 import tinker_kit
03 import time
04
05 kit = tinker_kit.kit(board)
06
07 relative_position = 0.0
08 absolute_position = 0
09 speed = 0.0
10
11 while True:
12     time.sleep(0.02)
13     speed = kit.joystick.right
14     relative_position = (relative_position + speed) % 45
15     absolute_position = int(relative_position) - 25
16     kit.matrix.fill(0)
17     kit.font.text('IKEA', absolute_position, 0, 1)
18     kit.matrix.show()
```

OK, if that now is working. Do you think you could write this program with three less lines of code?

Consolidation like that is often done, It's a delicate balance to use few lines of code, without making the code hard to understand.

So now you're done with project 7! Nice work!



Blank Page

8 MoveTheDotAround

```
01 import board
02 import tinker_kit
03 import time
04
05 kit = tinker_kit.kit(board)
06
07 x = 3.0
08 y = 3.0
09
10 while True:
11     up = kit.joystick.up
12     right = kit.joystick.right
13     x = (x + right)%8
14     y = (y + up)%8
15     time.sleep(0.05)
16     kit.matrix.fill(0)
17     kit.matrix[int(x),int(y)] = 1
18     kit.matrix.show()
```

9 LimitTheMovement

```
01 import board
02 import tinker_kit
03 import time
04
05 kit = tinker_kit.kit(board)
06
07 x = 3.0
08 y = 3.0
09
10 def clamp(n, smallest, largest):
11     return max(smallest, min(n, largest))
12
13 while True:
14     up = kit.joystick.up
15     right = kit.joystick.right
16     x = clamp( (x + right), 0, 7.5)
17     y = clamp( (y + up), 0, 7.5)
18     time.sleep(0.05)
19     kit.matrix.fill(0)
20     kit.matrix[int(x),int(y)] = 1
21     kit.matrix.show()
```

10 PaintSomeYellow

```
01 import board
02 import tinker_kit
03 import time
04
05 kit = tinker_kit.kit(board)
06
07 x = 3.0
08 y = 3.0
09 x_old = 3.0
10 y_old = 3.0
11 paint = False
12
13 def clamp(n, smallest, largest):
14     return max(smallest, min(n, largest))
15
16 while True:
17     paint = paint or kit.joystick.push
18     up = kit.joystick.up
19     right = kit.joystick.right
20     x = clamp( (x + right), 0, 7.5)
21     y = clamp( (y + up), 0, 7.5)
22     kit.matrix[int(x),int(y)] = 1
23     if int(x) != int(x_old) or int(y) != int(y_old):
24         if paint :
25             kit.matrix[int(x_old), int(y_old)] = 1
26             paint = False
27         else:
28             kit.matrix[int(x_old), int(y_old)] = 0
29     kit.matrix.show()
30     time.sleep(0.05)
31     x_old = x
32     y_old = y
```