Buttons

So far we have created code that makes the device do something. This is called *output*. However, we also need the device to react to things. Such things are called *inputs*.

It's easy to remember: output is what the device puts out to the world whereas input is what goes into the device for it to process.

The most obvious means of input on the micro:bit are its two buttons, labelled `A` and `B`. Somehow, we need MicroPython to react to button presses.

This is remarkably simple:

```
from microbit import *

sleep(10000)
display.scroll(str(button_a.get_presses()))
```

All this script does is sleep for ten thousand milliseconds (i.e. 10 seconds) and then scrolls the number of times you pressed button `A`. That's it!

While it's a pretty useless script, it introduces a couple of interesting new ideas:

1. The `sleep` *function* will make the micro:bit sleep for a certain number of milliseconds. If you want a pause in your program, this is how to do it. A *function* is just like a *method*, but it isn't attached by a dot to an *object*.
2. There is an object called `button_a` and it allows you to get the number of times it has been pressed with the `get_presses` *method*.

Since `get_presses` gives a numeric value and `display.scroll` only displays characters, we need to convert the numeric value into a string of characters. We do this with the `str` function (short for "string" ~ it converts things into strings of characters).

The third line is a bit like an onion. If the parenthesis are the onion skins then you'll notice that `display.scroll` contains `str` that itself contains `button_a.get_presses`. Python attempts to work out the inner-most answer first before starting on the next layer out. This is called *nesting* - the coding equivalent of a Russian Matrioshka doll.

Let’s pretend you’ve pressed the button 10 times. Here’s how Python works out what’s happening on the third line:

Python sees the complete line and gets the value of `get_presses`:

```
display.scroll(str(button_a.get_presses()))
```

Now that Python knows how many button presses there have been, it converts the numeric value into a string of characters:

```
display.scroll(str(10))
```

Finally, Python knows what to scroll across the display:

```
display.scroll("10")
```

While this might seem like a lot of work, MicroPython makes this happen extraordinarily fast.

Event Loops

Often you need your program to hang around waiting for something to happen. To do this you make it loop around a piece of code that defines how to react to certain expected events such as a button press.

To make loops in Python you use the `while` keyword. It checks if something is `True`. If it is, it runs a *block of code* called the *body* of the loop. If it isn't, it breaks out of the loop (ignoring the body) and the rest of the program can continue.

Python makes it easy to define blocks of code. Say I have a to-do list written on a piece of paper. It probably looks something like this:

```
Shopping
Fix broken gutter
Mow the lawn
```

If I wanted to break down my to-do list a bit further, I might write something like this:

```
Shopping:
    Eggs
    Bacon
    Tomatoes
Fix broken gutter:
    Borrow ladder from next door
    Find hammer and nails
    Return ladder
Mow the lawn:
    Check lawn around pond for frogs
    Check mower fuel level
```

It's obvious that the main tasks are broken down into sub-tasks that are *indented* underneath the main task to which they are related. So `Eggs`, `Bacon` and `Tomatoes` are obviously related to `Shopping`. By indenting things we make it easy to see, at a glance, how the tasks relate to each other.

This is called *nesting*. We use nesting to define blocks of code like this:

```
from microbit import *

while running_time() < 10000:
    display.show(Image.ASLEEP)

display.show(Image.SURPRISED)
```

The `running_time` function returns the number of milliseconds since the device started.

The `while running_time() < 10000:` line checks if the running time is less than 10000 milliseconds (i.e. 10 seconds). If it is, *and this is where we can see scoping in action*, then it'll display `Image.ASLEEP`. Notice how this is indented underneath the `while` statement *just like in our to-do list*.

Obviously, if the running time is equal to or greater than 10000 milliseconds then the display will show `Image.SURPRISED`. Why? Because the `while` condition will be False (`running_time` is no longer `< 10000`). In that case the loop is finished and the program will continue after the `while` loop's block of code. It'll look like your device is asleep for 10 seconds before waking up with a surprised look on its face.

Try it!

Handling an Event

If we want MicroPython to react to button press events we should put it into an infinite loop and check if the button `is_pressed`.

An infinite loop is easy:

```
while True:
    # Do stuff
```

(Remember, `while` checks if something is `True` to work out if it should run its block of code. Since `True` is obviously `True` for all time, you get an infinite loop!)

Let's make a very simple cyber-pet. It's always sad unless you're pressing button `A`. If you press button `B` it dies. (I realise this isn't a very pleasant game, so perhaps you can figure out how to improve it.):

```
from microbit import *

while True:
    if button_a.is_pressed():
        display.show(Image.HAPPY)
    elif button_b.is_pressed():
        break
    else:
        display.show(Image.SAD)

display.clear()
```

Can you see how we check what buttons are pressed? We used `if`, `elif` (short for "else if") and `else`. These are called *conditionals* and work like this:

```
if something is True:
    # do one thing
elif some other thing is True:
    # do another thing
else:
    # do yet another thing.
```

This is remarkably similar to English!

The `is_pressed` method only produces two results: `True` or `False`. If you're pressing the button it returns `True`, otherwise it returns `False`. The code above is saying, in English, "for ever and ever, if button A is pressed then show a happy face, else if button B is pressed break out of the loop, otherwise display a sad face." We break out of the loop (stop the program running for ever and ever) with the `break` statement.

At the very end, when the cyber-pet is dead, we `clear` the display.

Can you think of ways to make this game less tragic? How would you check if *both* buttons are pressed? (Hint: Python has `and`, `or` and `not` logical operators to help check multiple truth statements (things that produce either `True` or `False` results).